How to test your JavaScript

Emil Wall

July 18, 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

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.

10.3 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[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.

10.4 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.

10.5 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.

10.6 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.

10.7 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.

11 Appendix

11.1 Transcript of Interview with Johannes Edelstam

Vad är dina erfarenheter av hur folk testar sitt JavaScript?

För 1,5-2 år sen, höll jag i en träff om testning av JavaScript. Jag hade handuppräckning och frågade "Hur många testar sitt JavaScript?". Då var det bara tre personer som hade gjort det överhuvudtaget. Nu, nästan två år senare, är i princip alla händer i luften. Trenden har gått från "Vad konstig du är som testar ditt JavaScript" till "Vad kör du för tools" på bara ett och ett halvt år.

Vad tror du har lett till det?

Verktyg, dels att de har blivit bättre och att det finns fler exempel på hur man gör. Folk var inte emot testning eller test-driven utveckling, många Ruby-utvecklare hade redan positiva erfarenheter av att testa, men hade inställningen att JavaScript var för inriktat på gränssnitt för att det ska gå att testa det.

Det tycker jag att man fortfarande kan se lite grann.

Samtidigt är det konstigt att säga så, för i ett Ruby-projekt testas saker och ting rigoröst, trots att det är en massa gränssnitt som du testar igenom. Det handlar bara om att det är dålig tooling, inte att det inte behövs göras.

En annan aspekt kan vara att folk helt enkelt skriver kass kod. Om man i ett läge inte vet hur man ska testa en viss sak så kan det vara så att det inte går för att koden inte har skrivits på ett sätt som gör den testbar.

Jag funderar på om att använda ramverk som strukturerar upp koden är en lösning för att faktiskt kunna testa kod bättre.

Vad tänker du på för ramverk?

Det jag hittills hunnit skaffa mig erfarenhet av är AngularJS, som har vyer och controllers. Controllers är enkla att testa.

Precis.

Sen det här spelet som jag tänkte köra en workshop om framöver, det kanske jag inte har berättat något om. Jag tänkte köra med ett arkadspel, Asteroids, och TDD:a lite på det.

Schysst.

Det är inte uppbyggt kring något ramverk, men är uppdelat i klasser och känns i allmänhet som rätt så schysst kod. Då går det att testa. Det fanns inga tester, så det har jag lagt till nu i efterhand (vilket har varit lärorikt). Jag tror att om man inte är såpass duktig som han som skrev det var, och faktiskt är så disciplinerad att man ser till att dela upp så att det bildas klasser med metoder som går att testa, så blir det kaos.

Verkligen.

Det där med Ruby vs JavaScript har jag sett innan också, jag inleder min rapport med det just nu, ett liknande citat som det du just sa, bara att det var från när det fortfarande bara var tre händer.

Det är vanligt och gäller överallt, folk gillar att hitta ursäkter till varför man inte ska testa grejer. Det är en liknande paradox som den att man tror att man blir så effektiv när man multitaskar, man luras att tro att man begränsas av testerna. Det som folk ofta menar när de säger att de begränsas av testerna är att de i själva verket inte riktigt vet vart de ska. Om man inte vet vad det är man ska bygga och hur allting ska fungera så tar det väldigt lång tid om man ska test-driva någon form av lekstuga. Det blir rätt stökigt, så där handlar det om att hitta bra egna principer för hur man tar sig fram till den punkt då man vet vad något ska bli. När man väl har en klar bild av vart man vill så är det ofta ganska lätt att börja skriva tester och det väcker bra tankar som man inte har stött på när man tagit sig fram till sin målbild.

Det som ofta blir när man skriver tester är att man lägger mycket av sin och fokus i att skriva testerna på ett bra sätt och känslan blir att man inte skriver lika mycket av produktionskoden. Vilket är en bra sak! För då ägnar man en massa tid åt att fundera på problemet och hur man ska boxa in det. Det leder till eleganta och små lösningar, för att man lyckats skapa sig en uppfattning om vad problemet är.

Framförallt blir man bättre på att göra begränsningar, om man kodar utan tester så är det lätt att man kommer på att en viss funktion skulle kunna göra flera saker, medan om man skriver tester för koden så är det enklare att inse att en extra parameter kan innebära att det behövs dubbelt så många tester. Just i JavaScript är det ett vanligt misstag att frångå single purpose-principen och det kan man alltså undvika med hjälp av tester.

Ja, i JavaScript är det ju enkelt att skriva funktioner som tar olika antal parametrar och bara kör på oavsett hur de anropas.

Jag skulle säga att jag testar betydligt mer av just logik-kod än till exempel grafer. Det är mest för att de ändras snabbt och risken för fel är lägre. Det är värre att ha ett konsekvent logik-fel än om en stapel blir lite för kort.

Du tänker på Tink?

Ja, precis. Det är samma mönster som när jag testdriver Ruby-kod. Man driver beteende, inte utseende. Sen tror jag också att det har att göra med att det är dåliga tools, det är omständigt att skriva hållbara och värdefulla för att en svg har rätt parametrar. I situationer där något kan vara rätt ändå, fast det inte är precis som man skrev, så uppstår bräckliga tester om man inte har tagit hänsyn till detta när man skrivit sina tester. Dåliga tester kännetecknas av att man behöver ta bort dem för att kunna komma vidare i utvecklingen. Det är en svår gränsdragning var man ska lägga sin energi.

Jag läste ett inlägg på twitter alldeles nyss av Kent Beck, om att han hade spenderat 6 timmar på att skriva ett test. Han hade researchat och konstaterat att det var ett komplicerat domän och någon påpekade att detta var anledningen till att folk inte skriver tester, att andra hade gett upp här. Kent Beck kontrade med att om han inte vet hur han ska skriva testet så vet han verkligen inte hur han ska skriva koden.

Exakt! Det där är jätteintressant, jag tror att folk hänger upp sig för mycket på att det är att skriva testerna som är svårt. Om man sätter sig och skriver testerna först så är det där man springer på problemet. Någon annan hade säkert lagt samma sex timmar fast på att stirra på produktionskod.

Sen kände jag att det har lite med vana att göra också. Nu när jag skrivit tester till det här spelet på senare tid, så går det mycket fortare att se vad som går att testa och hur testkoden kan skrivas. I början blev jag frustrerad över de nybörjarmisstag jag gjorde.

Det är en viss startsträcka så att det tar längre tid att komma igång, vilket avskräcker folk. Det där har blivit mycket bättre genom verktyg som Yeoman och Brunch, där man får projekt med tester och allting uppsatta, det är bara att börja skriva ungefär som i Ruby on rails. Sådant gör att folk blir mer vänligt inställda till tester.

I början när jag kollade på det här så var det väldigt experimentellt att överhuvudtaget köra tester i terminalen. Innan PhantomJS var bra så fanns det headless drivers som var svåra att använda.

Hur bra är PhantomJS nu?

Nu är det bra, jag kör testerna i princip jämt genom PhantomJS. Det kunde fortfarande vara bättre, men för det mesta så är det nog testramverken som inte använder det på rätt sätt. Vissa test failures måste man fortfarande dra upp i browsern för att få bra felrapportering. Vi kör våra tester i PhantonJS i byggen på Jenkins, så det är en del av byggflödet.

Det jag har gjort nu är att jag har använt JsTestDriver, det känns som att det kan vara knepigt att få in det i Jenkins.

Det är det säkert, men där handlar det om att man måste skilja på tester som testar ren logik där värdet av att testa i en massa olika webbläsare kanske inte är sådär gigantiskt stort.

Nej, mina tester skulle jag kunna köra headless.

Exakt, och då kan man lika gärna köra dem i något abstrakt som till exempel phantom, för du får samma resultat och det är mycket smidigare. Jag har i och för sig inte provat att sätta upp JsTestDriver och har nog aldrig kört det faktiskt och har hållt mig till andra verktyg.

Det är egentligen bara att man har en jar-fil som man använder som lokal server och så skickar man requests dit. Det är smidigt att använda men oklart hur det skulle funka med Jenkins, eftersom man behöver hooka upp webbläsare mot servern för att ha någonstans att köra testerna. Att få det att ske automatiskt är kanske inte trivialt. Dessutom så har det varit lite buggigt, att om något test handlar i en oändlig loop så har jag varit tvungen att stänga ner den fliken i webbläsaren, starta om servern och återansluta. När jag satt med Mac innan så var jag tvungen att starta om servern varje gång datorn gått i vänteläge, typiskt sådant som man som användare blir frustrerad av.

Sånt kan vara väldigt irriterande och det är sådana saker som gör att folk inte vill hålla på med tester.

Du hade använt Mocha rätt mycket, det tänkte jag titta på sen.

Mm, vad kör du nu?

Nu kör jag JsTestDriver och Jasmine.

Det är väldigt likt, Jasmine och Mocha.

Så du kör Mocha istället för Jasmine?

Ja.

Är Mocha en test driver? För Jasmine kan väl inte köra testerna i en webbläsare, utan är i första hand ett assertion framework?

Det var så länge sen jag körde det, men ja man kanske behöver ha en driver till det för att kunna köra i webbläsaren.

Jag har uppfattat det som att Mocha är lite mer kompetent som driver och liknande.

Ja, det är det. Det är enkelt att dra upp tester i webbläsaren om man vill ha det.

Jag gillar nästan bättre att skriva tester i JavaScript än i Ruby, det finns så mycket praktiska språk-features.

Att manuellt kunna definiera om en funktion.

Precis, och hur lätt det är att göra nya fake-objekt, det är så enkelt. Ofta blir mocknings- och stubbningsramverk ganska överflödiga, förutom till att göra vissa call-assertions som man använder det till, men i övrigt så försöker jag att alltid hålla mig till fakes som är skrivna för hand för att det blir lättare för någon annan som sätter sig in i det.

När nästa person tittar på koden så är det en fördel om den personen inte behöver lära sig ett helt ramverk eller sätta sig in i hur DSL:er är uppbyggda bara för att förstå testerna. Om det bara är vanlig JS-kod så blir det lätt för folk att förstå vad det är som händer, så på det sättet är JavaScript väldigt väl lämpat för att skriva tester till.

Sen kan det vara så att hela Node-rörelsen också har bidragit till att det testas mer.

Det har väl också ändrat vad JavaScript används till, rätt mycket?

Ja, det har det garanterat.

Argumentet "det är gränssnitt så det är svårt att testa" håller inte riktigt. Jag fokuserar på klientsidan nu, för att jag var tvungen att avgränsa på något sätt och för att det känns som att problemen ligger på gränssnitt snarare än Node och liknande.

Om vi tar ett login-formulär som exempel, med en vy-klass som du kan anropa render på. Alla dependencies, inklusive templates, som den använder, laddas via RequireJS. Så i mitt test så använder jag RequireJS för att få in den filen och då är den redo att köra och det går att anropa render. Här gäller det att ha sina dependencies rätt, för om den vet om en massa saker utanför så uppstår situationer där man behöver mocka väldigt mycket.

Det är ytterligare en fördel med tester, att det blir tydligt vad som krävs för att kunna använda ett visst objekt och man blir uppmärksam på dependencies som inte ska vara

där. Bara genom att anropa render och göra en submit på formuläret så hanteras det i vyn och man kan testa att rätt tjänster anropas, m.m. Igen så handlar det om att man måste hitta bra modularitet i hur man lägger upp det, så att man inte gör anrop direkt till ett API från en vy, utan ha ett servicelager emellan så att man kan swappa ut testerna och kolla att den går mot servicelagret. Det är sånt som folk fastnar på, att man har för dålig separation.

Vid själva utvecklingen av det här, vad skriver du tester för först då? Skriver du tester för vyerna innan du har de här tjänsterna igång?

Det beror på, det typiska man vill testa här är en validering. I det fallet skulle jag sätta upp formuläret först och sen fixa ett failande test som letar efter en validering som inte finns. Först därefter skriva själva valideringen. Man får ta det till en bra nivå, det känns inte jättevärt att testdriva varje steg av boilerplate-grejer när man vet exakt vad det ska bli och det nästan är copypaste-varning på det man gör. Då ger det mer att testdriva när man börjar komma till nästa nivå. Det känner man själv, när man måste börja tänka efter.

Det gäller att höra de varningssignalerna.

Ja, det här tror jag också är det svåra med testning, att det handlar om avvägningar. Det är som allt man gör med kod, att man måste ha med sunt förnuft när man gör det. Folk gillar att sätta upp principer och det tror jag att andra blir provocerade av. Principer behöver användas i sitt sammanhang och man får inte bli för religiös kring dem. Man behöver känna när det ger något och när det är i vägen.

Mm, det jag hoppas få som slutsats i mitt examensarbete är något i stil med "om du är i den här situationen, gör så här". "Om du gör en sån här app, använd ett ramverk av den här typen".

Ja, men då tror jag tyvärr att du kommer att landa i att "är du i den här situationen, använd ditt sunda förnuft".

båda skrattar

För det där tycker jag att man ser rätt tydligt, som nybörjare så behöver man kunna reglerna för att bryta mot dem. De som är mer seniora kan reglerna och är inte så rädda att bryta mot dem när de är i vägen. Man måste vara väldigt pragmatisk med sina principer. Det är om man inte är det som det uppstår konflikter av typen "att skriva tester tar så mycket tid från det riktiga jobbet". Om det inte ger dig något värde att skriva tester, gör inte det då! Det är ingen idé att göra det om du inte tror på det. Om du har något bättre sätt, go ahead! Om 20 år så är det kanske inte TDD som är grejen längre för att det kommit fram mycket smartare sätt och det kanske är ditt sätt.

Man måste se testning som ett verktyg man har, något man kan ta fram och stödja sig på i vissa situationer. Man kan också hamna i att man blir så inne i BDD och TDD att man slutar tänka. Att tänka på arkitektur blir någonting fult, man ska skriva tester först, det är det enda man får göra först. Att stå vid en whiteboard och rita blir plötsligt att gå händelserna i förväg och det tror jag är farligt.

Man ska inte glömma att precis som ens kod lätt blir ens baby så kan ens tester också

bli det. Man kan skriva tester som man är riktigt nöjd med och då kommer det att ta emot att kasta den koden. Det ska man akta sig för, att bli för kär i sin kod. Då måste man ibland tänka till lite först, innan man bara sätter sig och hamrar igång. Där måste man hitta sin avvägning i vad man tror på.

Jag funderar nu på vilken nivå man väljer att hålla testerna på. De flesta av testerna jag skrev till asteroids-applikationen återspeglade hur jag förväntade mig att programmet skulle bete sig. Sedan var det någon kod som var rätt dåligt modulariserad och då hamnade testerna väldigt mycket på detaljnivå och det blev väldigt många tester som var svåra att överblicka. Testerna kan ju bli som en kravspecifikation och det är svårt att läsa dem som en sådan när det blir för mycket. Jag funderar på hur man egentligen ska tänka, ska man hålla en jämn nivå? Om man börjar inse att ok, nu skriver jag väldigt mycket tester på detaljnivå, jag kanske borde slänga de här testerna och skriva om koden istället?

Ja, egentligen så tror jag att det där är sådant som man verkligen måste känner efter vid varje situation. Om man till exempel har en jättekomplicerad och ointuitiv prisberäkning, för all del, skriv hur mycket tester du vill, men jag ser inget egenvärde i att ha en 100-procentig test coverage, för det slinker ändå igenom saker. Det finns alltid scenarios som du inte har tänkt på. Det är förmodligen det fallet du inte har skrivit ett test för som skapar problem, och det är för att du inte kom på det när du skrev testerna. Det finns ingen större mening med att försöka trycka in alla fall, utan det viktiga är att man skapar en känsla av att täcka in det ganska bra, att man har rimliga fall och framförallt tänka på den som kommer efter.

Om du öppnar en fil med 700 rader testkod så kommer du inte att engagera dig på samma sätt för det är för svårt att förstå sig på all den koden. Där måste man verkligen använda sunt förnuft och det var också någon tweet jag läste att om du ber en utvecklare review:a 10 rader kod så kommer du att få hur mycket feedback som helst, om du ber en utvecklare review:a 500 rader kod så kommer hen att säga att det ser bra ut. Så är det ju, det är för mycket att sätta sig in i och bland annat därför så ska man inte vara rädd för att ta bort tester. Kasta gamla tester som inte behövs längre.

De bästa utvecklarna som jag har jobbat med har haft ett netto att det försvinner kod med varje commit, så att det kommer till grejer men samtidigt så försvinner kod. Det är imponerande, och har att göra med att de är bra på att se vad som inte behövs och hur man kan tänka om, ibland genom att ändra på vissa grundantaganden för att få koden att bli renare. Med det vill jag inte säga att man ska gå över till någon sorts kodgolfande, men det är ändå intressant att det kan bli resultatet.

Vi kom in lite på att lägga till tester till kod som redan finns, vilket jag upplever är rätt så jobbigt och förstår att du tycker är lite dumt. Men är det inte också risk för dubbelarbete om man tänker att man bara skriver tester för ny kod och ersätter gammal kod som man inser inte funkar så bra?

Det beror på vad den gamla koden är i för skick.

Det kan vara lätt att tro att den är i sämre skick än den egentligen är. Om man tar över kod som någon annan har skrivit.

Javisst, om jag skulle ersätta ett banksystem, då är det klart att jag skulle börja försöka skriva tester för precis den lilla biten man ska in och härja i. Men med ny kod så tänker jag också som så att om man ska lägga till någonting nytt eller bara är inne och roddar, då kan man se till att lägga till tester för just det man gör. Då gör man det som en del i det nya man skriver. Det är svårt, för det är inte alltid det går. En del gammal kod är så ruttet skriven att testerna ändå blir så komplicerade att man måste mocka precis allting för att ens komma till objektet och logiken där bygger på så krångliga kedjor av anrop att testerna blir bräckliga och svåra att förstå oavsett. Frågan är om det ger så mycket då. Å andra sidan, i fallet med banksystemet, så måste det bli rätt. Då måste man hitta ett sätt att verifiera att systemet fortfarande beter sig rätt. Jag har nog aldrig varit i ett sådant fall att jag suttit med något så kritiskt som ett banksystem eller en cancerlaser eller liknande där jag behövt ersätta gammal kod, men det finns bra böcker på det där, som tar upp de situationerna.

De gånger det inte går att skriva meningsfulla tester så tror jag att det beror på att koden inte är skriven för att vara testbar. Oftast när folk pratar om legacy JavaScript så är det kod som ändå inte är banksystems-kritisk. Då skulle jag säga att en bättre approach är att testa allt nytt man skriver än att försöka täcka in allt gammalt.

Min tanke nu var att i min workshop/kompetensdragning så vill jag ge möjlighet att se om man har råkat ha sönder någonting gammalt och se den effekten av testerna. Jag tänkte ställa några frågor efteråt, ”hjälpte testerna er någonting?”

Det är nog bra, för det är ju också ett jättevärde med tester, när man släpper in nytt folk i ett projekt. Det är alltid så att folk tenderar att ha sönder grejer när de är inne och härjar i dem för första gången, vilket inte är så konstigt.

Och framförallt är rädda att ha sönder saker.

Ja, precis. Finns det då en bra testsvit så är det bara att köra. Jag tror att det är ett bra sätt att komma in i nya projekt också, genom att det finns en bra testsvit som man kan börja bygga vidare på så blir man tryggare i det man gör och kan imitera hur de tidigare testerna har skrivits. Då kan man lita på det man skriver, att det blir rätt.

Vinsten med att lägga till tester i efterhand så som jag har gjort är att jag verkligen fått lära mig, det känns nästan som att det är jag som har skrivit koden jag testat, trots att jag bara har varit och fingrat på några få rader i den.

Verkligen, man får en helt annan uppfattning för den. Men det är en utmaning att skriva bra kod som blir testbar. Sen det här problemet som du också var inne på, att det kan finnas integrationstestnings-problematiker, det kanske jag skulle vilja säga är det svåraste. Fallen då man har egna API:er. Hur man ska testa det rakt igenom.

Du skrev i ditt mail att du tyckte att det var knepigt att testa privata API:er för att de ändras så mycket.

Säg att jag integrerar mot Twitter eller någonting, deras API:er. Det är lätt att testa, genom att kolla att de anropas på rätt sätt. Det är väldokumenterade, stora, bra API:er. Det som är problemet är att hålla hastighet när man ändrar på sina egna API:er. Ett bra sätt att göra det internt är att du har ett test för någon typ av domän-modell som definierar vad som gäller för en viss sak, till exempel hur ett cykelhjul ska fungera. Då

har man en uppsättning tester som testar att ett objekt är ett giltigt cykelhjul, och så kan man köra samma tester på sin fake av ett cykelhjul för att verifiera att den också är ett giltigt sådant. Denna fake kan sedan användas i andra test, som gör saker med cykelhjul. Om API:et för cykelhjulet ändras så kommer testerna för fakeobjektet också att faila, eftersom samma tester används på det riktiga objektet som på fakeobjektet. Då uppdateras den tills den uppfyller testerna för att vara ett giltigt cykelhjul och då kommer de andra testerna som involverar fakeobjektet att börja faila.

Egentligen skulle man vilja göra ungefär samma sak med sina egna API:er, genom att med tester definiera vad som är ett giltigt respons. Då kan man ha samma tester till att kolla att API-endpointen ger ett giltigt respons och att ens fakerespons är ett giltigt respons. Problemet man ofta har är att man separerar det där, istället för att ha en delad testsvit. Det blir oerhört komplicerat, framförallt om man skriver i olika språk. Det är ett problem som jag vet att många har och som jag inte har sett någon riktigt bra lösning på.

Och det är lite mitt emellan Selenium och enhetstester då?

Ja, precis.

Hur stort är behovet av att ha de här testerna?

Tyvärr så är det stort. Dels är det något som ändras snabbt och så är det någonting som påverkas mycket. Det är ju egentligen ingen skillnad på de testerna och tester mot andra API:er som du har i din kod, bara att de ligger i samma kodbas. Det är ju precis lika viktiga API:er.

På samma sätt, när du gör ändringar i en endpoint som du använder på något specifikt ställe så kan du missa att den även används på två andra ställen. Testerna kommer inte att uppmärksamma dig på detta för där är anropen fakeade till att härma endpointens tidigare beteende. Så behovet är ungefär samma och uppstår när ett team hanterar all kod. När ett API ligger utanför ens kontroll så har man tillgång till dokumentation och endpoints ändras inte utan vidare, så då kan man använda sig av fixtures för hur ett response ser ut och kan dessutom ofta använda versionerade API:er. Det är knappast troligt att någon på twitter får för sig att utan vidare ändra så att till exempel users får heta followers istället, utan att det blir en ny version, då skulle ju allt gå sönder.

Ur samarbetssynpunkt, när det gjorts en lokal överrenskommelse som inte alla fått ta del av så kan det vara knepigt för andra att veta vad som har beslutats om det inte finns tester som specificerar detta. Alternativet att ha ett kravdokument är inte så lockande.

Det är väl mycket det som många ser som vitsen med tester nu, att de blir som ett kravdokument fast ett bra sådant, som man inte behöver läsa utan man kör det istället.

Jo men verkligen, så är det ju.

Vad skulle du ge för tips till någon som är ovan vid att testa? Om vi säger att du skulle ha träffat mig för ett halvår sen, innan jag skrivit mitt första JavaScript-test?

Testa inte gammal kod, haha. Idag skulle jag säga ”kör Yeoman, sätt upp ett projekt, prova lite”. Det finns tyvärr fortfarande rätt så lite skrivet om det tror jag, inga riktigt

bra resurser om JavaScript-testning, men jag skulle nog rekommendera att läsa clean code-böckerna och förstå vad det är du ska ha testning till. Utan den grundläggande förståelsen så är det svårt att se värdet i det och man hamnar lätt i tankesätt som ”men vadå, trots att jag har skrivit det här testet så kan det ju ändå bli buggar”. Utan att förstå konceptet testning och varför man gör det så har man svårt att väga olika verktyg mot varandra och motivera de val man gör.

Sen pratade vi också om att involvera alla i testningen. Det kan kanske vara en hjälp för en enskild programmerare att istället för att behöva kämpa för att få testa sin kod och hetsas till ett högre tempo ha testning som en del av det man förväntas göra.

Ja, absolut. Där handlar det igen om att vara överens om att det är viktigt och att det är ett sätt man vill jobba på, annars tror jag att det är svårt.

Vilka ser du som viktiga att involvera då? Teamet gissar jag är fundamentalt, men vill man även få ut det till produktägare, till chef, till kund, till användare? Var vill man lägga detaljnivån?

Det där beror så mycket på vad det är för kund och så vidare. Men framförallt så kommer det i arbetet att speca upp stories och liknande, för även om man kan komma bort från kravspecen så tror jag inte att man kommer bort från kravarbetet. Först och främst, hur funkar prisalgoritmen, hur är prissättningen, vart kommer grejerna ifrån? Där har man inget annat val än att jobba med produktägare och kund i att ta fram vad som ska hända, så det är snarare att vara involverade i story-arbetet tillsammans. Där tar man fram input till testerna så att man är överens om den. Sen tror jag inte så mycket på att man ska sitta och gå igenom testerna tillsammans med kunden för det brukar inte ge så mycket och blir ett onödigt steg.

Cucumber och liknande?

Ja, precis. Det finns ju bra exempel på det, men man ska veta vad man håller på med om man gör det.

Vi hade en tanke i vårt tidigare projekt om att låta användare skriva selenium-tester genom att ha ett plugin i firefox som genererar tester, så varje gång någon hittar något som inte funkar så skapas ett test för det.

Det är ju ascoolt.

Det blev aldrig så.

Det vore tufft om man kunde göra det, att en användare får repetera vad man gjorde. Det kan potentiellt bli hur häftigt som helst.

Hur ser du på selenium-tester i allmänhet, hur mycket tycker du att man ska använda det och till vad?

Det där är svårt, jag har kört det ganska mycket förut men det är mest för att jag inte har kunnat få ihop en runtime annars. I en gammal webb där man var tvungen att requesta sida för sida så är det ditt enda sätt att testa flöden. Nu när jag jobbar med en SPA (single page application) så kan man testa flöden ändå, då är man inte lika

beroende av selenium. Jag tror inte heller på att testa flöden utöver att speca ner den mest grundläggande funktionaliteten.

Lunch

Jag tycker att det var kul att du ser JavaScript som vilket annat språk som helst, för det är det verkligen inte alla som gör.

Nej, det är lite stående på mitt jobb också, när man hittar diverse bra grejer och reaktionen blir "Ah, det är nästan som att programmera på riktigt" och man får flika in att de borde vara tysta.

skratt

Det har att göra med vad man gör i språket, om man använder ett språk till riktiga saker så blir det ju till ett riktigt språk.

Sen så funderar jag på alla de här ramverken, vad det är som du brukar tänka när du väljer vilka du ska använda, tycker du att det är svårt att veta vad de kan och så?

Jag går ofta fram och tillbaka där, det viktigaste är att man inte sätter sig i situationer där man måste jobba mot ramverket. Det leder ofta till dåligt resultat för att "det blir en massa kod som ingen förstår som motverkar en massa annan kod som ingen förstår" (obegriplig kod). Det där är ett problem som man nästan alltid hamnar i med de här magic bullet-ramverken, som Ruby on Rails till exempel. Man inbillar sig att man har ett ramverks som gör allt åt en, så att man kan göra en blogg på en kvart eller liknande. Så kan det också vara i små projekt, och då är det bra. När man däremot använder det i ett större projekt så uppstår lätt en känsla av att något inte är riktigt som det ska. Man lägger oproportionerligt mycket tid på att använda ramverket till saker det inte är tänkt för från början. Frågor som "hur gör man det här i ramverket?" uppstår och när man kommit till det stadiet så är man verkligen låst i ramverket. Det blir att man försöker anpassa sig till hur ramverket är tänkt att användas snarare än att man tar fram sin lösning utefter den vision man har.

I fallet med rails så tror jag att en stor del av problemet är att man har "felriktade dependency-pilar", man har bakat ihop persistence med domänobjekten till en kombination som är svår att reda ut.

Det viktigaste när man väljer ramverk är att hitta ett ramverk där man känner att man har frihet att strukturera koden som man behöver. Det är samtidigt något som man behöver känna sig fram med.

Jag tänker på det du sa om integrationstestning, att man ofta har separerat det, det skulle kanske kunna vara en ramverksfråga också? Om man har ett ramverk där man får uppfattningen av att man behöver strukturera koden på ett visst sätt så kan det hindra en från att skriva integrationstester för saker som tvingats isär av ramverket?

Det är ett stort problem och något man inte vill hamna i, att man inte kan testa grejer för att ramverket säger så. Det är verkligen grunden i att välja ramverk att man måste kunna undvika det.

Du sa att du gärna har ganska små ramverk.

Ja, och där kan man nämna backbone som ett exempel. Jag gillar att det håller sig ur vägen (unobtrusive). Den ger förslag till hur man ska göra vissa grejer som har med infrastruktur att göra och sen så får jag själv bestämma till exempel hur modeller ska läsas upp. Tyvärr är det inte alltid man kan bestämma vilket ramverk man ska ha och ibland så måste man välja stora ramverk för att man är beroende av ett stort CMS för något projekt eller liknande och då är det inte så mycket att göra egentligen.

Det är väl ofta så inom valtech att kunden säger "jag vill ha EPiServer" eller liknande.

Ja och då får man gilla läget mer, men det är klart, får man välja själv så... Ska det vara någonting litet tycker jag.

Jag har inte hunnit skaffa mig stenkoll på alla de här. Grunt, har du använt det?

Grunt är ett byggverktyg skulle man kunna säga. Vi använder det för att köra testerna, då tar det hand om att starta en lokal server och att packa ihop en distribution. Det är något av en motsvarighet till Rubys Rake eller Javas Maven. Det är viktigt för att det ska bli av att man kör tester och liknande, det måste vara lätt och tydligt hur man ändrar och därför behöver man bra byggverktyg.

Sen tänkte jag på stubbning med Sinon, vanillaJS och egentligen Jasmine också.

VanillaJS är ju bara ett skämt, man driver med...

När du själv skriver funktioner och ersätter...

Ja, men återigen så är det samma princip som i att välja små ramverk, när det gäller mockning och stubbning så tycker jag att det är bra att använda JavaScript så långt det går. Vilket jag egentligen tycker gäller för alla språk, även i Ruby använder jag i första hand det som finns inbyggt i språket.

Jag har läst en bok som är skriven av Christian Johansen som gjort SinonJS, Test-Driven JavaScript Development, där hela boken går ut på att man använder sig av manuell stubbning. Du sparar en kopia av en funktion, skriver över den med en egen och återställer efter testet. Det är först i sista kapitlet som han skriver "och förresten, det finns det här ramverket som jag råkar ha skrivit".

Det ligger ju jättemycket i det, för det är samma sak där som för andra ramverk. Annars sitter du där och slåss mot ett stubbningsramverk, helt ovärt, det är tid du aldrig kommer att få tillbaka. Dessutom blir det bräckligt, ju fler dependencies du har desto bräckligare blir det.

Jag insåg ganska sent att Jasmine har spies, och du kan välja om de ska anropa funktioner du stubbar eller inte, så varför använde jag sinonJS då? Sen insåg jag att Jasmine verkar återställa alla metoder du stubbar automatiskt efteråt, men det betyder ju också att jag inte kan stubba en sak, återställa och stubba på ett annat sätt i ett test. Låt oss säga att jag har en testsvit med en describe som jag har 20 test i och alla förutom ett stubbar en viss metod på ett visst sätt, då blir det knepigt. Det kanske går att komma runt, jag har inte försökt. Där funderar jag på vad man ska ge för rekommendation, för det är ändå det jag vill göra någonstans, "om du inte kommer göra det här och det här

fancy grejerna så behöver du inte sinonJS, du kan nöja dig med Jasmine och att stubba manuellt".

Jag tror att det är bra med exempel, där man visar hur man kan lösa olika fall.

Tidigt i mitt arbete så insåg jag att de exempel som finns är ofta väldigt grundläggande, "så här testar du ett hello world". Men det är ju inte det som folk är oroliga över.

Det där är också problemet, för ofta vill man ju demonstrera en princip som är lika sann för hello world som jättestora JavaScript-applikationer. Folk gör det ofta väldigt lätt för sig så det vore nog bra att försöka hitta lite svårare fall, men inte krångla till det onödigt. Ofta gäller samma principer, tricket är ju att lyckas separera det så bra att ditt test blir som ett hello world.

Om jag skulle skriva en bok så skulle jag inte vilja ha med mina tester. De skulle ju innebära en massa stubs som lyssnar på metodanrop som går härs och tvärs. För det är inte något jag är speciellt nöjd med utan det är så men jag önskar att det vore bättre skrivet. Så det finns inte någon situation där jag kan rekommendera till någon annan att "skriv det här testet" utan snarare "var smartare än vad jag är, skriv bättre kod och lös problemet på ett bättre sätt".

Till slut så kommer man ner till problemet att man måste hitta bra abstraktioner. När det är precis rätt så blir det elegant. Därför tror jag att många exempel är så korta, för man är inte nöjd med de komplicerade testerna, det är inget man vill skryta med.

Sen är det ju så att när man väl har insett varför man vill testa och bestämt sig för att göra det så kommer man väl komma fram, men det vore skönt om jag kunde göra den resan lite enklare för folk.

Nej men ta fram bra exempel för det där om sånt som du hittar på vägen, det tror jag är bra.

Just ja, du skrev spikes i vår mailkonversation, om hur du gör när du inte vet hur du ska testa någonting.

Det är ett TDD-begrepp. Säg till exempel att man ska bygga något och det finns många osäkerhetsfaktorer kring vad slutresultatet ska bli och vad som kommer att fungera. Då är det användbart att ha versionshantering med Git, det är bara att skapa en ny branch och köra loss, skriv inte test utan gör bara. Kasta in grejer, prova, gör, kör hårt. Sen när du har gaffat ihop din prototyp och den börjar fungera något sånär, då går du ur den branchen, gör en ny branch från samma ursprungscommit och börjar om fast med tester. Det är att göra en spike.

Man vill inte ha för långa spikes, för då blir det bökigt. Man ska inte sitta i två veckor och sen kasta och göra om, utan det handlar bara om att hitta en riktning och en känsla för hur man ska lösa problemet. Ofta är det så att man tror att det borde gå om man tänker på ett visst sätt och då kan man göra en snabb validering av det. Ofta kommer man till ett läge då man känner att det kommer att gå att göra på ett visst sätt, man är inte klar men börjar se hur saker och ting kommer att hänga ihop. Då är det läge att avsluta sin spike och börja om med tester.

Hur undviker man känslan då av att man gör dubbelarbete?

En motfråga är, vilken kod sätter du dig och skriver en gång? Du itererar, det är bara att du tar en iteration tidigare, en spike kan vara den första iterationen av din kod. Det är jämt att ”typing is not the bottleneck”, det är inte att skriva koden som tar tid, utan det som tar tid är att komma på hur man ska lösa problemet. Har man väl den lösningen, om man fattar vad som är rätt svar, då är det ganska lätt att skriva tester för det och sen skriva koden. Det där är intressant, för där kan man testa sig själv om det blir samma lösning om man testdriver något som när man bara häver ur sig något.

Det är väl en väldigt bra grej. Jag tror att det är något som många upplever med TDD, att de inte har det här verktyget och då står de där och försöker komma på tester för något som de inte har en aning om hur de tänker implementera.

Där har vi det igen, principer är inte till för att begränsa dig. Om man låter sig begränsas så hamnar man i läget där man känner sig oproduktiv, man måste ju tro på det man gör. Eller i varje fall vilja testa det, sen när man kan reglerna så får man bryta mot dem.

Du hade skrivit i ditt mail att du bara testar det publika API:et.

Ja, och då menade jag publikt API i termer av att jag aldrig skulle skriva ett test för en privat metod. Nu har man ju inte riktigt det i JavaScript, men...

Jag var nyligen hos Sony-mobile teamet och pratade med Kristoffer och han ville verkligen testa en privat metod. Det vi kom fram till var att vi kunde exponera den i testmiljön och gömma den i produktionsmiljön.

Jo, men... Jag tycker ändå inte att det är ett bra test. För det är en implementationsdetalj, det är som att testa variabelnamn. Om det blir för stort så kanske man exponerar fel saker. Man kanske vill ha ett annat API, dela upp i två objekt till exempel, som har API:n sinsemellan. Jag tror att... det finns säkert undantag. Vet man vad man gör, by all means, testa privata metoder, man kommer inte att hamna i TDD-domstolen för det. Men generellt så är det en varningsflagga när man vill skriva ett test för en privat metod, då är det läge att tänka efter.

Min reaktion var att han kanske kan skapa en klass för det som han vill testa. Då var det tydligen tio rader kod och han var rätt nöjd med att det såg ut så.

Jag hade nog inte gjort så, men det är ju en pågående diskussion och folk säger att de har ett case för att göra det och visst jag kanske någon gång hamnar i ett case där jag känner att nu måste jag testa den här privata metoden, men jag har inte varit där än.

Jag hade planer på att skriva tester för det vi gjorde i våras med konsultprofil-sidan, och där hade vi mycket jQuery och använde module pattern. På ett felaktigt sätt förmodligen. När jag sedan tittade på det här och frågade mig hur jag ska kunna testa det så var det verkligen som du sa innan om legacy JavaScript att det inte gick att testa alls. De enda testerna jag kunde skriva var ”existerar den här funktionen?” och det är ju i princip att testa variabelnamn.

Ja och det är ju ganska pointless (meningslöst) egentligen, det kommer du inte ifrån. Den viktiga grejen med att inte testa privata metoder är att tester är till för att kolla att

saker blir rätt och i övrigt är du fri att göra vad du vill. Jag tänker att problemet med hans privata metod det är när du kommer in i kodbasen och har ett mycket elegantare sätt att lösa hans problem på, då finns det ett test som säger att den här metoden måste fungera så här och då kan du ju inte göra din eleganta lösning för den måste ju tydligen göra så. Då måste du gå till någon som vet och fråga vad tanken bakom testet är för att i bästa fall kunna våga ta bort det. Det är faran med att testa för djupt.

Det kan jag nog känna med de tester jag skriver nu också, inför det här workshopen, att det kan eventuellt uppstå en sådan situation att jag har skrivit ett test utifrån en specifik implementation och missat poängen med den. Att hitta rätt nivå att lägga sig på är inte lätt. Framförallt inte när man skriver tester i efterhand, vilket han ju också gjorde.

Nej, det är ju inte det. Då är det rätt hopplöst. Man får även skilja på konceptet privat metod som i att det står private i koden och att det är en privat metod. I JavaScript så kan man inte göra en metod privat, men den kan ändå vara tänkt att användas som om det vore det och inte är till för att exponera. Det är bra om detta framgår i testerna.

*Du kan testa att en metod är privat? *skratt**

Även om du skulle fälla in (inline) den metoden direkt i din kod så skulle testerna fortfarande gå igenom. Koden gör fortfarande rätt även fast den är skriven på ett annat sätt.

Kodduplicering, när man ser till att hålla testen korta så är det kanske inte ett lika stort problem om det bara är tre rader i varje test.

Jag tycker att där är läsbarhet viktigast. Man ska inte behöva läsa 700 rader testkod för att komma till kärnan med det.

Det är många som säger att DRY-principen inte behöver tillämpas på tester och det är samtidigt många som säger att tester behöver vara maintainable.

Det är absolut en avvägning, men jag tror helt klart att caset är annorlunda jämfört med när man skriver produktionskod. Det är också det här med premature optimization, att folk använder DRY-principen fel, både i test och i produktion. Om man tolkar det som att samma kodrad inte får förekomma två gånger i ett projekt så har man missat poängen, det handlar ju snarare om att man inte ska duplicera funktionalitet. Om två olika saker råkar som en del av deras funktionalitet göra samma sak så betyder inte det att det ska dras ut till en metod, alltid. Det är likadant med tester, ofta är det så att saker ser väldigt lika ut, som att de gör samma sak, men ofta är så inte fallet. Då hamnar man i dåliga abstraktioner som i själva verket gör testerna mindre maintainable.

Det tycker jag ofta att man kan märka, man tänker ”ok, om jag ska ta ut det här till en separat metod, vad ska jag döpa den metoden till?” Om namnet börjar innehålla villkor ”om det är så här gör så här” och liknande så kanske det inte är rätt sak att göra.

Där kan jag ofta testa mig själv genom att göra kodduplicerings-spåret och i efterhand fråga mig hur lika de olika delarna blev. Ofta visar det sig att det inte var så mycket samma grej som man skulle göra i de olika situationerna. Så det är en princip som har

förstörts lite, det har på vissa håll gått för långt och övergått till att man kodgolfar.

Ja det är ju ofta man kan minska antalet rader kod och det är ju det folk börjar sikta på nu så. Det är väl det du menar med kodgolfning?

Ja, precis, och det är inte alltid så bra.

Hur mycket tid har vi kvar förresten?

Jag måste nog börja röra på mig.

Jag kan maila frågor om det är så.

Ja men gör det, absolut. Hade du något mer?

Nej, inget viktigt.

Då ska jag nog börja dra mig. Men vad kul, det blir spännande att se, du får gärna skicka ditt arbete till mig när du är klar.