How to test your JavaScript

Emil Wall

August 26, 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, 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

Why testing, you may ask. Jack Franklin, a young JavaScript blogger from the UK, gives three reasons: it helps you to plan out your APIs, it allows you to refactor with confidence, and it helps you to discover regression bugs (i.e. when old code breaks because new code has been added). Writing tests to use a library before actually writing the library puts focus on intended usage, leading to a cleaner API. Being able to change and add code without fear of breaking something greatly accelerates productivity, especially for large applications. [3]

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[4] 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[5] 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[6]. 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[7, 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[8] 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[7, 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]Continuous Integration build servers are used for automatic production launch

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[9, 10] and since then several other test framework has appeared such as QUnit [11] and JsUnits sequel Jasmine [12], as well as tools for mocking[2] such as Sinon.JS[13]. 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] mocking and stubbing involves simulation of behavior of real objects in order to isolate the system under test from external dependencies

2.1 Consequences of JavaScript testing

There are consequences (good and bad) of testing JavaScript both from a short and from a longer perspective. The development process is affected; through time spent thinking about and writing tests, shorter feedback loops, executable documentation and new ways of communicating requirements with customers. The 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 Technical background

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

3.1 Browser Automation

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

Selenium WebDriver is a collection of language specific bindings to drive a browser, which includes an implementation of the W3C WebDriver specification. It is based on Selenium RC, which is a deprecated technology for controlling browsers using a remote control server. A common way of using Selenium WebDriver is for user interface and integration testing, by instantiating a browser specific driver, using it to navigate to a page, interacting with it using element selectors, key events and clicks, and then inspecting the result through assertions. These actions can be performed in common unit testing frameworks in Java, C#, Ruby and Python through library support that uses the Selenium-Webdriver API. [14]

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

4 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[15] 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[12] (+ Jasmine-species[16]), qUnit[11], Karma[17], Mocha[18], JsTestDriver[19], Buster.JS[20] and Sinon.JS[13]. 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.

4.1 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.

4.2 Interview Questions

4.2.1 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?

4.2.2 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?

4.2.3 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?

4.2.4 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?

4.2.5 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?

4.2.6 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?

4.2.7 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?

5 Previous work and Delimitations

There exists academic papers on testing web applications and a few focus on JavaScript specifically. Some focus on automatically generating tests[21] 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[22] and Ocariza et al. have investigated frequency of bugs in live web pages and applications[23]. 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[15] by Christian Johansen which deals with JavaScript testing from a TDD perspective. Johansen is the creator of Sinon.JS[13] 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[24] and cucumis[25] 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[10] and JSpec[26] have deliberately been left out of the evaluation. Others have been left out because of fewer users or lack of unique functionality; among these we find TestSwarm, YUI Yeti and RhinoUnit. They are still useful tools that can be considered but including them would have a negative impact on the rest of the evaluation work because of the extra time consuming activities that would be imposed.

6 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

7 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)

8 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

9 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

10 The frameworks

Programming languages in use today typically have frameworks to help with standard structure and other generic problems. JavaScript is certainly no exception, the number

of web application frameworks that focus on JavaScript has increased a lot during the last few years alone.

A full evaluation of the most popular MVC and testing frameworks is not within the scope of this thesis, but others have done it. [3, 27]

11 Draft without title

11.1 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.

11.2 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.

11.3 JsTestDriver and Jasmine integration problems

For instance, when setting up JsTestDriver (JSTD)[19] 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.

11.4 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[28, 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.

11.5 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[29, 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.

11.6 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"[30]

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

11.7 Definitions

A mock has pre-programmed expectations and built-in behaviour verification[15, 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[15, p. 471].

12 Real world experiences

12.1 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.

12.2 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). In the interview with Johannes Edelstam, he agreed that this is one example of something that deters people from testing[31].

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[32].

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.

12.3 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[33].

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[33, 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.

12.4 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.

12.5 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.

12.6 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.

12.7 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.

13 Interview summary

The interview with Johannes Edelstam raised many interesting points, which are summarized in this section. He is an experienced Ruby and JavaScript developer, organizer of the sthlm.js meetup group and a former employee of Valtech, now working at Tink. He has a positive attitude towards JavaScript as a programming language and has extensive experience of test driven development.

It seems that in the last couple of years, the number of people testing their JavaScript has increased significantly[31, question 1]. This has been observed through asking people that do it to raise their hands during tech talks, two years ago only a few raised their hands when asked such a question whereas now almost every single one does it.

According to Edelstam, a likely reason for this change is that the tools have become better and that there are more examples of how to do it. This has caused the opinion that JavaScript is too user interface centered to recede, as people have realized how it can be done. Few people were ever against TDD or testing in general, much thanks to positive experiences from testing in Ruby on Rails projects, which actually act as great examples of that testing wide ranges of interfaces is possible and that many feel that it

is necessary. Perhaps the most common reason for not testing is that the code has not been written in a testable fashion. [31, questions 2-3]

A common experience when writing tests is that you put a lot of effort into the tests and do not write that much production code in comparison, but that can be a good thing! Because then you spend more time thinking about the problem and possible abstractions, which tends to lead to elegant solutions. If you write the tests before the code, you will run into the same problems as you would have done if you wrote the code first, the difference is that you get to think about the design rather than staring at incomplete code when solving the problem. [31, question 8]

The feeling of being limited and not productive when writing tests can stem from a badly chosen level of ambition or that the focus of the tests is wrong, which in turn can be based on poor understanding of what tests are good for. Coding without tests can be much like multitasking, you get an illusion of being more productive than you actually are. One of the positive effects of TDD is that it can prevent you from losing track of direction, and helps you in making clear delimitations, since trying to be smart by allowing a function to do more than one thing means more tests. Testing will not automatically provide you with good ideas regarding where you are heading, but once you have gotten such an idea, testing tends to be easier and help you discover new aspects and scenarios which might would have been left unnoticed without tests. [31, question 8]

When deciding what to test, it pays off to focus on parts that are central to how the application is perceived, for instance pure logic and calculations might be more important than pictures and graphs. An error that propagates through the entire application is more serious than if a single picture is not displayed properly. If a test turns out to be difficult to write or frequently needs to be replaced by another test, it is usually worth considering not testing that part at all. [31, questions 9-10]

In June this year, Kent Beck wrote the following tweet[3]:

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

One of the responses was that many would have given up on writing that test. Beck replied that if you don't know how to write the test, you don't know how to write the code either[34]. What many probably fail to realize about why testing can be time consuming and hard, is that when writing tests you encounter problems that you would have to solve anyway. The difference is that you solve the problems by formulating tests rather than staring at production code for the same amount of time. [31, question 11]

Tools are an important part of facilitating testing, so that people are not so deterred by the initial effort required to get started. Yeoman is one example of a framework that can help you in quickly getting started with a project that is structured so that testing becomes easier. For already existing projects, the increased maturity of tools such as PhantomJS and Mocha is also truly helpful. [31, questions 11-12 and 20]

Error reports are useful feedback that tests provide. The quality of these reports vary

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

depending on which testing frameworks you are using and how you write your tests. When using PhantomJS to run tests, some test failures require you to run the tests in a browser in order to get good error reports. [31, question 12]

An important difference between using a headless browser such as PhantomJS to run your tests compared to JsTestDriver, or other drivers that allow you to test in several browsers, is that a headless browser provides no information about how your code performs on different JavaScript implementations. Reasons why you might still decide to do so include speed, easier integration with build tools such as Jenkins, and that it yields the same results as long as the tests focus on logic rather than compatibility. [31, questions 13-15]

One could argue that JavaScript is better suited for testing than most other programming languages because of the built in features than often make stubbing frameworks redundant. The object literals can be used to define fake objects and it is easy to manually replace a function with another and then restore it. An advantage with using manually written fakes is that it tends to make the code easier to understand since knowledge about specific frameworks or DSLs (Domain Specific Language) is not required. [31, questions 20-21]

The problems that some people experience with testing frontend interfaces sometimes have to do with poor modularity. For instance, the presentation layer should not be responsible for calling large APIs. Tools such as RequireJS can be used to handle dependencies but if each part of the application has too many or large dependencies, mocking becomes a daunting task. Typically, these kinds of problems can be solved by introducing new layers of logic and services that separate responsibilities and allows for much cleaner tests. [31, question 23]

A common case of user interface testing is form validation. Being test driven does not necessarily mean that you should test that the form exists, has a working submit button and other boilerplate things, unless the form is written in a way that is unique or new to you in some way. A typical approach would rather be to write the code for the form and then add a test that searches for a non-existing validation. The difficult part here is to strike a balance and be alert to when the code is getting so complicated that it is time to start testing, and to avoid writing tests when they are in the way and provide little value. [31, questions 24-25]

Being too religious about testing principles leads to conflicts like “writing tests take too much time from the real job”. If the tests do not provide enough value for them to be worth the effort then they should be written differently or not at all. There is no value in writing tests just for the sake of it. Thinking about architecture and the end product is usually a good thing, because you need awareness of the bigger picture in order to prioritize correctly and make sure everything will fit together in the end. There is the same risk with tests as with other pieces of code, sometimes you are especially proud or fond of a certain test and unwilling to throw it away. In order to avoid that situation it is often better to think ahead and try things out than to immediately spend time writing tests. [31, question 27]

Writing implementation specific tests is a common phenomena that can stem from poor

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

A large number of tests is not necessarily a good thing, it is harder to overview and maintain a large collection of tests. Unless the system inevitably is so complicated that extensive testing is justified, it rarely pays off to strive for 100 % coverage, since it takes too much time and the scenarios that might cause problems are at risk of being overlooked anyway. [31, question 28]

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

13.1 Testing private APIs

Testing private APIs differs from testing calls to an external API in that the latter are often well documented, versioned and rarely changes. When using an external API, the main concerns are making sure that the correct version is used and that the API is called in a correct way with respect to that version. A private API on the other hand may change frequently which means that the fake objects representing requests and responses of the API in other tests need to change as well. Private APIs need to be tested because endpoints frequently change and may be used in several places. It is then important to have integration tests that can detect when the fakes need to change. [31, questions 34, 36]

Introducing versioning for private APIs is not always feasible, for at least two reasons. There can be difficulties in keeping track of the different versions and there can be a risk of hampering development since the parts that use the API are dependent on the new version of the API to be released before they can be updated. However, there exists testing techniques that can be employed regardless if versioning is in place or not. One such technique is to write tests for what characterizes an object that the API serves as an interface for, to be able to determine if an object is a valid response or request. These tests can then be used on both real objects and on the fake objects that are used in other tests, which means that when the API changes the tests can be updated accordingly and then the tests involving the fake will fail. Changing the fakes so that the tests passes will hopefully result in that any incompatibility is discovered since the fakes are used in other parts of the testing suite as well. [31, question 34]

Testing a private API is not the same thing as testing a private method. In JavaScript there is formally no private methods, only functions not exposed to the current scope, nevertheless methods that are not intended to be called from the outside can be considered private. It is a common principle that you do not test a private method, if you feel the need to, then that method is probably too complex to be private and the

functionality that it is meant to provide should be moved into a new object or similar. Private methods represent implementation details, tests should be focused on behaviour and functionality, not implementation. [31, questions 62-63]

13.2 Culture, Collaboration and Consensus

Testing is not about eliminating the risk of bugs or ensuring that every line of code is executed exactly the way it was originally meant to be executed. In order to better understand the main benefits of testing, it is recommended to read books such as Clean Code, and to work together with experienced programmers and testers. Without this understanding there is a risk of frustration, inability to decide which tools to use and difficulties in motivating choices related to testing. Collaboration problems may occur when members of a team have different opinions about the purpose of testing. [31, question 38]

To get as much benefit as possible from testing, every developer should be involved in the testing effort and preferably the product owner and other stakeholders too. The work of specifying stories can involve agreeing on testing scenarios, thinking about what the desired behavior is and coming up with examples that can be used as input and expected output in end-to-end tests. [31, questions 39-40]

13.3 Frameworks and tools

There are a number of BDD frameworks available that are meant to bring testing closer to the specification and story work, as proposed in section 13.2. They aim to mimic the way stories are written to create a ubiquitous natural language that can be understood by both programmers and non-programmers. The way this is done is by building sentences with a Given, Then, When (GTW) form. Cucumber is a popular framework that was designed for this in the Ruby programming language and there is a JavaScript version called Cucumber.js. Yadda, Kyuri, Cucumis and Jasmine-given are other examples of JavaScript GTW-frameworks. [35, section 8.4][25]

Apart from testing frameworks, your ability to write tests is also influenced by what application framework you are using. According to Edelstam, if you are able to choose, it is often preferable to have small rather than "magic bullet" frameworks for large projects because they tend to be less obtrusive. Exceptions may include when the framework is very well known to the developers or when there is a perfect fit between what you want to achieve and what the framework is designed for, but one should bear in mind that requirements tend to evolve. [31, questions 48-50]

13.4 Testability and Selenium

As mentioned in section 3.1, Selenium is currently the most popular tool for browser automation. Sometimes it is the only way to test the JavaScript of an application without rewriting the code (as mentioned in section12.6) but then tests tend to be brittle and

provide little value. In general, since Selenium tests take such time to run they should only cover the most basic functionality in a smoke test fashion. Testing all possible interaction sequences is rarely feasible and should primarily be considered if it can be done fast, such as in a single page application (SPA) where sequences can be tested without reloading the page. [31, question 44]

With Selenium IDE (see section 3.1), there is a possibility of recording a certain interaction and generate code for it. This could potentially be used to reproduce bugs by letting the user that encountered the bug record the actions that led to it, and communicate with a developer so that proper assertions are added at the end of the sequence. This has some interesting implications, but experience shows that it is hard to do it in practice since it requires the users to be trained in how to use the plugin and to invest extra time whenever an error occurs. [31, questions 42-43]

13.5 Build tools

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

Naturally, there are build tools designed specifically for JavaScript as well, Grunt being the most popular, which can be installed as a node.js package, has plugins for common tasks such as lint, testing and minification, and can be invoked through CLI. [36][31, question 52] Jake and Mimosa are other well known and maintained alternatives. It is also possible to use Rake, Ant or similar. [37]

13.6 Stubbing and Mocking

In JavaScript, tools for stubbing can be superfluous because you are able to manually replace functions with custom anonymous functions, that can have attributes for call assertion purposes. The stubbed functions can be stored in local variables in the tests and restored during teardown. This is what some refer to as VanillaJS [31, question 53]. It might come across as manual work that you can avoid by using a stubbing tool, but the benefits include fewer dependencies and sometimes more readable code, as mentioned in section 13. [31, questions 54-55]

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

13.7 Patterns, Examples and Documentation

Examples are useful when learning idiomatic ways of solving problems. Code tends to end up being more complicated than examples you read because it is hard to come up with useful abstractions that make sense. Those who write examples to illustrate a concept always have to find a tradeoff between simplicity, generality and usefulness, and tend to go for simplicity [31, questions 56-57]. Combining different concepts may help to achieve code with good separation, that can be tested by simple tests. Today blog posts and books about patterns in JavaScript are in abundance and you can often find examples in the documentation of frameworks. When it comes to examples of testing in general there are several classics to refer to [38][7]. For examples of JavaScript testing specifically the alternatives are scarce but they do exist [15][35][39].

13.8 Spikes in TDD

Although writing tests first is a recommended approach in most situations, there is a technique for when you need to try something out before you write tests for it, without compromising testability. Dan North, the originator of BDD, came up with a name for this technique: spiking [40]. The idea is that you create a new branch in your version control repository, and hack away. Add anything you think might solve your problem, don't care about maintainability, testability or anything of the sort. If you find yourself not knowing how to proceed, discard all changes in the branch and start over. As soon as you get an idea about how a solution could look like you switch back to the previous branch and start coding in a test first fashion. [31, question 59]

There is an ongoing discussion about whether or not you have to start over after a spike. Liz Keogh, a well known consultant and core member of the BDD community, has published posts about the subject in her blog, in which she argues that an experienced developer can benefit from trying things out without tests (spiking) and then stabilizing (refactoring and adding tests) once sufficient feedback has been obtained to reduce the uncertainty that led to the need for spiking [41]. She argues that this allows her to get faster feedback and be more agile without compromising the end result in any noticeable way. In another post, she emphasizes that this approach is only suitable for developers who are really good at TDD, while at the same time claiming that it is more important to be "able to tidy up the code" than "getting it right in the first place" [42]. It may seem like an elitist point of view and a sacrilege towards TDD principles but in the end, whatever makes you most productive and produces the most valuable software has raison d'être.

Counterintuitive it may seem, throwing away a prototype and starting from scratch to test drive the same feature can improve efficiency in the long run. The hard part of coding is not typing, it is learning and problem solving. A spike should be short and incomplete, its main purpose is to help you get your mind set on what tests you can write and what the main points in a solution would be. [31, question 60]

13.9 The DRY principle in testing

The point of the Don't Repeat Yourself (DRY) principle is *not* that the same lines of code cannot occur twice in a project, but that the same *functionality* must not occur twice. Two bits of code may seem to do the same thing while in reality they don't. This applies to eliminating duplication both in production code and tests, do not overdo it since that will harm maintainability rather than improve it. Sometimes it is beneficial to allow for some duplication up front and then refactor once you have a clear picture about how alike the two scenarios actually are, if necessary. [31, questions 69-70]

14 References

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

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

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

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

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

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

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

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

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

[10] Github. *pivotal/jsunit*. URL: `https://github.com/pivotal/jsunit` (visited on 04/03/2013).

[11] The jQuery Foundation. *QUnit: A JavaScript Unit Testing framework*. URL: `http://qunitjs.com/` (visited on 04/03/2013).

[12] Running documentation. *Jasmine is a behavior-driven development framework for testing JavaScript code*. URL: `http://pivotal.github.com/jasmine/` (visited on 04/03/2013).

[13] Christian Johansen. *Sinon.JS: Standalone test spies, stubs and mocks for JavaScript*. URL: `http://sinonjs.org/` (visited on 04/05/2013).

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

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

[16] Rudy Lattae. *Jasmine-species: Extended BDD grammar and reporting for Jasmine*. URL: http://rudylattae.github.com/jasmine-species/ (visited on 04/05/2013).

[17] Friedel Ziegelmayer. *Spectacular Test Runner for JavaScript*. URL: http://karma-runner.github.io/ (visited on 05/29/2013).

[18] TJ Holowaychuk. *mocha - simple, flexible, fun javascript test framework for node.js & the browser*. URL: http://visionmedia.github.com/mocha/ (visited on 04/03/2013).

[19] Cory Smith et al. *JsTestDriver*. URL: https://code.google.com/p/js-test-driver/ (visited on 04/05/2013).

[20] August Lilleaas and Christian Johansen. *BusterJS: A powerful suite of automated test tools for JavaScript*. URL: http://docs.busterjs.org/ (visited on 04/05/2013).

[21] 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.

[22] 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.

[23] 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.

[24] Alexis Sellier, Charlie Robbins, and Maciej Malecki. *Vows: Asynchronous behaviour driven development for Node*. URL: http://vowsjs.org/ (visited on 04/05/2013).

[25] Eugene Ware. *Cucumis: BDD Cucumber Style Asynchronous Testing Framework for node.js*. URL: https://github.com/noblesamurai/cucumis (visited on 04/05/2013).

[26] TJ Holowaychuk. *JSpec on Github no longer supported*. URL: https://github.com/liblime/jspec (visited on 04/05/2013).

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

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

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

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

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

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

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

[34] Kent Beck. *Twitter - It almost never takes me 6 hours to write one test.* URL: https://twitter.com/KentBeck/status/350039069646004224 (visited on 07/26/2013).
[35] Marco Emrich. *Behaviour Driven Development with JavaScript.* Developer.Press, 2013. ISBN: 9781909264113.
[36] Ben Alman. *GRUNT The JavaScript Task Runner.* URL: http://gruntjs.com/ (visited on 08/21/2013).
[37] Philip Rose. *Stack Overflow - Build process tools for JavaScript.* URL: http://stackoverflow.com/questions/7719221/build-process-tools-for-javascript (visited on 08/21/2013).
[38] Kent Beck. *Test-Driven Development By Example.* ADDISON-WESLEY, 2002. ISBN: 9780321146533.
[39] Mark Ethan Trostler. *Testable JavaScript.* O'Reilly Media, 2013. ISBN: 9781449323394.
[40] Dan North. *Twitter - I think I was the first to name and describe the strategy of Spike and Stabilize but there were definitely others already doing it.* URL: https://twitter.com/tastapod/statuses/371352069812527105 (visited on 08/26/2013).
[41] Liz Keogh. *Beyond Test Driven Development.* June 24, 2012. URL: http://lizkeogh.com/2012/06/24/beyond-test-driven-development/ (visited on 08/24/2013).
[42] Liz Keogh. *If you can't write tests first, at least write tests second.* Apr. 24, 2013. URL: http://lizkeogh.com/2013/04/24/if-you-cant-write-tests-first-at-least-write-tests-second/ (visited on 08/24/2013).

15 Appendix

15.1 Transcript of Interview with Johannes Edelstam

1. 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.

2. 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.

3. 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.

4. 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?

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

Precis.

6. 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.

7. 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.

8. 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 tid och sitt 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.

9. 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.

10. 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 tester 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.

11. 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.

12. 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 PhantomJS i byggen på Jenkins, så det är en del av byggflödet.

13. 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.

14. 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.

15. 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.

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

Mm, vad kör du nu?

17. Nu kör jag JsTestDriver och Jasmine.

Det är väldigt likt, Jasmine och Mocha.

18. Så du kör Mocha istället för Jasmine?

Ja.

19. 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.

20. 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.

21. 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.

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

Ja, det har det garanterat.

23. 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.

24. 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.

25. 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.

26. 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".

*27. *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.

28. 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 känna 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 han eller hon 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.

29. 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.

30. 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.

31. 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.

32. 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.

33. 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.

34. 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.

35. Och det är lite mitt emellan Selenium och enhetstester då?

Ja, precis.

36. 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 överenskommelse 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.

37. 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.

38. 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.

39. 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.

40. 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.

41. 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.

42. 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.

43. 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.

44. 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.

*45. *Lunch**

46. 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.

*47. *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.

48. 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.

49. 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.

50. 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.

51. 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.

52. 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.

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

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

54. 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.

55. 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.

56. 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.

57. 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.

58. 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.

59. 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.

60. 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.

61. 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.

62. 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...

63. 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.

64. 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.

65. Jag hade planer på att skriva tester för det vi gjorde i våras med konsultprofilsidan, 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.

66. 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.

*67. 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.

68. 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.

69. 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.

70. 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.

71. 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.

72. Hur mycket tid har vi kvar förresten?

Jag måste nog börja röra på mig.

73. Jag kan maila frågor om det är så.

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

74. 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.

15.2 Transcript of Interview with Patrik Stenmark

1. Du skrev i ditt mail till mig att en nackdel med JavaScript är att det är "mycket ceremoni för ett dynamiskt typat språk". Vad menade du med det?

Jag borde kanske ha skrivit funktionellt språk snarare än dynamiskt typat. Jag menar att det till exempel är ganska klumpigt syntaxmässigt att skicka med en anonym funktion i ett anrop. Du måste skriva function med parenteser, klammer, i vissa fall semikolon efter definitionen, 20 rader nedanför, i andra fall ska det inte vara det, beroende på om det är ett objekt eller bara en funktion du definierar. Man behöver fundera rätt mycket på syntax.

2. Det känner jag igen som har suttit rätt mycket med python, där tycker jag att det är bättre.

Jag är ju Ruby-utvecklare om jag får välja själv. Python är ännu renare, för i Ruby finns det en del saker som man får fundera på, men i JavaScript så är det så i stort sett hela tiden. Om man till exempel ska flytta en funktion från att vara anonym till att bli en egen funktion för att den har blivit för stor, då är det inte bara att flytta den utan man behöver även fundera på om den hamnar i ett objekt så att det måste finnas kommatecken på rätt ställe eller om det blir en vanlig funktionsdefinition som ska avslutas med semikolon. Om den flyttas sist i ett objekt så ska det inte vara kommatecken, för då blir det syntaxfel i vissa versioner av Internet Explorer. Det är mycket sånt som jag tycker blir väldigt jobbigt. Visserligen finns JSLint och JSHint och liknande som kan hjälpa till, men det är fortfarande lite för mycket saker som man inte borde behöva tänka på.

3. Du skrev att du har använt JavaScript till både små saker och till lite större. Tycker du att det lämpar sig bra till stora applikationer?

Nej, *skratt*, egentligen inte. Jag tycker inte alls om JavaScript som språk, jag tycker att det är alldeles för mycket specialfall och konstiga quirks. Objectmodellen är väldigt konstig och känns schizofren i att man har ett prototyp-baserat system men också en new-operator som emulerar new-operatorn i Java, fast inte riktigt. Det är mycket sådana designbeslut som jag tycker är ganska konstiga. Däremot så är det ju ett språk som finns tillgängligt överallt, så det gör att det ändå funkar att hålla på med. Om det hade varit så att alla webbläsare hade stöd att exekvera Ruby utan att först ladda hem emscripten så skulle jag definitivt föredra det. Sen så finns det ju ramverk och grejer som hjälper till och gör att det fungerar bättre, men i grunden så nej, jag tycker inte att det är ett lämpligt språk att skriva stora applikationer i.

4. Så det är mest att det är enda alternativet?

Ja.

5. Nu är jag inte helt insatt, men Single Page Applications (SPA) känns som en typisk grej där JavaScript är enda valet.

Ja, där har du ju inte något alternativ egentligen. I varje fall så måste slutprodukten vara JavaScript. Den största grejen jag har gjort, som är en SPA, skrev jag i Coffeescript, men det är ju JavaScript.

6. Hur ser du på testning när det kommer till stora program, blir det mer relevant då?

Ja, definitivt.

7. Tror du att man klarar sig utan det överhuvudtaget?

Ja klarar sig gör man ju, men utifrån min erfarenhet med en tidigare applikation som jag har jobbat med, som tog in statistik på ena sidan och byggde upp grafer som gick att filtrera på olika parametrar, så blev jag flera gånger hjälpt av testerna, särskilt när ny funktionalitet skulle läggas till. Då fanns det en bekräftelse på att jag inte hade haft sönder någonting tidigare och testerna manade mig till att skriva lite mer välstrukturerat.

Från vad jag sett i andra projekt så när man inte har tester så blir det bara en stor soppa till slut där varenda ändring man gör tar jättelång tid och har sönder fyra andra funktioner. För att man inte var medveten om att en viss jQuery-handler måste köras före en annan och så har man bytt plats på lite kod för att öka läsbarheten och helt plötsligt funkar ingenting längre.

8. Jag vet inte hur mycket du har hållit på med node?

Ingenting.

9. Det jag funderat lite på är hur attityden gentemot testning har förändrats genom tiden, det känns som att node har varit en milstolpe som gjort att man börjat utveckla i JavaScript mer seriöst.

Det har ju påverkat testning inom webb-JavaScript också, för trots att jag aldrig har kodat för node så har jag ändå haft nytta av att ha en JavaScript-runtime tillgänglig som gjort att man kunnat köra testverktyg. Tidigare behövde man skriva en html-sida och ladda i webbläsaren för att köra sina JavaScript-tester. Det var ganska meckigt, att sätta upp en ny testsvit var att sätta upp en ny html-sida. Det node har gjort för mig är att jag kan köra mina tester i terminal med PhantomJS eller Selenium beroende på vilken nivå jag vill lägga det på. Så på sätt och vis har jag använt node, men jag har inte kodat node.

10. Varför tror du att JavaScript-kodare ofta föredrar quick-and-dirty?

Jag skyller på jQuery. *Skratt* Det har mycket att göra med hur JavaScript började, för 10 år sen så hade du inte kunnat skriva en SPA för du hade knappt AJAX men jag kan tänka mig att många har lärt sig JavaScript för att kunna animera en knapp, vilket är enkelt att göra genom att skriva några få rader jQuery i en fil och inkludera den. Sen utvecklas det till att man vill ha lite mer menyer som expanderar och andra effekter vilket i sin tur övergår till att man implementerar faktiska funktioner i JavaScript. Då har man läst tutorials på Internet som berättar att man skapar en viss handler i vilken man läser in något från DOMen och manipulerar det och använder en selektor för att

hämta ut ett visst element och i det läget så är testbarheten i stort sett förlorad även om man har en utvecklarbakgrund sedan tidigare.

Sen så tror jag att många JavaScript-utvecklare kommer från designer-sidan, som inte har kunskaperna som krävs för att riktigt veta vad det är de gör. De är designers som har lärt sig jQuery och kan det, utan att egentligen vara så insatta i programmering i stort. Sen så sitter de och lappar ihop saker och kan absolut vara asgrymma på jQuery och åstadkomma jättehäftiga effekter och bra saker på så sätt, men de har inte strukturen och koddesign-tänket som man får som utvecklare när man bygger stora saker. Det har jag märkt när jag jobbat med frontend-utvecklare att de tycker att det är helt rimligt att ha en fil med 3000 rader kod med en funktion som är 250 rader lång som bara innehåller DOM-uthämtning och AJAX-request och allting i en stor soppa. De tycker inte att det är något konstigt överhuvudtaget, för det är så de har lärt sig. Tidigare har det varit väldigt lite information på Internet om hur man gör. Tutorials, dokumentation och liknande visar inte det utan det är de snabba lösningarna som visas oftast.

11. Så, det har varit lite information om hur man gör det bra.

Ja.

12. I ditt mail nämnde du några ramverk som du använt: Backbone, Ember, Angular. Framförallt Backbone, vad jag fattade det som. Och så lite testning men Buster, Jasmine och Sinon. När tror du att de är lämpliga att använda sig av?

Idag skulle jag nog inte använda Backbone överhuvudtaget, det var jättebra när det kom men det känns som att det hade en hel del brister som har stannat kvar och sen så har det kommit en massa andra som har sett Backbone och tyckt att det varit bra men att vissa saker saknats och därför gjort något nytt som är snäppet bättre.

Det finns olika komplexitetsnivåer på det, Backbone är ju i den lägre skalan, det är väldigt simpelt egentligen. Den typen av ramverk känns rimliga när det är små applikationer. Kanske som en del av en större site som hålls isolerad från övriga delar av siten. Däremot så om man ska bygga en SPA-style så skulle jag nog inte gå mot Backbone utan istället utgå från hur komplext det är och välja Angular eller CanJS om det var av medelstorlek eller Ember om det gäller en applikation i stil med Gmail och avancerade forum-mjukvaror, som ska konkurrera med desktop-applikationer. Att dra fram Ember för att göra en liten Todo-lista är bara overkill och onödigt.

13. Du tror inte att det är risk att det är i vägen när det är så stort?

Jo, det är det, men jag tror också att när det är en så komplex applikation så är det värt att ha ett ramverk som hjälper till rätt mycket, i synnerhet när man måste använda JavaScript. Men absolut, jag är ju Rails-utvecklare och i den världen så är jag väldigt mycket inne på att försöka frikoppla så mycket som möjligt från Rails när jag bygger saker. Jag har inte sett det göras ordentligt eller på ett bra sätt i JavaScript-världen ännu. Det kanske går, jag kanske ändrar mig om ett par år eller någonting, men just nu så känns det som att det enklaste och mest effektiva är att ha ett ramverk som hjälper till rätt mycket. Sen kan man säkert hamna i att man får slåss lite mot det eller anpassa vad man vill göra till vad ramverket klarar av.

14. Vad är de största vinsterna med de här då, vad är det man får hjälp med?

När det gäller Ember så får du ju strukturen på applikationen, hur den ska byggas upp. Allting bygger på någon slags state machine som styrs av vilken URL du är på. Sen att du får en riktig MVC, eller i varje fall riktigare än de flesta andra ramverk, och att du har databindningarna både på ren data och på det som kallas computer properties, så du kan ha en funktion som genererar ett värde från andra värden som du också kan binda till DOM-element, vilket gör att mycket av situationen där en händelse ska trigga en mängd andra händelser blir enklare.

När det gäller de enklare verktygen så får man främst organisation och hjälp med boilerplate-kodning. Man vet vad vyer och controllers förväntas innehålla, vilket ger struktur.

15. Känner du att det hjälper för att skriva tester också?

Tankepaus Ja, det gör det ju. Det får lite samma funktion, de hjälper varandra kan man säga. Du kan inte skriva tester för dåligt strukturerad kod utan att det blir komplicerat, så det hjälper ju dig att du redan har någon form av struktur, men jag tror också att testerna hjälper dig att få ännu bättre struktur även om du använder Backbone eller Ember. För du kan ju fortfarande skriva dålig kod i de ramverken, men då hjälper testerna till att förhindra det.

Om du inte hade något ramverk från början så skulle du nog hamna i en bra struktur ändå om du bara körde testdrivet. Jag påstår inte att det blir bra bara man har tester, men testerna kan hjälpa dig även om du inte har ett ramverk. Båda hjälper från varsitt håll.

16. Hur mycket har du testat just gränssnitt då?

En del, men du har det inte varit så mycket i JavaScript utan mer Selenium från Ruby. (Jag har skrivit tester med Jasmine som körs med Selenium aldrig så att alla tester körs via JavaScript.) Det är väldigt sällan värt det, för allting blir så känsligt för förändringar i utseende, det är långsamt och så fort du har någon form av asynkronitet så får du tester som failar ibland. Det kan bli så att de failar vissa tider på dygnet och kontentan blir att de gör mer skada än nytta för att de inte går att lita på och tar så lång tid att köra att man till slut inte orkar vänta på testerna.

Däremot så tror jag att det är bra att ha väldigt övergripande för de viktigaste flödena. I mitt nuvarande projekt så har vi Selenium-tester för de flöden som har med pengar att göra trots att de involverar en del JavaScript, men vi testar inte att dropdown-menyer, växling mellan olika vyer och liknande fungerar utan bara det absolut viktigaste som ett smoke test.

17. Vad menar du med smoke test?

Att prova att gå till sajten, se om den exploderar ungefär. Väldigt övergripande för att se om standardfallet fungerar som det ska. Testa siten och se om det ryker från den eller inte. *Skratt*

18. Så det är just Selenium du har använt för gränssnitt då?

Ja, eller nej, inte bara. Jag har gjort en del genom att ladda in någon form av template i någon testhtml-sida och sen köra jQuery mot den div:en. Då hamnar man i det som jag

nämnde i mitt mail, att man behöver hålla det i synk med vad som finns i produktion. Det kan bli så att man ändrar i designen på siten men att testerna fortfarande går igenom eftersom de är baserade på templates som du har skapat trots att siten har blivit trasig. Då har testerna körts av Selenium på en CI-maskin men det har inte varit Selenium som har klickat runt saker utan det används bara för att ladda sidan och sen har jQuery gjort resten.

19. Jag hade några följdfrågor här men de är inte så relevanta nu. Vi pratade i vår mailkonversation om externa APIer, du tyckte att det var svårt att testa dem. Vad är det som gör det svårt?

Det är ju lite samma sak som med gränssnitt, att du har ett beroende på något som du själv inte har full kontroll över. Facebook är till exempel kända för att ändra i sina APIer lite hur som helst och när som helst, när de känner för det. Det finns inget vettigt sätt att testa dem mot live-miljön, för du vill inte att du går in och like:ar något tusen gånger per dag för att du kör testerna hela tiden. Då kan man förstås göra någon slags mock och bara testa att rätt metoder anropas och så där, men då har man problemet att om Facebook ändrar i sitt API så kommer ens tester fortfarande att gå igenom men siten kommer inte att fungera.

Samma sak gäller för vilka APIer du än använder, det kan även vara så med APIer du har själv, för du vill ju inte köra dina enhetstester mot en riktig server för då är de inte riktigt enhetstester längre utan har blivit någonting annat. Det är enklare när det är APIer man själv har kontroll över för då kan man sätta upp någon slags integrationstestmiljö som man använder bara inför release eller liknande för att kontrollera att de mockade testerna fortfarande är uppdaterade. Så gjorde jag när jag arbetade med det där statistikhanteringsverktyget som jag nämnde tidigare och det fungerade riktigt bra, kanske för att det var ett API som jag själv byggde och hade full kontroll över. Så fort det blir något som man själv inte har kontroll över så blir det mer komplicerat.

20. Det är väl så att privata APIer ändras ännu oftare och att det ytterligare späder på problemen med att den mockade datan kommer i osynk med den riktiga datan, så som du skrev i ditt mail till mig. När jag intervjuade Johannes Edelstam så föreslog han att man kan ha tester som kontrollerar att ett objekt är ett korrekt request eller ett korrekt response, och köra dem både på de riktiga objekten och på din fake som du har när du mockar ett API. Då märker du att om testerna slutar gå igenom på de riktiga objekten då behöver du ändra på dem och då kommer de att börja faila på faken, och då ändrar du på faken, och då kommer de tester som faken används i att faila. Har du använt dig av det någonting?

Inte i JavaScript-världen, men i Ruby-världen så gör jag det ofta nu för tiden. Man behöver oftast komma ut på Internet för att kunna göra det, vilket ibland är ett problem och ibland inte. Jag hade någon betalningslösning en gång där det kostade pengar att göra testbetalningar, och då vill man förstås inte göra en testbetalning varje gång man kör sina tester. Men ibland så är det inga problem, om det till exempel gäller en sökning på Twitter så är det säkert inga problem för det får du göra ganska många. Jag tycker absolut att det är en bra idé och jag använder den själv när jag jobbar med backend-kod, jag har dock inte provat det i JavaScript.

21. Du skrev i ditt mail att DOM-beroende tester blir långsamma. Var det just Selenium du tänkte på då?

Oavsett om du kör Selenium eller PhantomJS eller någonting så är de ju långsammare än rena JavaScript-tester.

22. Är det för att det är stora objekt som de skapar eller...?

Ja, och DOM:en är ganska tung att jobba med, antar jag. Jag vet faktiskt inte varför det går långsamt även när du kör Phantom, men det blir väl mer att göra helt enkelt. Det kanske inte är så jobbigt att köra ett test, men om du har 100 tester som är oberoende och varje test tar 100 ms så är du ändå uppe i 10 s körtid. I mitt nuvarande projekt så har vi väl 6-700 enhetstester och om alla de skulle ta 100 ms så skulle vi ju ha blivit gråhåriga vid det här laget. På enhetstestnivå så blir det rätt många tester, i varje fall för mig.

23. Vad strävar du efter då när du har så många tester, försöker du täcka in många olika fall vill du bara ha ett test som talar om vad en sak ska göra?

Nej, på enhetstestnivå så vill jag ju täcka in allt i den mån det går. Det är ett slags mål som jag vet inte riktigt går att uppnå, men ändå det jag satsar på.

24. Just för att kunna känna dig trygg sen när du gör ändringar?

Ja, precis. När jag ändrar och framförallt när andra ändrar. Om jag sitter själv på ett projekt så har jag oftast koll på vad som händer, men när man är flera personer, kanske geografiskt separerade som vi är i vårt nuvarande projekt med fem personer i Stockholm och två på Malta, så är det en stor fördel om man kan förvissa sig om att ingen annan har haft sönder någonting genom att se att testerna fortfarande går igenom.

25. Jag såg att du skrev på intranätet att du kom tillbaka från semestern och fick sätta dig och fixa tester, var det någon annan som hade...?

Ja, eller, det var någon som hade stängt av testerna för att det skulle fortsätta vara grönt i Jenkins. Jag köpte inte riktigt det tankesättet. Så fort man har tester som ibland inte fungerar, så kan man lika gärna kasta dem, för om de spontanfailar så att man inte kan lita på dem så blir det bara onödigt jobb.

26. Var det det som var fallet nu?

Ja, en av testerna hade lite för låg timeout tror jag att det var, så ibland hann den inte svara. Istället för att lösa det problemet så hade de bara stängt av alla testerna.

27. Det är väl viktigt att folk har samma ambitioner, kanske.

Mm.

28. Du listade lite olika saker som du brukar ta hänsyn till när du avgör vad du ska testa. Du sa att du gick mycket på känsla också, men. Hur relevant något var, det tolkade jag som att det var just funktionaliteten, hur relevant den var. Att om det är väldigt relevant så är det större chans att du testar.

Ja, som jag sa tidigare. Betalningsflöden kör jag i Selenium för att vara hundra på att det fungerar, men en liten dropdown-meny som visar kontaktinformation kanske jag

bara fulhackar ihop med jQuery utan att skriva något test för det. Sen finns det en skala därimellan.

29. Sannolikhet till förändring, om något förändras ofta, är det en anledning till att testa det?

Ja, precis. Om vi vet att vi kommer att lägga till funktioner och ta bort och ändra så vill man se till så att det man har byggt fungerar, men om man vet att det är något man gör en gång och som sen kommer att ligga kvar där, då tjänar man nog på att manuellt testa det så att det funkar och sen kommer det inte att förändras så därför kan man låta det vara. Den är väldigt svår, för man kan ju idag tro att något inte kommer att förändras och sen kommer en produktägare in imorgon och säger att det visst behöver ändras. Det är ganska vanligt. I de flesta fall skulle jag skriva något väldigt lätt happy path-test bara för att se till att ha den infrastrukturen på plats och förhoppningsvis så skulle personen som gör de här ändringarna inse att det var dåligt med tester just där och skriva dem då. Så försöker jag göra, att när jag ska göra en ändring på något som är otestat så försöker jag, om det går och passar in i projektkulturen, lägga upp det så att det får någon slags test runt sig.

*30. Det vore ju grymt att ha en sån i teamet. *Skratt**

Mm, men det kräver ju att teamet gör det, inte att en person gör det, för annars kommer den personen att bli galen efter ett tag. Om ingen annan gör det så kommer det att kännas som att alla andra har sönder saker.

31. Hur mycket känner du att du får med dig andra när du gör så här? Behöver du slåss för att det ska bli som du vill?

Det har varit väldigt olika på olika ställen. Allt från ”va, ska man köra testerna också?” till folk som aldrig har kört testdrivet tidigare men som när man berättar fördelarna reagerar ”oh, det här är ju det bästa jag har hört talats om, det är klart vi ska göra det här”, och sen de som redan är medvetna om att det är bra såklart. Jag har varit tvungen att anpassa mig, på några projekt har jag velat göra mycket mer men ingen annan gör det och då tar jag hellre och gör mindre just för att jag annars blir den som inte producerar någon kod, jag skulle bara sitta och skriva tester. Som konsult så tycker jag att det viktigaste är att man anpassar sig till den redan existerande team-kulturen även om jag aldrig skulle sluta försöka övertala folk.

32. Men om man hamnar i en sådan situation att man nästan bara skriver tester, då låter det som att man skriver tester till existerande kod?

Ja, precis. Det skrev jag väl också, att om det bara var ett nytt greenfield-projekt så skulle jag pusha ganska hårt för att få någon form av testkultur i teamet, men om man kommer in någonstans där det finns en massa kod som kanske inte är sådär välskriven, det är då det tar tid att skriva testerna. Då måste man ofta skriva om saker för att få dem testbara.

Om man är den enda som vill göra det så kan det bli lite konstig stämning i teamet. Man sitter i två dagar och skriver om kod så att den går att testa, sen sitter man i två dagar och skriver tester och sen ägnar man en dag åt att faktiskt göra det man skulle göra från början. Om man däremot har ett team där alla är med på det så tror jag att det är en

väldigt bra sak för du kan komma till ett läge då det går långsamt första tiden men ju längre man kommer desto snabbare går utvecklingen för att man blir säkrare på att man inte har sönder saker och man kan skriva om kod så att den blir mer lättförstådd. Det är i så fall en långtidsinvestering, som man inte kan göra som enskild team-medlem.

33. Du nämnde komplexitet också, det framgick inte om det var komplex kod du syftade på eller om det var komplexa tester. Jag gissar att om det är väldigt komplex kod så vill man testa den och om det blir komplexa tester så vill man kanske inte ha dem.

Ja, precis. Det skrev jag väl också någonstans om det här med bra och dåliga tester, att man vill ha så lättförstådda och så enkla okomplicerade tester som möjligt. Om det är ett komplicerat flöde så blir ju testerna oftast komplicerade, men man kan ändå sträva efter att ha så enkla tester som möjligt. Vad gäller att bestämma vad som ska testas så är det nog ändå i första hand komplexiteten i det som ska testas som avgör, både i termer av komplexa algoritmer och affärsregler, det kan vara ganska enkel kod men många olika regler som spelar in. Ju mer komplext desto mer tester.

34. Du skrev någonstans att man ska "lyssna" på testerna. Vad menar du med det?

Det går ihop med att jag tycker att tester är mer ett designverktyg än ett verifikationsverktyg. Låt oss säga att jag har ett test som kräver 70 rader setup-kod för att få alla beroenden uppsatta och jag måste initiera 17 olika objekt med 10 parametrar var bara för att kunna testa att jag får ut "Hello World" på skärmen, då kanske jag har en dålig arkitektur som jag borde förenkla eller förändra på något sätt. När man börjar hamna där så gäller det att lyssna på att det är någonting som börjar bli jobbigt, ta ett steg tillbaka och börja fundera på vad det egentligen är man håller på med och om det går att förenkla. Kanske introducera något nytt objekt som tar hand om lite av det ansvar som jag håller på med nu, eller ändra ansvarsområden för de redan existerande objekten, istället för att bara köra på och pressa in ännu mer saker.

35. Hur tror du att det här hänger ihop med hur enkelt det är att köra tester?

Jag vet inte om det hänger ihop så jättemycket med att köra testerna, eller, på någon nivå gör det väl det.

36. Hur resultatet presenteras.

Ja, alltså...

37. Eller är det mer vid själva skrivandet av testerna som man behöver känna av sådant här?

Ja, fast det är både och, det finns olika nivåer på beroenden. När det handlar om att sätta upp ett enskilt test så handlar det ofta om beroenden mellan klasser och objekt, där har det inte så mycket att göra med att köra dem, men man kan ju däremot ha beroenden på en högre nivå: "Jag måste ha en Java-applikationsserver igång för att kunna köra de här testerna och den måste vara konfigurerad på det här sättet med de här inställningarna". Det tycker jag också är ett beroende på någonting som kräver en massa setup, men då är det ju på en högre nivå. Jag tror att man vill undvika den typen av beroenden så långt som möjligt.

I idealfallet så tycker jag att man ska kunna checka ut koden och köra ett kommando så

ska testerna köra. Det är inte alltid lätt att hamna där, men det är det som är målet, på samma sätt som jag tycker att 100 % test coverage är mål som man ska sträva mot så tycker jag att man ska sträva mot att ha tester som man kan köra när som helst utan att behöva starta upp servrar och liknande. Om man vill ha ordentliga integrationstester så är det oftast svårt, eller omöjligt, men det ska ändå inte kräva att man startar upp en specifik tomcat-server manuellt utan allt ska skötas av test runnern.

38. Jo, att automatisera kan verkligen vara guld värt, då blir det att man kör testerna oftare också, så att man verkligen får ut mer värde av dem, mer feedback. Vi pratade lite om hur det påverkar sättet man jobbar på, om man har tester jämfört med om man inte har det. Om vi tänker oss att du har en kund som tycker att testning är en bra sak, hur påverkar det hur du jobbar?

Om man bortser från alla andra faktorer, så tror jag att man kan jobba mycket mer med refaktorisering och kodkvalitet om man har en vettig testkultur, för att man kan vara mer säker på att förändringar man gör inte har sönder någonting. Det tror jag betalar sig i längden, för man kan hålla högre hastighet.

39. Man blir mer flexibel också.

Ja, den andra biten är att man förhoppningsvis får en bättre arkitektur som gör leder till ett mer flexibelt system. Man kan ta till sig nya krav istället för att vissa saker inte går att göra i det system som man har byggt. Ny funktionalitet kan fortfarande ta tid att implementera, men förhoppningsvis så finns det alltid en möjlighet. Det finns förstås en mängd andra faktorer som också spelar in för att man ska kunna åstadkomma hög kodkvalitet, men en bra testkultur är helt klart ett steg i rätt riktning. Man behöver fortfarande anstränga sig, men det blir enklare att göra det med testning.

40. Vad tror du om att använda tester som ett kommunikationsredskap? Både gentemot kund och nya utvecklare.

Det låter väldigt bra. *Skratt* Jag har aldrig upplevt att det har fungerat.

41. Om du tittar på någon annans kod och det finns tester, kollar du på dem först då eller försöker du förstå koden först?

Jag brukar kolla på testerna, men de brukar väldigt sällan vara skrivna på ett sätt som gör det uppenbart.

42. Det tänkte jag på innan när du sa det här med att man försöker täcka in så mycket som möjligt med sina enhetstester, att det skulle kunna finnas en vits med att skriva ganska enkla enhetstester som inte alls har ambitionen att hitta alla buggar utan istället är tänkta mer som en specifikation.

Jag tycker att om man har ett vettigt testverktyg så kan man dra upp det på olika nivåer, både Buster och Jasmine har ju nästlade describe-block, så att man kan skilja grundfunktionaliteten från specialfallen. Det är något som jag vill bli bättre på för det är riktigt bra när det fungerar, som i vissa Open Source-projekt. Jag har aldrig sett ett kundprojekt där det har varit så.

RSpec och Cucumber har så att deras webbsite är deras features i ett mer webbvänligt format, vilket är häftigt. Vi har även boken Specification by Example av Gojko Adzic

som tar upp mycket av det. Jag tror att det på vissa ställen skulle vara väldigt värdefullt att tänka så, i synnerhet ut mot resten av organisationen men även på enhetstestnivå. Jag vill träna på det mer och faktiskt kunna få till sådana tester som gör att man kan förstå hur allting fungerar direkt utifrån att köra spec-outputen.

Testning är sjukt svårt, jag har försökt köra någon form av testdriven utveckling i sex år och jag tycker inte att jag riktigt kan det ännu. Det kan vara så att dåligt skriva tester är sämre än inga tester, för de är ofta så beroende av implementationen att fördelarna av att kunna ändra på saker uteblir för att en ändring av implementationen gör att testerna slutar fungera. Då har du tester som bara är i vägen, och samma sak blir det när du inte lyssnar på dina tester. Du hamnar i att du har beroenden mellan allting, men i testerna så sätts allting upp och därför går de ändå igenom. Då blir det så att du ändrar en sak och allting ändå går sönder, så att man blir frustrerad över att behöva fixa tester som slutar fungera av helt orelaterade anledningar. I de lägena är det kanske bättre att inte ha några tester alls, och det var där jag var för kanske fem år sedan. Jag hittade kod från den tiden för ett år sedan där vi hade Ruby-tester med 160 raders setup-block för att sätta upp nästan alla objekt i hela applikationen. Jag minns att vi svor dagligen över de testerna för att de kändes i vägen och onödiga. Det är svårt. Det är ännu svårare i JavaScript, för där har man en DOM som har en tendens att komma in överallt.

43. Sista frågan var just vad du skulle ge för tips till någon som vill börja testa sin JavaScript.

Försök inte att göra det på redan existerande kod, är väl det första. I princip enda sättet du kan testa en jQuery-byggd site är med Selenium eller en liknande helt integrerad lösning. Det är inte värt att försöka gå in och skriva enhetstester med en DOM inblandad. Sen skulle jag ta det steg för steg, så att man skriver tester för de delar man är inne och meckar i.

Vad gäller att lära sig så har Katas fungerat väldigt bra för mig. Genom att köra kodkatas så kan man lära sig grunderna, som behövs för att man inte ska få det väldigt jobbigt. Börja med de enkla fallen, som att testa en strängomvändare och liknande. Då slipper du lära dig alla saker samtidigt, och kan istället fokusera på att lära dig testramverket först och sen lägga på saker allt eftersom. Det är också väldigt givande om du har någon som kan mycket som du kan sitta tillsammans med.

Börja någonstans, använd tester i egna projekt eller kundprojekt om det finns tillräckligt stöd från resten av teamet. Någonting nytt då, inte något som redan har massor med JavaScript, för då blir det bara jobbigt.

44. Jag tänker också på det nu när jag kommer in i Live, där det just nu finns en ambition om att börja testa mer. Hur det kommer att gå till där, vad tror du skulle vara lämpligt?

Det är svårt att säga allmänt.

45. Det jag känner spontant är väl just det att man ser till att skriva tester för det man går in och ändrar.

Läs refactoring-boken och jobba med att kontinuerligt förbättra kodbasen, för det räcker

inte med att skriva ett test och sedan känna sig nöjd. Annars kommer du att ha en testsvit på flera hundra tester som tar flera minuter att köra lokalt och då kommer ingen att orka köra dem. Du kanske har en CI-maskin som kör testerna varje natt.

46. Selenium tänker jag försöka undvika, förutom för det absolut mest kritiska.

Det är oftast dit du måste gå om du inte börjar förbättra kodstrukturen, annars blir det ofta så att man inte kommer undan att behöva kolla i sin DOM någonstans. Jag ska inte säga att du inte ska vara rädd för att strukturera om kod, för det kommer man att vara, men gör det ändå. Det spelar egentligen ingen roll om det är .NET-kod eller JavaScript-kod, samma principer gäller ändå. Se till att automatisera allt, för annars kommer ingen att orka köra testerna.

Börja med inställningen att stoppa produktionslinan om testerna går sönder, som i Toyota-fabrikerna. Inte skjuta på att fixa tester som gått sönder. Det har med kulturen att göra, man måste få alla med på det. Det är problem jag haft på många ställen att folk säger att de vill göra det och är väldigt positiva när man pratar om det men sedan så märker man att vissa tester på byggservern har varit röda i nästan ett dygn och får ett svävande svar om att det ska fixas imorgon när man frågar varför det blivit så. Då hamnar man i att ingen litar på testerna efter ett tag, för att man inte vet om det är rött för att något är trasigt eller för att testerna inte har uppdaterats på grund av att någon inte förstod någonting. Så fort man får den osäkerheten så går testernas värde ner ganska hårt.

48. Tack ska du ha!

15.3 Transcript of Interview with Marcus Ahnve

1. Vad har du använt JavaScript i för sammanhang?

Webb, alltså webbapplikationer.

2. Mycket frontend, eller backend också?

Ja, nej, inte alls, jag programmerar allt möjligt, jag flyger lite fram och tillbaka. Jag kommer från backend, men har tvingats lära mig frontend också.

Senaste gången vi körde testdrivet JavaScript var hos en kund (av sekretesskäl kallar vi kunden för X) för tre år sedan. Då tog vi till oss idéer som... (oavslutad mening) Vi hade en del testdriven JavaScript hos en annan kund också (vi kallar kunden för Y).

Det var två relativt avancerade JavaScript-delar.

3. Vad var det som gjorde dem avancerade då?

Det var ganska avancerad vylogik, kunderna frågade efter saker som vi inte hade sett förut, som troligtvis inte hade gjorts tidigare av någon. Kund X skickade nyhetsmeddelanden som kunde gå iväg på två språk, engelska och svenska. När man öppnade ett sådant så skulle det komma upp en flik på sidan med svenska som förvalt språk och möjlighet att lägga till engelska. När man valde engelska så skulle en ny flik komma fram med nya fält, som skulle gå att ta bort.

Det svåra var i första hand att hantera Rails. Med de ramverk som finns idag så hade vi gjort på ett annat sätt, förmodligen med AJAX eller något enklare, men det fanns inte dokumenterat hur man kunde göra det vi behövde göra. Vad vi skulle vara tvungna att göra för att få det här att fungera med Rails var att inputfälten var tvungna att heta något eller ha ett index för att bli tolkat på rätt sätt av Rails. Vi förrenderade HTML och la i ett hidden-fält för att få tillgång till fältens namn, som bestäms av Rails namnkonventioner. När en ny flik skulle skapas, så kopierades allt och ändrades med reguljära uttryck utifrån det högsta id:t just nu och liknande.

4. Och det här gjorde ni testdrivet?

Ja, precis. Då hade vi varit på XP-konferensen i Trondheim (Agile Processes in Software Engineering and Extreme Programming 11th International Conference, XP 2010, Trondheim, Norway, June 1-4, 2010, Proceedings) där det var två som berättade om hur de körde testdriven JavaScript, vilket vi inte hade hört talats om innan överhuvudtaget. Problemet med testdriven JavaScript är ju DOM:en. De var de första jag såg som separerade DOM-access från resten av koden genom att lägga det i ett separat lager. De gjorde inga jQuery-anrop i affärslogiken, utan updaterade vyn som en separat sak. Det fanns då vy-lager där all jQuery och annat hamnade. Det var jag och Jimmy Larsson som satt där och jag minns att vi frågade oss varför vi inte hade tänkt på det förut. Man behöver inte skriva callback-soppa bara för att det är JavaScript.

En sak till som var ännu värre (än anpassningarna som behövdes för att de komplicerade vyerna skulle fungera med Rails) var att vi använde rich text editors, alltså för webben. Det som var grejen var att vi använda två editorer per flik. Jag minns inte vad det var för någon men det var en av de mer vanliga varianterna. Dels använde vi två på samma sida, vilket inte var speciellt vanligt, men framförallt så var det problematiskt att instansiera dem dynamiskt, det var det ingen som hade gjort förut. En sådan editor tar en textarea, gömmer den, och ersätter med ett gigantiskt JavaScript. Sen när du trycker på submit så finns den en master-klass som känner till alla de här, läser ut all text, populerar textarean och skickar submit. Det där var riktigt svårt att hålla ordning på, för det var helt odokumenterat. Det var inte så svårt när vi listade ut hur allt fungerade, men innan dess var det mycket problem med att se till att viss kod kördes överhuvudtaget. Just den biten var kanske inte testdriven, men det tillkom tester på det efter ett tag. Även fast det egentligen bara var textinmatning så blev det ganska komplext ändå.

5. Minns du hur ni testade det?

JsTestRunner körde vi.

6. Jag tänker själva testerna, hur ni...

Vi hade dem integrerade i Rake, genom att lägga till en test-target för JavaScript-testerna som kördes automatiskt. De gick väldigt fort. Jag minns inte om vi la till Evergreen (https://github.com/jnicklas/evergreen) på slutet eller inte. Det var då som Jonas Nicklas på eLabs i Göteborg byggde den här pluginen. Det var någonting med en browser inblandat som skulle startas varje gång testerna skulle köras. Jag tror faktiskt att vi ersatte alla tester efter ett tag med Jasmine.

7. Vad var det som fick er att göra det?

Jasmine är mycket mer läsbart än JsTestRunner och ganska snyggt gjort i jämförelse. Nuförtiden så är folk rätt förtjusta i Jasmine, jag tycker personligen att Buster ser lite bättre ut.

8. Mm, och det var just för att få mer läsbara tester?

Ja, precis. JsTestRunner gjorde att testerna blev väldigt tekniska. Framförallt i JavaScript är verktygen viktiga och där är det stor skillnad idag, att det numera finns vettiga ramverk som CanJS och Angular, som ger en tydlig uppdelning för vad det är du testar.

För tre år sen var det fortfarande mycket hackande av callback-soppa. När folk skrev tester i JavaScript så var inte det mycket annat än att man kickade igång en browser och kollade att rätt sak hände, det fanns ingen separation mellan event och action. Det är den stora skillnaden idag, nu tror jag att det skulla vara lättare att skriva tester. På den tiden var det mer så att om man hade varit riktigt duktig så hade man kunnat komma på att skriva ett CanJS själv *skratt* men det gjorde vi inte.

9. Svårt att motivera gentemot kunden också kanske?

Äh, givet hur mycket tid vi la ner så. Jag är inget vidare på JavaScript, men har gjort ganska avancerade grejer i det ändå måste jag säga.

10. Den här konferensen, var den under tiden som ni höll på med det här?

Nej, det var precis innan. Det var rent flyt, vi insåg fördelarna med att testa JavaScript och bestämde oss för att göra det.

11. Vad hade ni för nytta av testerna?

Trygghet, regression, det gamla vanliga. En känsla för att de viktigaste sakerna fungerade. Även att tvinga fram vettig modularisering, det är ofta det som jag tycker är en av huvudfunktionerna med tester. Om man upplever att något inte går att testa så är det i princip uteslutande så att det är fel på koden.

Det var rätt svårt, jag minns att vi hade en kille som i övrigt är en väldigt duktig utvecklare men som hade svårt att bryta upp sättet som han skrev JavaScript på. Han chokade fullständigt, sa att han inte fattade och började skriva callback-soppa istället. Det var det han var van vid, det var så JavaScript skulle se ut. Function, function, function, function...

12. Hur tror du att trenden var, för några år sen, har det förändrats nu med de nya verktygen?

Ja! Jag har inte någon särskilt positiv syn JavaScript-communityt så som det såg ut, det har tummats mycket på kodkvalitet och det har funnits en attityd som gått ut på ungefär ”skit i det, det funkar”. Jag skulle säga att testdrivet kom från Ruby-hållet, från backend-programmerare som klev in i frontend-programmering. Det stämmer väl med många inom Rails-communityt, Rails har ju väldigt mycket tester och när de började skriva sina skript-grejer så insåg de att de inte hade några tester för det och Jasmine kom av...

13. RSpec?

Precis, och de som skrivit Jasmine är... är inte det FortBot? Jo det är det. (Pivotal Labs?) Alltså, det är ju Rails-gäng. Väldigt många är det så med: BusterJS, PhantomJS, allt det där kommer från folk som har hållit på med Rails och är vana vid att skriva tester. Det existerande JavaScript-communityt har fortfarande en del folk som till exempel person A (onödigt att nämna namn) som inte är så intresserad av tester så länge koden fungerar. Så det beror helt på var man kommer ifrån.

14. När jag började med mina efterforskningar så fick jag intrycket från allt jag läste att det inte testades mycket alls och sen när jag pratade med Johannes Edelstam så lät det på honom som att det börjar komma igång nu.

Jo, Johannes, som för övrigt var med i projektet för kund Y, är ju väldigt duktig. Ska jag vara helt ärlig så är det ju oftast så att om du åker ut och tittar på vissa projekt som drivs av mindre ambitiösa företag så ser du mindre av testning. Jag tror att det handlar om varifrån du kommer, om man mest har ägnat sig åt HTML, CSS och PHP så har man troligtvis inte skrivit så mycket tester, kommer man från Rails, backend och är van vid det och dessutom skriver frontend då funderar man mer över var man ska lägga testerna.

15. Frågan är då ifall det har förändrats

Ja, jag tycker att det har förändrats. En skillnad är att det överhuvudtaget finns. Jag tittar på Angular som säger att det går jätteenkelt att testa. CanJS, som jag tycker bättre om, är också väldigt enkelt att testa. För mig är det mycket positivt.

16. Vad tycker du kännetecknar bra tester?

Utan att läsa utifrån någon form av definition av bra tester som Pragmatic Programmer.

17. Utifrån din erfarenhet

Jag tycker att tester ska vara snabba, små och testa en sak. Idealet är en assert per test, men det behöver inte alltid vara så. De ska vara meningsfulla, det finns en del människor som skriver tester som inte är det. Deskriptiva, jag är förtjust i idén bakom BDD, att testerna ska beskriva ett beteende i systemet, varför det gör saker. Folk som inte tänker i BDD-termer har en tendens att skriva tester som testar metoder, till exempel testAddition.

18. Vad är det du menar när du säger att tester ska vara meningsfulla?

De måste ha någon form av relation till det affärsvärde som systemet löser. De måste på något sätt beskriva någonting. Ett exempel skulle kunna vara att någon ska skriva en metod som heter add(x, y) och bestämmer sig för att skriva ett test som heter testAddXY med tre asserts som kollar att 3+2 blir 5 osv. BDD-sättet är ju då snarare att säga att 2+3 ska bli 5, det är ett exempel. Fördelen är att när ett test failar så får du ut vettiga felmeddelanden. Om du har ett felmeddelande som säger testAdd failed så är det svårt att veta varför det failade och man måste lusläsa testkoden för att förstå. Just testAdd kanske är lite väl rudimentärt, men det kan till exempel vara testUpdateUser failed, då

undrar man vad som menas med att uppdatera en användare. Om testet däremot heter ”User should be saved to database” så är det mycket tydligare vad som gick fel.

19. Det är väl också mycket det att man har möjlighet att specificera förutsättningarna: Given, When och de där.

Ja, kontext är väldigt bra, det är också en fördel med BDD. Särskilt i Java kan den delen saknas i traditionella TDD-ramverk, det är inte ett lika stort problem i Python och Ruby, för då kan du skriva fler testklasser i samma fil. Då är det inte ett lika stort problem att du är begränsad till en setup och en teardown per test. I Java har du ofta så att det är en klass som mappar fram och tillbaka, då har det åtminstone för mig inneburit problem att det bara finns en setup.

20. Den blir ganska stor

Ja, jag kan göra en koppling till J. B. Rainsberger, som har skrivit JUnit Recipes: Practical Methods for Programmer Testing, en av de bästa böckerna om jUnit, för nästan 10 år sedan. Han körde en dragning på XP 2011 i Madrid om hur man jobbar med BDD i vanliga jUnit. Hans poäng är att man ska sluta testa klasser och testa beteenden på system istället, möjligen implementerat genom att testa klasser men i första hand så ska man sträva efter att beskriva systemets beteende. Det är egentligen det som jag har velat göra med Cucumber, att få hjälp med att testa utifrån. En liknelse jag brukar göra är att du ska undvika att sitta bredvid processorn med sladdar och grejer och tänka att du testar tekniska saker för att du är en tekniker. Det är mycket intressantare vad systemet gör för att fylla affärsvärde.

Jag börjar luta åt att vi bör skriva mer och mer enhetstester som beskriver affärsvärden men som implementeras på enhets-nivå. Jag tror att det finns problem i att vi bara beskriver affärsvärden i integrationstester och att vi fokuserar för mycket på implementationsdetaljer och lågnivå-saker i våra enhetstester.

21. Hur uppnår man det, jag tänker att man behöver vara rätt så konkret när man skriver enhetstester? Exemplet tidigare om att uppdatera en användare i databasen är väl ganska bra i och för sig.

Det finns ju en poäng med integrationstester, men de ska nog i första hand vara happy path och smoketest, men det ska mest handla om ifall grejerna sitter ihop. Jag kan fortfarande bli irriterad på gamla ”Uncle Bob-människor” som tycker att man aldrig ska testa GUI:t. Varför inte då, om det är det du vill testa? Om GUI:t är viktigt så tycker jag att man ska testa det. Min gissning är att attityden beror på att man är inlåst på att man skriver en viss sorts system, och generaliserar för mycket.

De som började skriva TDD en gång i tiden jobbade i första hand i finansbranchen, där var GUI:t oviktigt och affärsregler oerhört viktiga, så de skrev mycket tester för just affärsregler. Sedan så kom TDD ut i webbvärlden där affärsreglerna inte är så framträdande men istället så har man en stor mängd flöden, som är viktiga. Att då utgå från att man inte ska testa GUI:t blir närapå absurt, för resten är ju ofta närapå trivialt så då behöver man knappt några tester.

Att man behöver enhetstester är en annan sak. Om vi tar CanJS som exempel, där du kan ha en explicit vy som tar en template och genererar HTML. Det går att testa

genom att parsea den genererade HTML-koden och se om specifika värden dyker upp. Man kanske behöver mocka vyn, men i övrigt så är det precis så man vill ha det.

22. Jag har bara provat Angular än så länge, jag ska nog kika lite på Can sen.

Can är snyggt och jag tycker om det för att det är så modulärt. De har routing, som går att använda helt oberoende av allt annat i ramverket. Om man vill använda deras controllers så går det också, och blir enkelt. Det tog lite tid för mig att förstå deras dokumentation därför att när jag läste om deras routing så tog det ett tag att komma till avsnittet om hur man gör om man även vill använda controllers, och då kändes allt mer intuitivt. Samtidigt är det bra att de separerat på det viset, eftersom det blir mer modulärt, lättviktigt och fint, vilket jag tycker om.

Jag har tittat lite på Angular och fått det beskrivet för mig. Den bild jag fått är att det är rätt så mycket "all or nothing", det går inte att plocka bara de delar man behöver och vill ha. CanJS skiljer sig på det sättet att du kan använda det för att strukturera upp din jQuery-kod med lite mustache-templates och sen är det bra så. Sen är det snygg kod, om man dessutom lägger in requireJS så blir det ruskigt snyggt.

23. Drog ni några lärdomar från projektet med kund Y?

Ja, lärdomen var väl att när man väl såg Backbone och liknande komma så insåg man rätt snabbt att de fyllde behov som man tidigare behövde brottas med. När man skriver stora mängder JavaScript-kod så blir det oerhört viktigt att den är välstrukturerad. Johannes kom in i det projektet och fixade saker som hade med JavaScriptet att göra långt efteråt att det hade skrivits. Han sa att det hade hjälpt honom att det var testdrivet.

Vi tog över projekt från en annan organisation och det var en märklig upplevelse, ungefär som att han som hade skrivit koden hade läst en kurs i hur man strukturerar JavaScript men inte förstått varför man ska göra det. Det var inte speciellt svåra saker som koden skulle göra, man hade bara gjort dumheter såsom flera hundra rader kod enbart för validering av fält.

Klasser som dolde funktioner och publicerade publika grejer i slutet, jättestrukturerat och jättefint bara det att när man till slut lyckades ta sig igenom det här så insåg man att det ändå var hårdkodat mot specifika fält och det gjorde det bara jobbigt, det hade varit enklare att hantera en callback-soppa, då hade man i varje fall kunnat gå in och dra isär den. Det fanns naturligtvis inga tester, och det slutade med att vi var tvungna att göra ganska mycket fulhack. En av de fulaste saker jag gjort någonsin faktiskt var då när jag gick in mitt i hans 300-rader långa funktion och la till ett villkor att valideringen endast skulle köras om ID:t hade ett visst värde. Det blev hårdkodat för att jag inte orkade bryta ut det när det var så fruktansvärt mycket kod.

Andra grejer han hade gjort var att en knapp tändes på onBlur om ett visst inmatat ID var korrekt. När kunden kom med ett önskemål om att fältet skulle vara förifyllt så försökte vi lösa det genom att skicka in det med en url-parameter och populera fältet men återigen hade vi situationen att det var hårdkodat. Så vi tog faktiskt page.onLoad och anropade onBlur. Jag reagerade med att fråga mig själv om man överhuvudtaget fick göra så, är det lagligt? Det gick bra och löste problemet men det kändes ju inte bra. Det var ett praktexempel på hur JavaScript-communityt kan fungera. Han ville nog väl

men det blev väldigt fel.

Det finns ett problem med folk som kodar JavaScript och aldrig har sett någon annan kod, som definierar sig själva som frontend-utvecklare och har attityden att de inte rör backend-kod. Jag tror att det är en nackdel för kodare generellt sätt, för hur du kodar. Det är nyttigt att få se andra språk, kunna använda det och få den inputen. Om du går in för att enbart läsa och skriva JavaScript i resten av ditt liv så kommer du att få så begränsad input att du inte förstår varför du ska strukturera din kod och utan de insikterna så blir det inte bra även om du försöker.

24. Så budskapet blir att det är bra att strukturera sin kod, men att man behöver se till att man vet varför man gör det.

Ja, absolut.

25. Ett liknande resonemang är att det är bra att testa, men att man bör se till att man vet varför man gör det. Se till att alla vet varför.

Ja, men samtidigt så beror det på. När du är nybörjare så saknar du den kontextuella kunskapen som krävs för riktig förståelse. Då får man finna sig i att göra som man blir tillsagd, som i kampsportsträning där du som nybörjare inte ska ifrågasätta varför din tränare säger åt dig att slå på ett visst sätt. Det är först när du kommer upp i en högre nivå och har gjort något länge som du kan börja ifrågasätta och till slut även komma på egna förbättringar. Att bara testa när man förstår varför gäller egentligen bara de utvecklare som vet varför de ska testa. Om du aldrig har testat så kan du inte avgöra när du bör göra det och när du inte bör göra det. Jag skulle föredra att man alltid testar mot att man aldrig gör det, tills man kommer till den punkt att man vet när man inte ska göra det.

Idealet är förstås att sätta nybörjare med erfarna människor, så kan de erfarna bestämma och förklara varför det ser ut som det gör. Det blir många iterationer och mycket jobb om en nybörjare ska behöva återuppfinna hjulet helt själv.

26. Vad är din syn på testning av JavaScript jämfört med andra språk, är det någon skillnad i hur du skriver tester?

Ja, det borde vara det. Hittills har det nog inte varit det, jag har jobbat väldigt objektorienterat och jobbar nu gradvis med att bli en bättre människa och skriva mer funktionell kod. Förhoppningsvis kommer det att leda till att jag skriver mer funktionellt baserade tester, och därigenom slippa hålla på och instansiera klasser och liknande. Annars är det inte så stor skillnad, jag försöker att tänka likadant.

27. Har du upplevt att det varit svårare eller enklare?

Om det har varit svårare så har det har nog mest att göra med att jag är bättre på TDD än på att skriva JavaScript. Nuförtiden behöver det inte vara någon skillnad, för tre år sen var det definitivt svårare för det fanns inte några exempel att titta på. Det var inte lika väl dokumenterat hur man hanterar specifika fall och det fanns mindre vettiga ramverk. Förr så hade du en massa jQuery som kördes på document.onLoad och då var det verkligen inte lätt att veta hur man skulle bära sig åt för att testa den koden. Det var märkligt att ingen kom på idén att ta fram ett MVC-ramverk för att

lösa problemen.

28. Och sen exploderade det med sådana ramverk.

Ja, precis.

29. Vi pratade lite om ambitioner med testning tidigare, om man väljer att fokusera på integrationstester eller enhetstester, min fråga som jag hade här sen innan var: vilka brukar vara dina ambitioner när det kommer till testning?

Jag brukar testa utifrån och in. På så sätt kan jag identifiera vad det egentligen är jag vill göra. Det behöver inte utgå från klienten, däremot så strävar jag efter att tänka i termer av APIer, vad varje del ska göra.

Ett praktexempel var en gång när jag var med och arrangerade en konferens för Agila Sverige, då vi skulle göra en schema-snurra. Jag gjorde det klassiskt objektorienterade misstaget att jag började modellera upp en jättesnygg objektmodell för schemat som skulle gälla i två dagar med två parallella tracks, med olika tider och lokaler. När jag väl skulle börja använda modellen i HTML så insåg jag att det var helt värdelöst. Jag blev så arg på mig själv, jag höll ju kurser i BDD och annan testdriven utveckling och gjorde det nybörjarmisstaget själv. Det jag borde ha gjort var att börja skriva gränssnittet och fråga mig vilka APIer jag vill ha, istället för att börja med något i förhoppningen om att det kommer att behövas. Det tankesättet går att tillämpa generellt när det gäller utveckling.

30. Så de första testerna du skriver är...?

Jag skulle göra som så här, att om jag skriver en HTML-sida så börjar jag med att skriva HTML-koden snabbt och enkelt och utan tester. Sedan kommer jag till en punkt då jag inser att jag skulle vilja ha ett API-anrop, idag om jag får välja så händer det i en mustache-template. Då skriver jag kod som anropar något som inte finns, och ett test för det. När man väl kommit dit så vet man att det är en komponent som behövs och då kan man börja skriva den, och även testa den, eftersom man då vet vad den ska göra.

31. Man har ofta som princip att alla tester ska gå igenom hela tiden. Det finns många bra orsaker till det, man ska kunna lita på testerna och så. Samtidigt kan jag känna att om man har det här tillvägagångssättet att man jobbar mer top-down, så skulle man vilja skriva tester som är på hög abstraktionsnivå och i princip kräver att systemet är klart innan det går igenom.

Absolut. Så gör jag jämnt. Och om du tittar på gammal klassisk XP-terminologi så kan ett integrationstest faila ganska länge. I RSpec så finns ett koncept som heter Pending, man låter den vara grön till exempel genom att säga att den ska faila därför att det inte är klart. När du väl är färdig så behöver du ändra på testet. Alternativt så kan du lista den som Not Done eller något. Pending tycker jag är bra, för då får du bra indikationer, men sen så ska du väl inte ha så värst många pending liggandes.

32. Det kan vara bra för att få en tydlig riktning kan jag tänka mig.

Jag tycker det, jag tycker att det är en av de bästa sätten att jobba på. Däremot så kräver det ganska mycket disciplin. Om du ska skriva en helt vanlig HTML-sida eller gör

något som i princip enbart är boilerplate och du vet med dig att det kommer att vara många ögon på den koden i projektet så kanske det inte är värt att skriva ett långsamt integrationstest för det.

Det är väldigt trevligt med mustache och liknande som gör att du får ut logiken i riktig kod istället för att du har en massa märkliga if-satser i vyerna.

33. Du nämnde att det kräver mycket disciplin. Det har ju lite att göra med ens personlighet som utvecklare, tror du att testning är något för alla? Om man inte är en sådan person som har den disciplinen.

Nej, det tror jag inte. Jag tänkte säga ja, men sen insåg jag att det är fel. Om vi tar Notch (Markus Persson, skapare av Minecraft) som exempel, han skriver inga tester. Det vore fruktansvärt drygt och dumt av mig om jag sa att han borde skriva tester. Nej jag tror inte att det är för alla, jag tror att det går att skriva alldeles utmärkt bra kod utan det, däremot så tror jag att du skriver bättre, mer underhållsbar och mer modulariserad kod med tester. Det kan mycket väl vara så att du är en tillräckligt duktig programmerare, som skriver modulär och vacker kod även utan tester, men för mig är det i varje fall så att det är ganska uppenbart när folk inte använder tester.

Ett praktexempel är Android, det är uppenbart att de som gjorde Android-API:et inte använde sig av tester, för då hade de inte gjort det så som det är gjort.

34. Vad är det man ser det på, rent konkret?

Det är en gigantisk arvshierarki, som gamla klassiska J2EE, där du inte kan instansiera en servlet med mindre än att du har en applikationsserver tillgänglig eftersom den krävs längre upp i arvskedjan, eller JSP-sidor som ska kompileras av en parser som är inbyggd i din applikationsserver. Det här kom att utmanas av Freemarker som fungerar mer fristående. Mata in data, ange vad för template du vill ha och så får du HTML som du kan göra vad du vill med.

35. Lite enklare att testa.

Väldigt mycket enklare att testa, väldigt mycket enklare att förstå vad den gör. Det finns fantastiskt många människor som vill hjälpa stackars programmerare att inte förstå vad det är som händer.

36. Vilka ser du som de största riskerna med testning då? Om vi fortsätter på spåret om vilka som borde testa och inte. Det beror kanske lite på vilket projekt det är.

Det finns två risker med testning, det ena är att man litar för mycket på det, att man tror att det räcker med att skriva tester och att man inte behöver göra något annat. Att det gör all manuell testning överflödig. En annan risk uppstår i och med att tester precis som all annan kod blir gammal och dålig om du inte underhåller det.

Jag kom till ett projekt med Microsoft-utvecklare en gång där chefsarkitekten mycket stolt sa att de hade 2000 fitness-tester. Vad trevligt. Vad roligt. Och så skulle jag köra dem, och fick 1000 fails. När jag frågade varför så svarade han att testdatabasen inte var ordentligt uppsatt. Då frågade jag när den senast var uppsatt så att alla tester fungerade och det var det ingen som mindes. Då är det bara slöseri med pengar. Det hade varit bättre att inte skriva testerna överhuvudtaget.

Sen har du det att du behöver ha en vettig testmiljö, som fungerar. Det är inget fel på Microsoft-utvecklare, men de har en tendens att göra vissa saker på fel sätt ibland, som till exempel att alla delar på en databas för att man inte orkar sätta upp SQL Server till alla. Det kan uppstå absurda situationer där en utvecklare meddelar de andra att testerna kommer att sluta fungera för att personen ifråga ska lägga till en tabell eller ändra namn på en kolumn. Det beror ofta på att det är jobbigt att ge alla varsin och att de inte får byta till en annan databas.

37. I de projekt som du håller på med nu, skulle du vilja använda testning i alla?

Ja, absolut. I mitt nuvarande projekt så är det inte överdrivet mycket JavaScript just nu men det kommer nog att bli det så jag hoppas att det börjar testas även om jag går vidare till ett annat projekt.

38. Jag har tänkt göra någon slags analys av ekonomiska aspekter av testning, hur mycket det lönar sig, men det är lite svårt att mäta tycker jag.

Det är sjukt svårt, jag kan bara gå på magkänsla. Det bästa exemplet jag har är från när jag jobbade på en startup för ganska så länge sedan, där vi sålde mjukvara som skulle köras på Websphere-servrar, på ett visst antal databaser och ett visst antal operativsystem. Vi hade i princip 100 procent test coverage, och testservrar uppsatta med olika konfigurationsvarianter som kördes vid varje bygge och gav oss en tabell över vilka tester som gick igenom på vilka maskiner. Vår VD sa att det var vår bästa investering någonsin, för han visste alltid var produkten var vilket gav en enorm trygghet när han gav sig ut för att möta kunder. Det är i princip omöjligt att sätta fasta pengar på det, det är många som inte ens vet om att deras utvecklare testar.

Det har mycket att göra med vad som händer om det uppstår ett fel, graden av allvarlighet om en bugg uppstår i produktion. Om du har ett system som styr kylvattnet i ett kärnkraftverk, navigering av flygplan eller liknande så är det inte möjligt att förlita sig på enhetstester och continuous delivery. Man måste ha både automatiska tester och rigorösa manuella testningsrutiner, med flera olika personer som tittar på det regelbundet och med olika perspektiv, annars blir det livsfarligt.

Kund X är ett praktexempel. Samtidigt som vi hade vårt projekt hos dem så hade vi ett projekt hos en mediekoncern där det förelåg ungefär samma ekonomiska risk kopplad till buggar. Med systemet vi byggde åt kund X så skickades det ut finansiella nyheter åt börsbolag, vissa av dem var noterade på Nasdaq OMX vilket innebar att varje finansiell nyhet enligt lag var tvungen att nå ett visst antal människor, vilket var anledningen till att systemet fanns överhuvudtaget. Det stora med det här systemet var inte att vi la ut information på vår egen webbsida utan att det skickades till 7000 ställen: finansinspektionen, dagens industri, bloomberg, ... Allt gick ut till alla ställen. Man kunde ange exakt tid då ett pressmeddelande skulle skickas. Om det skulle gå fel så innebär det rejäla böter för att marknaden inte blivit informerad, för det är så stora pengar inblandade.

Bortsett från att vår produktägare hos kund X var inne och tittade inför varje release som ett smoketest så hade vi ingen manuell testning med protokoll eller liknande. Vi hade en testmiljö och en produktionsmiljö, med möjlighet att rulla tillbaka ändringar från produktion vid behov. Att släppa ut till produktion gick väldigt fort. Jag levererade

koden och så hade de en driftavdelning som insisterade på att göra saker själva, som ofta gjorde fel. Men det hade egentligen inte med vår testning att göra. Då litade vi på testerna.

Att jämföra med historien hos mediekoncernen, där projektledaren var en gammal testledare. Man ska alltid komma ihåg att tänka på vad folk bryr sig om när de träffar sina "peers" (kamrater, jämlikar). För det pinsammaste man kan säga på en presskonferens som testare är ju att man givit ut ett system med odokumenterade fel. Den ekonomiska risken i det här projektet var i första hand att man kunde gå miste om prenumerationskunder, vilket troligtvis var mindre allvarligt än i fallet med kund X. Ändå så hade de en process som i varenda release låg i form av en "hardening sprint" som gick på att testa i två veckor så att allt var perfekt innan det gick iväg. En av mina kollegor som jobbade i det projektet sa att det var som att sitta i en Ferrari och köra i 30 km/h.

Som affärsutvecklare måste man räkna på vad en bugg kostar i jämförelse med att skjuta på releasen i två veckor. Alternativet är att släppa funktionalitet så fort som möjligt och rulla tillbaka om det uppstår en allvarlig bugg. Det finns en felaktig konservatism i många delar av vår bransch som säger att det inte får finnas buggar. Om man frågar en person med den inställningen varför man inte bara kan rulla tillbaka så tenderar man att få som svar att det är för mycket jobb inblandat med att göra en release, och anledningen till att det är så mycket jobb är att man är så noggrann för att undvika att det blir fel. Ett cirkelresonemang som leder till att varje release blir jobbigare än föregående. Jag tycker att det ska vara löjligt enkelt att göra releaser, så enkelt som möjligt. Naturligtvis finns det exempel som banker och liknande verksamhet där man kan förlora extrema mängder pengar på en allvarlig bugg, där man behöver vara mer försiktig men i allmänhet så är det så här.

Jag noterade igår att SEB lyckats deploya en version av sin Internetbank som det inte gick att logga in på med dosan, enda sättet var om man hade mobilt bankID. De kan inte ha haft tester. När man matade in sitt personnummer och tryckte tabb så fylldes password-rutan med personnumret, hoppade du ner och skrev i passwordrutan så fylldes personnummerrutan. Det kan inte ha varit testat överhuvudtaget, inte ens manuellt. Nu var det här förvisso Chrome på Linux men det lär ändå ha varit så att det inte fungerade på något Chrome och då blir jag ännu mer nervös om det är så att de bara testade på explorer. Man vet aldrig, vår bransch upphör aldrig att överraska.

39. Jag tänker att det här kan ha att göra med val av verktyg, att man behöver ta med i beräkningarna om de möjliggör testning på olika plattformar. Med Selenium kan man parallellisera upp sina tester på en grid och olika test drivers är olika kompetenta när det kommer till det där.

Jag har inte haft tillfälle att bygga upp någonting stort, men i en större organisation så skulle jag nog vilja sätta upp någonting sådant.

40. Jag har tänkt att jag kanske borde intervjua personer med mindre koppling till valtech delvis av den anledningen.

Här på valtech så har jag i första hand testat i Chrome och Firefox och sen fixat till det till Explorer på slutet. Förhoppningsvis så är vi av med IE7 snart, men det är ju ändå inte förrän i IE10 som det börjar bli bra.

Ibland kan man lyckas, vi lyckades fantastiskt bra hos kund X och det var ändå tre år sedan. De skrev ”optimerat för Chrome, Safari och Firefox”, jag ville att det skulle stå ”byggt med moderna webbstandarder”. På den tiden fanns IE6 fortfarande i betydligt större utsträckning än idag och det vi gjorde var att vi lät rundade hörn vara kantiga och även om det här inte har med tester att göra så är det återigen en kostnadsfråga. Det blir en affärsfråga att avgöra om det är värt att sitta och lägga in gif-bilder för varenda bild.

41. Som ska transformeras och positioneras så att de hamnar rätt, och fungera i alla operativsystem.

Ja, eller så använder du CSS för rundade hörn på ett sätt som inte fungerar i gamla versioner av explorer. Då kanske kunden svarar att det inte går för sig och tycker att det är viktigt trots att det bara gäller 2 procent av besökarna, då brukar det hjälpa att komma med en uppskattning av vad det faktiskt kostar att implementera dessa rundade hörn ordentligt, det brukar få kunden att säga ”de får fan ha kantiga hörn”.

42. Då gäller det att kunna säga vad det kostar.

Ja, men det är inte så svårt. Det blir ju aldrig exakt, men man kan ändå säga att om man ska hålla en sådan sak vid liv genom hela projektets gång så kan det mycket väl innebära att det läggs totalt fyra veckors arbetstid på att se till att de rundade hörnen ser ut som de ska i IE6. Då brukar reaktionen vara att man inte vill ha det.

43. Det är intressant det här med kulturen, att det gick bra med kund X men att projektledaren med testarbakgrund hos mediekoncernen kunde ställa till med sådana problem.

Ja, jag vet ju inte hur de gjorde med just rundade hörn, men...

44. Nej men just med processerna vid release.

Ja, man ska alltid komma ihåg att folk bryr sig väldigt mycket om sitt eget skrå. Vad kommer folk vilja visa upp nästa gång de byter jobb? En AD kommer alltid att vara fokuserad på att det ser bra ut, om de gör något som inte ser bra ut så kan de inte visa upp det, UX:are samma sak. Testare vill inte ha något med buggar. Alla vill göra sin grej och har sin yrkesstolthet.

Egentligen är allt underordnat målet att tjäna så mycket pengar som möjligt. Folk har mer eller mindre svårt att förstå det. Jag har sprungit på UXare och ADs som hävdar att det de gör är affärsvärdet. Nej det är det inte, svarar jag då. Jo, det är det, det är ju vad folk upplever, användarupplevelsen är viktigast av allt. Nej det är den inte. Jo, men annars kommer folk inte att tycka om det! Jo men då är det dåligt, då börjar vi prata ekonomi, men huruvida folk upplever det som vackert är egentligen ointressant eller i varje fall underordnat huruvida de sen bestämmer sig för att betala pengar för det och att den som tillhandahåller tjänsten tjänar något på det. Nu har vi kanske lämnat JavaScript-testning lite långt bort.

45. Det är ändå intressant, just kulturen och det tänker jag nog skriva om i uppsatsen att folk behöver vara överens om hur mycket man ska testa, vad det är man ska testa, så att det inte blir konflikter inom teamet och med kunden.

Det kommer det att bli ändå. Jag har väldigt svårt att se ett JavaScript-projekt som lyckas på det sättet. Här på valtech kan man nog få till det ganska bra i och med att det finns gott om JavaScript-utvecklare som har förstått och vet vad testning handlar om. Om du däremot tittar på ett godtyckligt normalt frontend-projekt så kommer det oftast att finnas svårigheter att driva igenom en testningskultur, eftersom JavaScript-communityt har sett ut så traditionellt sett. Den testkultur som finns kommer från folk som har programmerat mycket i andra språk, till största del i varje fall.

46. Vad skulle du ge för tips till någon som tycker att det är riktigt svårt, som menar på att det inte går att testa gränssnitt? Att det bara blir långsamt och jobbigt?

Jobba med någon som vet hur man gör. Jag tror inte att det finns några riktigt vettiga böcker om att testa JavaScript så antingen det, eller så låter man helt enkelt bli.

47. Vad tror du om kod-katas?

Grymma, det är bara att köra. Det är faktiskt intressant, jag har nog aldrig sett en kodkata gjord i frontend-kod, det kan jag ta upp med Emelie, det är en bra tanke.

48. Då tycker jag att vi avslutar med den tanken.