Contents

Glossary

Automated Deployment Auto-deploy enables automatic release to test or production, possibly as part of a Continuous deployment strategy. 7

BDD Behaviour-Driven Development (BDD), see section 4.3 for a definition. 4

CI Continuous Integration (CI) is a practise based on frequently merging new code with a main code repository, commonly using principles such as automated build, testing and deployment. 7, 16, 28, 41, 51

DOM The Document Object Model (DOM) is a model for the content, structure and style of documents [1]. It can be seen as the tree structure of elements that HTML code consists of. A common use of JavaScript is DOM manipulation, which means dynamically changing attributes and style of the elements, or adding and removing (groups of) elements. 8, 11, 35, 49, 50

DRY 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. [2, questions 69-70]. 19

JS JavaScript (JS) is a scripting language primarily used in web browsers to perform client-side actions not feasible through plain HTML and CSS. It is formally defined in the ECMAScript Language Specification (5.1 version) [3], ISO/IEC 16262:2011. 4–18, 20, 21, 24, 26–29, 33–36, 38–40, 44, 48, 50–55

Matcher BDD terminology for an assertion. Also commonly called expectation. 41, 43, 45

MOOC "A MOOC [Massive Open Online Course] is an online course with the option of free and open registration, a publicly-shared curriculum, and open-ended outcomes. MOOCs integrate social networking, accessible online resources, and are facilitated by leading practitioners in the field of study. [...] The term came into being in 2008, though versions of very large open online courses were in existence before that time." [4, p. 10]. 12, 13, 21, 41, 43

SML Standard ML (SML) is a functional language that performs type inference at compile time and has few mutable state features, used primarily in computer science and mathematical research. SML is formally specified in The Definition of Standard ML [5]. 4–9, 11, 12, 21, 39–46

Spec (specification) BDD terminology for a test. 10, 16, 30, 43–45

TDD Test-Driven Development (TDD), see section 4.2 for a definition. 10

Test Fixture A test fixture is an environment or a fixed state that is needed for certain tests to run. It can be set up and teared down for each test or set up once and used multiple times, requiring tests to clean up after themselves to avoid problems of shared mutable state but providing a slight increase in speed [52, question 5]. 35, 37, 52

Testing Defined here as automated software testing, unless otherwise specified. Manual testing and most forms of acceptance testing is outside the scope of this thesis. 4

YAGNI The XP (Extreme Programming) principle YAGNI (You Ain't Gonna Need It) is about not introducing anything except if you are entirely sure that it will be needed. 15, 39

1 (ML1,JS1)Introduction

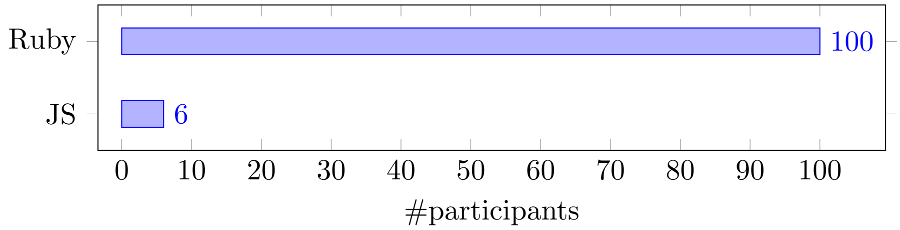
Imagine the following scenario: You have been working for many months on a medium size web application project, with demanding technical challenges, using a framework that was previously unknown to you and with constantly changing requirements. People have come and gone from your team and there are parts of the code that no one in your team dares to modify, because no one understands it and too much of the functionality depends on it. Over time, patches and hacks have spread to all over the code base and it feels as if for every bug you fix, new ones are introduced and the complexity of the application is constantly increasing. Every few weeks you feel unbalanced and nervous about the next release, with frightful memories fresh in mind and a feeling that you should be able to do better.

Now envision this: Despite the challenging requirements and conditions, you are relatively sure that the application works as it should. You feel safe in telling the customer when a feature has been implemented because the automated tests indicate that it works and that nothing else has broken. The application has a modular design and you have a good feeling of what every part is supposed to do and how the system works as a whole. This makes it easier to implement change requests and you spend relatively little time debugging, because the tests generally give you precise indications about which parts of the code are affected by your changes. Whenever there is a bug, you capture it with tests so that you will easily notice if it is re-introduced. Releasing a new version is simple and you feel proud of being part of the team.

The main difference between these scenarios is that the second one requires a pervading testing effort from the team. In this thesis, obstacles that make testing difficult have been investigated. Many of the topics discussed are applicable to any programming language, but it was decided to look at JavaScript (JS) and Standard ML (SML) specifically because the automated testing community around JS still has some ground to cover [6, p. xix] and for SML, a less known functional language, the situation is even more severe. SML and JS both have problems with testing culture, but for different reasons. Client-side JS testing in particular is a concern shared by many, and until recently there was no Behaviour-Driven Development (BDD) framework available for SML. The difference in testing efforts between programming languages is evident when comparing JS to other programming communities such as Ruby and Java. As illustrated by Mark Bates [7]:

> "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."

Percentage of crowd testing their code [7] (Boston Ruby Group presentation, 2012):



The main goal with this thesis was to investigate why JS and SML testing has been performed to such a small extent historically, describe some benefits and practises of doing it, and identifying problems that are yet to be solved.

1.1 (ML1,JS1)Motivation

Why testing, you may ask. Jack Franklin, a young JS 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. [8] Without tests, the ability to refactor (see section 4.8) is greatly hampered and without refactoring, making changes becomes harder over time, the code becomes harder to understand, the number of bugs increases and more time will be spent debugging [9, p. 47-49]. In order to avoid this, code should be tested, preferably as an activity integrated with the rest of the development rather than seen as a separate task. Writing tests first ensures testability, which may also imply adherence to principles such as separation of concerns and single responsibility [36, p. 35-37].

Tests can also serve as documentation and monitoring of the code. Given that the tests are readable and focus on the behaviour of the code, developers can rely on them to understand the production code that they are unfamiliar with or has not worked with for a long time, and measure progress in terms of implementing acceptance tests. There are also benefits for the product management: awareness of how the product performs on different platforms and software environments is reassuring when communicating with customers [10, question 38].

Figure 1 (page 6) can work as an analogy for how testing influences development pace. Developers writing code without tests are like pure sprinter swimmers, they will be fast in the beginning of a project but over time the increasing complexity of the code will force them to go much slower. Developers that writes tests as part of the development process are more like pure distance swimmers, they maintain a sustainable pace. It is arguably slower in the beginning, but productivity does not decrease as drastically over time. The analogy is not perfect – development pace is typically more dependent on system complexity than swimming speed is on distance, but this on the other hand only serves to further emphasise the point. Testing is a long term investment.

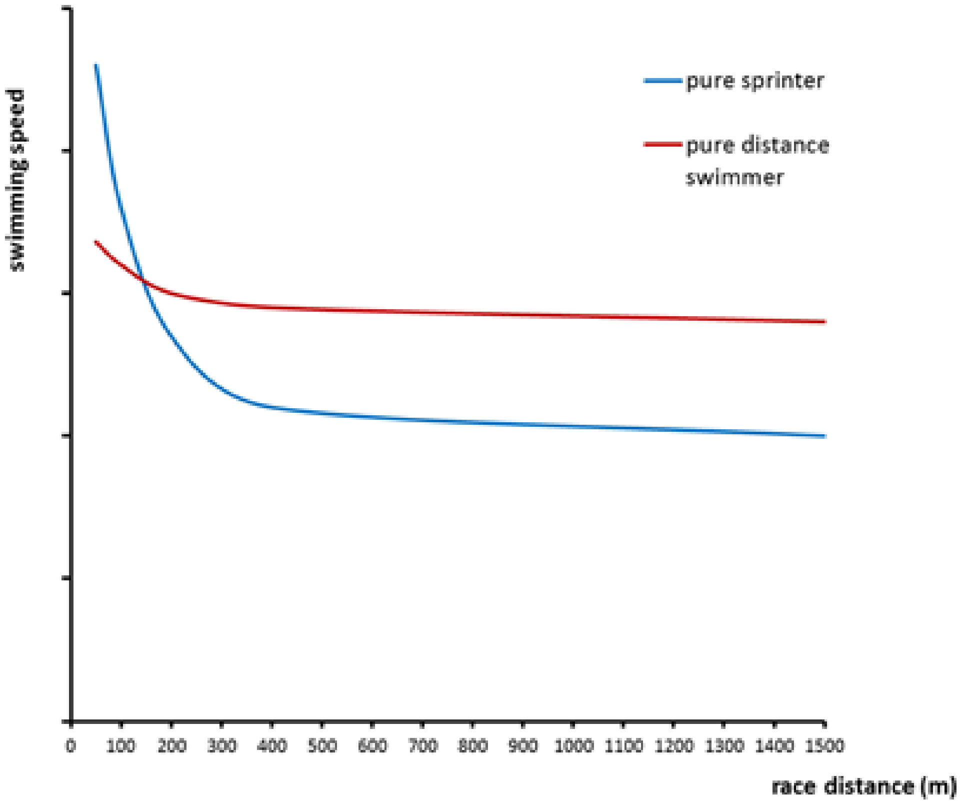


Figure 1: Long distance swimmers are initially slower but have better endurance than sprinters. Image courtesy of Paul Newsome at `www.swimsmooth.com`

Implementations of JS differ, especially for host objects such as document and XML-HttpRequest whose semantics are not fully defined in the specification but also for native objects that may be missing or behave differently in some implementations because they were not defined in previous specifications. Automated testing can help you discover bugs that appear due to such deviant behaviour in present and future versions. JS is also particularly important to test due to its dynamic properties [11] that complicate static analysis (however, there are lint tools, see sections 4.5 and 4.8), unwieldy syntax and aberrant object orientation support. JS is a complex language that requires many work-arounds because of its scoping rules and other unexpected behaviour [12, appendix A]. Despite the wide variety of testing frameworks that exists for JS, it is generally considered that few developers use them and instead rely on manual testing [11].

Several implementations of SML exist and just as with JS, there are some differences between them. SML has extensive static analysis capabilities and since side effects are relatively rare, the output of functions tends to be predictable, leading to lower complexity in many cases, at the cost of flexibility. SML has no built in support for automated testing and there are few testing frameworks available (see section 1.2).

More than 90 % of today's websites use JS [13] and its applications have become increasingly complex [14, question 23]. SML on the other hand, although not nearly as widespread, is used in critical applications. 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.

SML is a functional language just like JS, but is not implemented in browsers and has a static type system, lack of (prototype-based) object orientation support, and limited support for mutation. Because of this, SML is typically used in education, back-end logic and algorithm design than for web application front-ends. SML is well suited for formal verification, which in theory is excellent, but practical aspects such as how difficult formal verification is to do, how much benefits there are for maintainability, modularity and code design, and the time and resources required to do it need to be considered. In which scenarios is formal verification feasible? How does it affect productivity and the ability to get quick and frequent feedback whether the program is still correct? Even though testing can seldom provide the same guarantees as formal verification regarding correctness, there are many scenarios in which it is more cost effective and makes life easier for the developer. The two can of course also be used in parallel. See section 7.2 for more on formal verification.

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

This paves the way for test-driven development, which brings benefits in terms of the design becoming more refined, and increased maintainability. To achieve these gains from testing, the tests themselves need to be of high quality: they should be maintainable, fast and test the right thing. How to achieve this is an important part of this thesis, which is discussed in section 8.1.

1.2 (ML1,JS1)Background to Project

The first known JS testing framework JsUnit[1] was created in 2001 by Edward Hieatt [15] and since then several other test framework has appeared such as the testing framework for jQuery: QUnit[2], and JsUnits sequel Jasmine[3]. There are also tools for mocking such as Sinon.JS[4] (see section 4.6). It seems as if the knowledge of how to get started smoothly, how to make the tests stable and time efficient, and what to test, is rare. Setting up the structure needed to write tests is a threshold that most JS programmers do not overcome [7] and thus, they lose the benefits, both short and long term, otherwise provided by testing.

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

There is only one production quality testing framework available for SML, namely QCheck[5]. A few other frameworks exist but have not gained any traction and are relatively small, typically less than a year old and not under active development. The recent increase in the number of testing frameworks could be a consequence of developers being more willing to share their work as open source, or an indication that testing has been increasingly popular over the last one or two years, just as in the JS community [2, question 1]. An exhaustive list of the other SML testing frameworks and a short discussion about their differences can be found in the Appendix.

Material on how to test SML properly is hard to come by. Similarly, in guides on how to use different JS testing frameworks, examples are often decoupled from the typical use of JS – the Web – which is a problem [14, question 3] although it has become better compared to a couple of years ago [10, question 27]. Examples tend to illustrate merely testing of functions without side effects and dependencies. Under these circumstances, the testing is trivial and most JS programmers would certainly be able to put up a test environment for such simple code.

Examples are useful when learning idiomatic ways of solving problems. Code tends to end up being more complicated than examples you read because it is hard to come up with useful abstractions that make sense. Those who write examples to illustrate a concept always have to find a tradeoff between simplicity, generality and usefulness, and tend to go for simplicity [2, questions 56-57]. This can for example be observed in [9, p. 13-45] where the tests and their setup are omitted despite their claimed importance. Combining different concepts may help to achieve code with good separation, that can be tested by simple tests.

In contrast to examples often being simple and seldom providing a full picture, the problem domain of this thesis is to focus on how to test the behaviour of JS that manipulates The Document Object Model (DOM) elements, fetches data using asynchronous calls, validates forms, communicates through APIs or manipulates the appearance of a web page (see section 6). The domain also includes testing of SML in general.

1.3 (ML1,JS1)Scope and Delimitations

The scope of this thesis is mainly limited to *automated* testing of SML and *client side* JS. This means that the impact of frameworks specialised for server side JS code such as vows[6] and cucumis[7] has not been considered. Testing client side code is by no means more important than server side, but in many aspects it is different and sometimes harder. Reasons for choosing JS and SML over other programming languages have already been covered in the introduction.

JS testing frameworks that are no longer maintained such as JsUnit[8] and JSpec[9] was also deliberately left out of consideration. Others were left out because of a smaller user

[5] `https://github.com/league/qcheck`
[6] `http://vowsjs.org/`
[7] `https://github.com/noblesamurai/cucumis`
[8] `https://github.com/pivotal/jsunit`
[9] `https://github.com/liblime/jspec`

base or lack of unique functionality; among these we find TestSwarm, YUI Yeti and RhinoUnit and the majority of the SML testing frameworks (see Appendix). These are useful tools but could not be included due to time limitations.

Manual testing was not covered to any significant extent, since there were enough issues to deal with in automated testing. Naturally, there are many situations where manual testing is required, but in this thesis *testing* typically refers to automated testing.

Since SML has a smaller user base than JS, the majority of the research in this thesis has been focused on JS. Researching today's limited testing of JS may be done from different perspectives. There are soft aspects such as:

– Differences in attitudes towards testing between different communities and professional groups and knowledge about testing among JS developers (section 8)

– How JS is typically conceived as a language and how it is used (section 3.3.2)

– Economic viability and risk awareness (section 8.4)

There are also more technical aspects:

– Testability of JS code written without tests in mind (section 5)

– Usability of testing tools and frameworks (sections 8.5, 5.5 and 5.5)

– 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.

An important part of the scope has been to account for how to conveniently proceed with JS and SML testing. The ambition was to cover not only the simplest cases but also the most common and the hardest ones, and introduce available tools and frameworks. Many tutorials for 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 had to be paid to the varying needs of different kinds of applications. It was also important to describe how to write tests that are as maintainable as the system under test, to minimise maintenance costs and maximise gain.

Rather than proposing best practices for JS 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.

A full evaluation of the most popular MVC and testing frameworks is not within the scope of this thesis, but others have done it [8][16]. Popular JS testing frameworks include assertion frameworks such as Jasmine, qUnit, expect.js and chai, and drivers/test runners such as Mocha, JsTestDriver, Karma and Chutzpah[10] which may have their own assertion framework built in but are typically easy to integrate with other assertion frameworks using adapters or via built-in support.

2 (ML1,JS1)Previous work

Today blog posts and books about patterns in JS 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 [17][18]. For examples of JS testing specifically the alternatives have been scarce until recently [6][19][20][21].

There is a lot of previous work on testing patterns and how to write concise, useful and maintainable tests [18, part III][19, ch. 3-5][6, p. 461-474][20, p. 86-87][21, p. 13-14]. In todays BDD frameworks there is often a possibility to separate the basic functionality from the special cases by organising the spec (specification)s in nested describes. This can provide an overview of what an application does just by looking at the spec output and is commonly seen in open source projects [22, question 42].

In 2010, a student at the Swedish royal institute of technology called Jens Neubeck wrote a master thesis about test-driven JS application development [23]. In his thesis, he evaluated Crosscheck, HtmlUnit och Selenium, with the conclusion that none of them were mature enough to be used for the considered applications. Today, HtmlUnit and Selenium have evolved and there are new tools available such as PhantomJS, Buster.JS and Karma, so the conclusion might not hold anymore. The selection of frameworks could have been done more carefully, as useful tools such as JsTestDriver and Jasmine were not considered, despite being available at that time. Neubeck's results are based purely on his own experiments and he had no JS specific source to base his claims about JS testing on.

2.1 (ML1,JS1)Books

The main source of reference within the field of JS testing today is *Test-Driven JavaScript Development* [6] by Christian Johansen which deals with JS testing from a Test-Driven Development (TDD) perspective. Johansen is the creator of Sinon.JS[11] and a contributor to a number of testing frameworks hosted in the open source community. The book takes a rather practical approach to JS testing by explaining many aspects of how JS works and by including practise exercises. It is not very scientific but makes up for this with its pragmatism and roots in the software industry.

Lately, a large number of books about JS testing has appeared. *JavaScript Testing with Jasmine* [21] covers the Jasmine testing framework in detail, *Behaviour Driven Development with JavaScript* [19] presents a BDD perspective of JS testing and *Testable JavaScript* [20] looks at testability aspects of JS. All three of these were published this year (2013), and Johansen's book was published only a few years ago in 2010.

[10] `http://chutzpah.codeplex.com/`
[11] `http://sinonjs.org/`

Except for discussions on how to perform mathematical proofs and equality tests of polymorphic types, there are no books that cover testing of SML. The main sources of reference for SML are *The Definition of Standard ML* [5] which covers the language syntax and semantics, *The Standard ML Basis Manual* [24] which describes the standard library, *Elements of ML Programming* [25] which cover the features of SML in a more comprehensible fashion, *ML for the Working Programmer* [26] which is somewhat outdated but comprehensive, and various lecture notes [27][28][29][30]. None of these describe ways of doing automated testing and there seems to be an attitude against testing based on that it can not prove absence of errors in the way formal verification can [28, p. 16].

2.2 (ML1,JS1)Articles

There are many academic papers about testing web applications available, and quite a few of them focus on JS specifically (whereas practically none involve SML). Papers on testing of SML are hard to find even outside of the web applications context, most that are of any interest are concerned with formal verification or related to the QuickCheck tool which can be used to generate tests for Haskell code, using SML as an intermediate language.

This idea of generating tests has also been applied to SML and JS. *A Framework for Automated Testing of JavaScript Web Applications* [11] focus on automatically generating tests to achieve a high degree of code coverage. The problem with this is that the ability to employ test driven development is generally more valuable than high code coverage, due to its effect on the system design. Most likely, automatically generated tests tend to be harder to maintain and cause false positives as the code changes after the tests have been generated.

QuickCheck and its SML counterpart QCheck are based on another principle than the framework presented in the paper just mentioned, they are specified in terms of properties that should hold rather than generated from the code itself or based on user interaction data. While avoiding the circularity of generating tests based on the code that should be tested, this approach instead suffers from the difficulty of identifying and expressing properties that should hold, and there is uncertainty in how well the properties are actually tested.

Automated Acceptance Testing of JavaScript Web Applications [31] describes a way to specify intended behaviour of a web application and use a web crawler to verify the expectations. It appears as a promising alternative to using Cucumber in conjunction with Selenium (see section 4.7), but more case studies are needed in order to evaluate its usefulness, applicability and scalability.

Sebastien Salva and Patrice Laurencot has described how STS automata can be applied to describe asynchronous JS applications and generate test cases [32].

Heidegger et al. cover unit testing of JS that manipulates the DOM of a web page using techniques from software transactional memory (STM) to restore test fixtures [33].

Ocariza et al. have investigated frequency of bugs in live web pages and applications [34]. These are both aimed at testing client side JS that runs as part of web sites.

Phillip Heidegger and Peter Thiemann has addressed the issue of type related errors in JS by introducing JSConTest, a contract framework that enables guided random testing by specifying types and relations of the arguments and return value of functions [35].

3 (ML,JS)Methods

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

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

In order to describe ways of writing tests for JS, the practical work involved testing an existing application, performing TDD exercises from Test-driven JavaScript Development [6] and doing some small TDD projects during the framework evaluation. There were plans to have a workshop field study, where programmers would work in pairs to solve pre-defined problems using TDD, but in the end it was decided that it would be too difficult to extract useful data from such an activity.

3.1 (ML,JS)Literature Study

Relevant books, articles and Internet sources were identified and skimmed, some were read through thoroughly. As was mentioned in (ML1,JS1)Previous work (section 2), several new titles were published during the work of this thesis. Additionally, some older literature was also included as the need became apparent.

3.2 (ML,JS)Programming

Much time was spent programming, including working through the projects of [6, part III], adding tests to an existing JS application (see section 5.1), developing an SML testing framework (see section 7.3) and using the developed framework in practise to solve programming assignments in a MOOC (Massive Open Online Course), namely the Coursera Programming Languages course[12].

A thorough evaluation of frameworks for JS and SML was not part of the scope, but since they pose a significant part of how to solve problems with testing many were involved anyway. The following JS testing frameworks have been part of the practical work:

[12] https://www.coursera.org/course/proglang

Jasmine[13], qUnit[14], Karma[15], Mocha[16], JsTestDriver[17], Buster.JS[18] and Sinon.JS[19]. Apart from the MOOC programming assignments, all code is publicly available on my Github account *emilwall*, together with the LaTeX code for this report.

3.3 (ML0,JS)Interviews

Semi-structured case study interviews were used rather than surveys to gather individual views on the subject. This approach allowed for harnessing unique as well as common experiences which would not be picked up in a standardised survey. The interviews were between 20 and 60 minutes long and were conducted both in person and via Internet video calls.

3.3.1 (ML0,JS1)Interview Considerations

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

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

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

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

[13] `http://pivotal.github.com/jasmine/`
[14] `http://qunitjs.com/`
[15] `http://karma-runner.github.io/`
[16] `http://visionmedia.github.com/mocha/`
[17] `https://code.google.com/p/js-test-driver/`
[18] `http://docs.busterjs.org/`
[19] `http://sinonjs.org/`

3.3.2 (ML0,JS1)The Interviewees

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

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

The third person that was interviewed was Marcus Ahnve, a senior developer and agile coach that has been in business since 1996 working for IBM, Sun Microsystems and ThoughtWorks, and as CTO for Lecando and WeMind. He is currently working at Valtech. He is an experienced speaker and a founder of Agile Sweden, an annual conference since 2008. He is also experienced with test driven development in Java, Ruby and JS.

The next interview was with Per Rovegård, a developer with a Ph.D. in Software Engineering from Blekinge Institute of Technology. He has worked for Ericsson and is currently a consultant at factor10 where he has spent the last year developing an AngularJS application, with over 3000 tests. He is the author of the programatically speaking blog and have done several talks at conferences and meet-ups, most recently about Angular and TDD at sthlm.js on the 2nd of Oct 2013 but the interviews took place in Aug over Skype.

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

3.3.3 (ML,JS)Other People Involved

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

4 (ML,JS)Technical Background

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

4.1 (ML0,JS0)Basics of Testing

Every developer performs testing in one way or another. Running an application, interacting with it and observing the results is one form of testing. The more time spent on developing the application, the more evident the need for automated tests tends to become, to reduce the amount of manual work and time spent repeating the same procedures.

Automated testing is commonly divided into different categories of tests, that exercise the code in different ways and for different reasons. Unit testing and integration testing is perhaps the most common concepts. Unit testing focuses on testing units (parts) of an application, in isolation from the rest of the application, whereas integration testing is about testing that the units fit together. Sometimes there is an overlap between the concepts, where a unit is relatively large.

Commonly, unit tests are more low level and integration tests are more like user interactions. This does not have to be the case however, unit tests can be written in a business value focused fashion and integration tests may be written before there is even a GUI to interact with [10, question 20]. Thinking in terms of APIs, there is usually a possibility of defining what each part should do, so you can write tests in an outside-in fashion, which reduces the risk of developing something that eventually turn out to be redundant or misdirected (remember, YAGNI (You Ain't Gonna Need It)) [10, question 29]. Acceptance testing is often characterised by this property also, and may need to be put aside as pending tests, that does not have to be implemented any time soon [10, question 31]. So called pending tests can be useful to guide the development in a clear direction, but the number of pending tests should be kept to a minimum, or else they will become outdated. This can be seen as a consequence of such a test not being timely, thereby breaking the last component of the F.I.R.S.T. principle [36, p. 132-133].

There are potentially both good and bad consequences of testing, both from a short and from a long term perspective. A disadvantage is that setting up the test environment and writing the tests take time. If the testing process is not carried out properly, maintaining the tests can cause frustration. The advantages are that if time is spent thinking about and writing tests, the development of production code will require less effort. Testing provides shorter feedback loops, executable documentation and new ways of communicating requirements with customers. The quality and maintainability of the end result is likely to be positively affected and making changes becomes easier, so ideally, the pace of development does not stagnate. The extra time required to set up the test environment and write the actual tests may or may not turn out to pay off, depending on how the application will be used and maintained.

4.2 (ML0,JS0)Test-Driven Development

Test-Driven Development (TDD) is ...

TDD shifts focus from implementations to testing, thereby enforcing a thought process of how to translate requirements into tests. When using TDD, the most common reason for a bug is because the TDDer has moved on prematurely, i.e. not writing enough tests to find a scenario in which the code does not behave as it should or not fully understood the requirements. These kinds of bug would probably persist regardless of if TDD is used or not, but the thing to be careful about here is that there is a risk of not putting as much energy into writing flawless production code when TDDing, instead you rely on iterative improvement and refactoring.

A common principle of TDD, especially when carried out in a CI build, is that the application should frequently be brought to a state where all tests pass [52, question 2]. An exception from this rule is pending tests, but as mentioned in section 4.1, these should be kept to a minimum.

4.3 (ML0,JS0)Behaviour-Driven Development

Behaviour-Driven Development (BDD) is about describing expected behaviour of systems and writing tests from the outside in. It replaces and adds terminology traditionally used within TDD and encourages descriptive naming of tests (or specs) that helps readability and to make failure messages more helpful. [10, questions 17-18]

An advantage with BDD is how it encapsulates ideas about how tests can be organised, for intance through the Given, When, Then (GWT) form (covered in greater detail in section 8.5). Providing a context for specs can help to avoid having a single hard-to-read setup for all tests of a class, which is otherwise common in classical TDD and testing in general. Having a single setup can be problematic not only for readability reasons that can make them harder to understand, but also because it creates dependencies between tests. A common alternative to the GWT form is nested describe clauses with it-specs as in Jasmine and BDD style Mocha. This requires more discipline from the developer because the context has to be covered by the describe text. The classical BDD framework RSpec has a `when` directive that serves as a compromise between the two styles.[10, question 19]

Strictly speaking, you do not need a BDD framework to perform BDD. J. B. Rainsberger, the author of JUnit Recipes: Practical Methods for Programmer Testing, explained how to do BDD in jUnit at the XP 2011 conference in Madrid. The key to doing this is to divide the tests based on system behaviour rather than classes. This is the same concept as when writing specs in Cucumber, another Ruby GWT framework, and the same principle applies to BDD in JS (note that Cucumber can also be used to generate acceptance tests for JS). This is desirable because it helps you to not prioritise implementation details in favour of the parts that truly matters from a business value point of view.[10, question 20]

4.4 (JS0,ML0)Spikes in TDD

Although writing tests first is a recommended approach in most situations, there is a technique for when you need to try something out before you write tests for it, without compromising testability. Dan North, the originator of BDD, came up with a name for this technique: spiking [37]. The idea is that you create a new branch in your version control repository, and hack away. Add anything you think might solve your problem, don't care about maintainability, testability or anything of the sort. If you find yourself not knowing how to proceed, discard all changes in the branch and start over. As soon as you get an idea about how a solution could look like you switch back to the previous branch and start coding in a test first fashion. [2, 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 stabilising (refactoring and adding tests) once sufficient feedback has been obtained to reduce the uncertainty that led to the need for spiking [38]. She argues that this allows her to get faster feedback and be more agile without compromising the end result in any noticeable way. In another post, she emphasises that this approach is only suitable for developers who are really good at TDD, while at the same time claiming that it is more important to be "able to tidy up the code" than "getting it right in the first place" [39]. It may seem like an elitist point of view and a sacrilege towards TDD principles but in the end, whatever makes you most productive and produces the most valuable software has raison d'être.

Counterintuitive it may seem, throwing away a prototype and starting from scratch to test drive the same feature can improve efficiency in the long run. The hard part of coding is not typing, it is learning and problem solving. A spike should be short and incomplete, its main purpose is to help you get your mind set on what tests you can write and what the main points in a solution would be. [2, question 60]

A similar concept to Spike and Stabilise is to write markup without tests until a feature is needed that could use an API call. Then you write a test for it, and can start to work with that feature in a TDD fashion. [10, question 30] This way of thinking can lead to a useful rationale regarding when to test – add tests whenever you introduce an external dependency to make sure that dependency is called correctly under certain circumstances, then if that dependency is something that already exists you can just add it, otherwise you can develop it in a test-driven fashion.

4.5 (ML0,JS1)Lint Tools for JavaScript

Linting is a static analysis tool for detecting syntax errors, risky programming styles and failure to comply to coding conventions. There are lint tools available for JS such as JS-Lint, JSHint, JavaScript Lint, JSure, the Closure compiler and PHP CodeSniffer. JSLint does provide some help to avoid common programming mistakes, but does not perform flow analysis [40] and type checking as a fully featured compiler would do, rendering proper testing routines the appropriate measure against programming mistakes.

4.6 (ML,JS1)Mocking and Stubbing

Mocking and stubbing involves simulation of behaviour of real objects in order to isolate the system under test from external dependencies. This is typically done in order to improve error localisation and execution time, and to avoid unwanted side-effects and dependencies such as communication with databases, across networks or with a file system [41, ch. 2]. A mock is different from a stub in that it has pre-programmed expectations and built-in behaviour verification [6, p. 453].

In JS, 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 [2, question 53]. It might come across as manual work that you can avoid by using a stubbing tool, but the benefits include fewer dependencies and sometimes more readable code, as mentioned in section 10 [2, questions 54-55]. However, bear in mind that since JS has no notion of interfaces, it is easy to accidentally use the wrong method name or argument order when stubbing a function [6, p. 471].

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 overly complex stubbing needs can be a symptom for that the code is in need of refactoring [22, question 34], and strive for consistency by using a single method for stubbing – mixing VanillaJS with Jasmine spies and Sinon.JS stubs will make the tests harder to understand.

4.7 (ML0,JS)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. [42]

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.

4.8 (ML0,JS0)Refactoring

Refactoring is "a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behaviour", or the act of applying such changes [9, p. 46]. As mentioned in the (ML1,JS1)Motivation section (section 1.1), refactoring prevents program decay and helps to maintain productivity. Just like testing, refactoring is no magic bullet that solves all problems and it is important to know when to do it and when it might not be worth the effort. Common rules of thumb are to refactor when doing something similar for the third time (a pragmatic approach to the DRY (Don't Repeat Yourself) principle), when it helps to understand a piece of code, when it facilitates addition of a new feature, when searching for a bug and when discussing the code with others, e.g. in code reviews [9, p. 49-51].

Using refactoring tools can be very helpful to see which parts of the code is in most need of refactoring, and to automate certain refactoring actions by facilitating renaming, method extraction, etc. [41, ch. 5]. There is some support in IDEs such as Visual Studio (using JSLint.VS[20], ReSharper[21] and/or CodeRush[22]), WebStorm IDE[23] (or IntelliJ Idea[24] using a plugin) and NetBeans[25]. There are also standalone statistical tools such as kratko.js[26] and jsmeter[27] (not to be confused with the Microsoft Research project[28]) that helps you to identify which objects have too many methods and which methods do too many things (or have too many arguments). Relying on metrics such as lines of code is of course not always appropriate due to different coding styles, but at least it provides an overall picture [43].

There is a serious risk involved in refactoring untested code[9, p. 17], since manually checking that the refactoring does not introduce bugs is time consuming and difficult to do well. However, leaving the code untested means even greater risk of bugs and the refactoring may be necessary in the future anyway, in which case it will be even harder and more error-prone. This problem can be avoided by writing tests first.

[20] http://jslint.codeplex.com/
[21] http://www.jetbrains.com/resharper/
[22] http://www.devexpress.com/Products/CodeRush/
[23] http://www.jetbrains.com/webstorm/features
[24] http://www.jetbrains.com/editors/javascript_editor.jsp?ide=idea
[25] https://netbeans.org/kb/docs/ide/javascript-editor.html
[26] https://github.com/kangax/kratko.js
[27] https://code.google.com/p/jsmeter/
[28] http://research.microsoft.com/en-us/projects/jsmeter/

If refactoring is required in order to make the code testable, the architecture likely needs to change. Sometimes introducing a new object or changing how some dependency is handled is all that is needed. The important thing is to not continue in a direction that will cause the codebase to deteriorate over time. [22, question 34] If a lot of work goes into mocking dependencies then the tests take longer to write, require more maintenance and the need for integration tests increases. The proper remedy is usually not to reduce the isolation of the unit tests but to refactor the architecture so that each unit becomes easier to isolate. [22, question 42]

4.9 (ML,JS1)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 JS as well, Grunt[29] 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. [2, question 52] Jake and Mimosa are other well known and maintained alternatives. It is also possible to use Rake, Ant or similar. Just as JsTestDriver and Mocha have adapters for Karma and Jasmine (see section 8.5), Rake has evergreen[30] that allows it to run Jasmine unit tests. [44][10, question 6]

4.10 (ML,JS)Setting up a test suite

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

Build programs (see section 4.9) and version control systems play an important role in automating testing.

5 (ML,JS)Testability – Real world experiences

Adding tests for code that was written without testing in mind is challenging [6, p. 18]. In this section an effort to do so in JS is described, in order to highlight the problems and

[29] `http://gruntjs.com/`
[30] `https://github.com/jnicklas/evergreen`

solutions to some of them. Writing tests first can also be challenging, but for completely different reasons. For this reason, a TDD approach to solving programming exercises in SML in a MOOC is also described.

5.1 (ML0,JS)Description of the asteroids example

The first step in the case study of adding tests in retrospect was selecting a suitable application. The choice became an HTML5 canvas game, namely asteroids (see figure 2), based on that it combined graphical elements with relatively complex logic and a bit of jQuery, making it a reasonable representative for the typical JavaScript application. Most of the code was reasonably well modularised already from the start and it was easy to get started by simply cloning the repository and opening the supplied HTML file with the canvas element and script locations already defined.

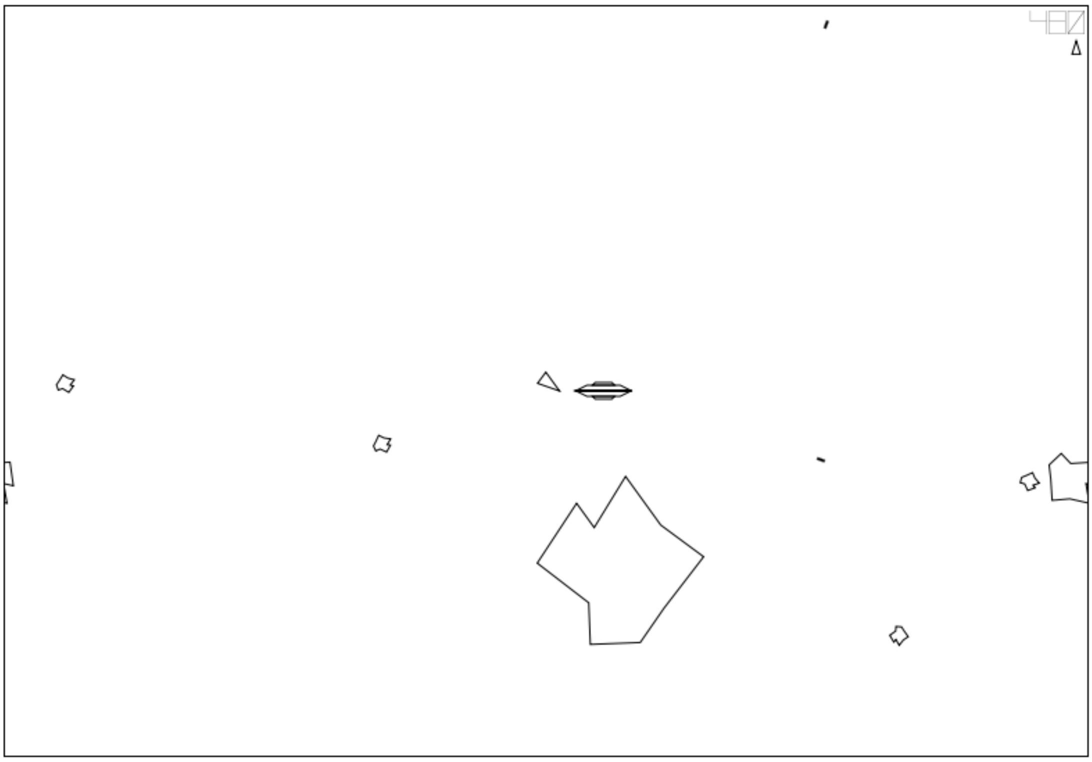


Figure 2: The asteroids HTML5 canvas game: here the player ship (left) is about to collide with an enemy ship (right) which will destroy them both (unlike what happens when asteroids collide, an example of desireable behaviour to test for)

The first action was to strip away any redundant files such as iPad specific code and sound files that were not used anyway. This led to a project structure of only game.js, index.html, the licence file, jquery 1.4.1 and a JS file containing the fonts (used to display score and messages in the game, see figures 5, 4 and 7). The next step was to split game.js into separate files in order to test each "class" in isolation. This could have been done the other way around, guiding the structure by writing tests first, but it felt natural to work with the source in order to understand it. Refactoring without tests is typically not a good idea[], but it turned out as reasonably easy to change the code even without tests as long as care was taken to preserve the order in which the code was executed when moving things to separate files. Since the exercises in Test-Driven JavaScript Development[6] had introduced JsTestDriver, using that with a Jasmine plugin felt like a natural choice at the time, although it involved some issues with syncing the tests in the browser and find the correct versions for the Jasmine adapter to work. The project structure is displayed in figure 3.

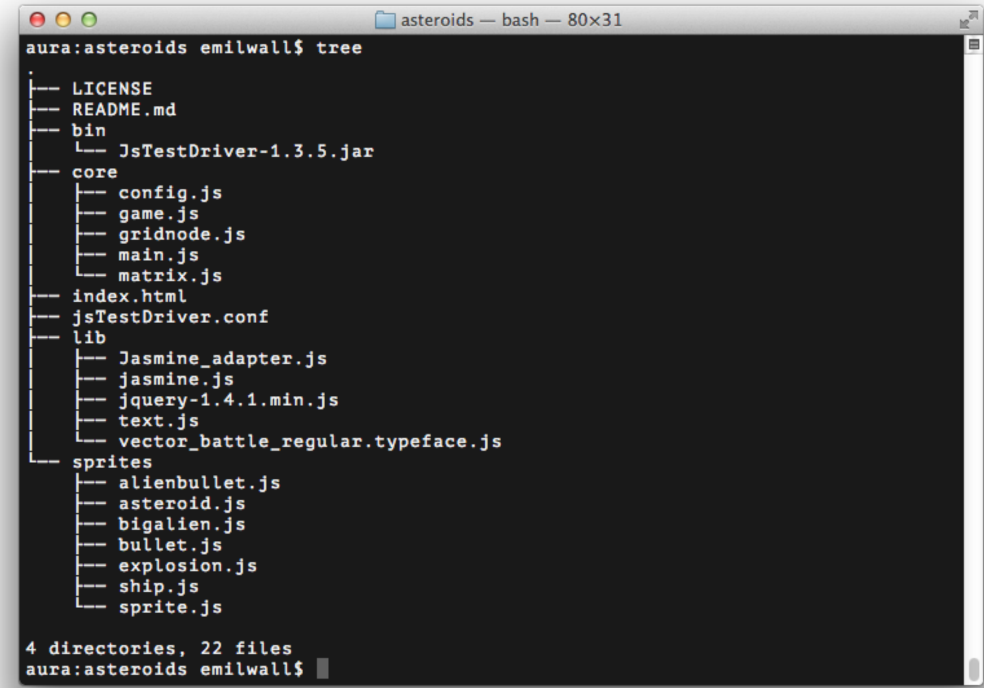


Figure 3: Project structure after splitting `game.js` into several files and adding JSTD and Jasmine, but before adding any specs

Later on, tests were added in new directories `sprites-spec` and `sprites-spec`, Sinon.js was added to the `lib` directory for better stubbing, rendering.js was extracted from main.js to enable more selective execution and a reset.js file was added to the `core` directory that ensured that any old definitions in the asteroids namespace was overwritten before reading the files anew. This last change, reset.js, should not really be necessary, but was done in order to not having to empty the cache in between each test run with JsTestDriver. For a more comprehensive picture of what all of this means, please see [31].

[31] `https://github.com/emilwall/HTML5-Asteroids`

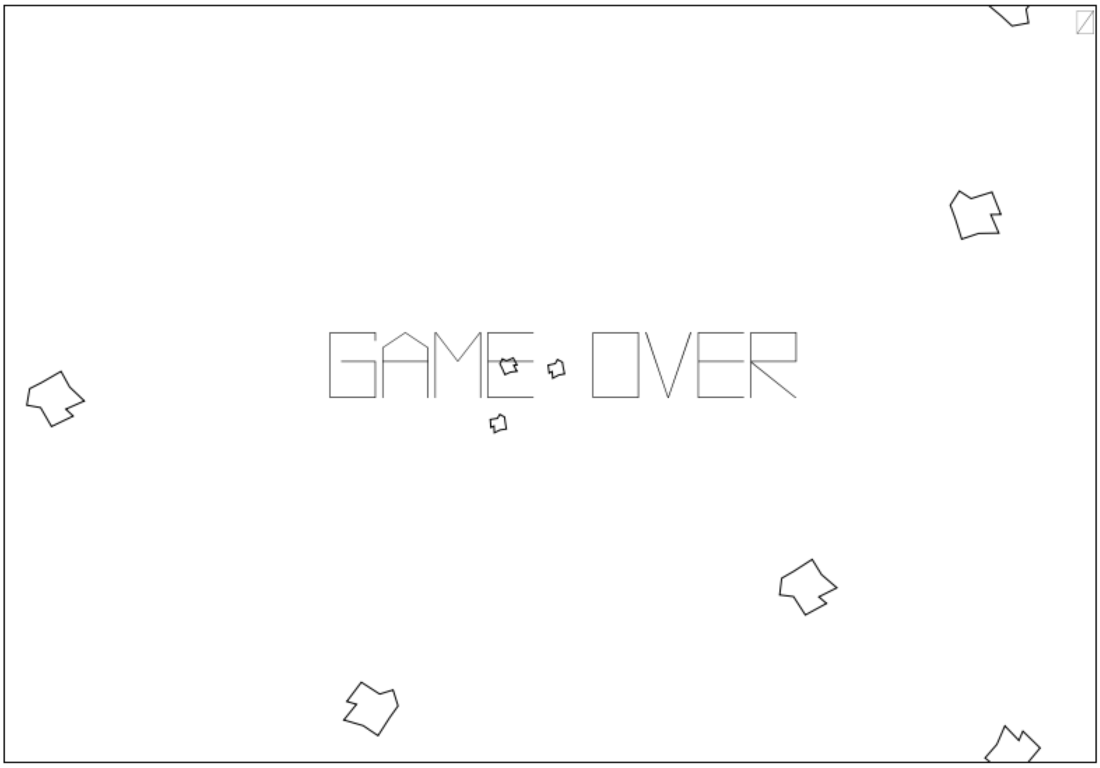


Figure 4: The game over screen, displayed for one second when all lives are lost

5.2 (ML0,JS)Attempts, observations

Once starting to write tests, the first problems were that some of the code was contained in a jQuery context and that the canvas element was not available in the unit testing environment, i.e. the code was executed in the inaccessible body of `$(function ()...`, with the selector `$("#canvas")`. Efforts to load the contained code using ajax were of no gain, so the solution instead came to involve exposing the contents of the context as a separate class (rendering.js), and manually inject the canvas and other dependencies into that class. This required turning local variables such as the 2d-context into attributes of the class, to make them accessible from the original location of the code (the jQuery context). This change introduced a bug that manifested itself only when a feature to draw grid boundaries (see figure 6) was activated. The feature was not important and could have been removed, but solutions were considered anyway for investigation purposes. The bug could be fixed by making a couple of variables in the rendering class globally accessible as attributes, but that would break certain abstractions, possibly making the code harder to maintain. A better solution, which was also eventually chosen, was to move the code for drawing grid boundaries into the rendering class, where it really belonged.

The event handling for key-presses could not be extracted since they relied on being executed in the jQuery context for the events to be registered correctly. The event

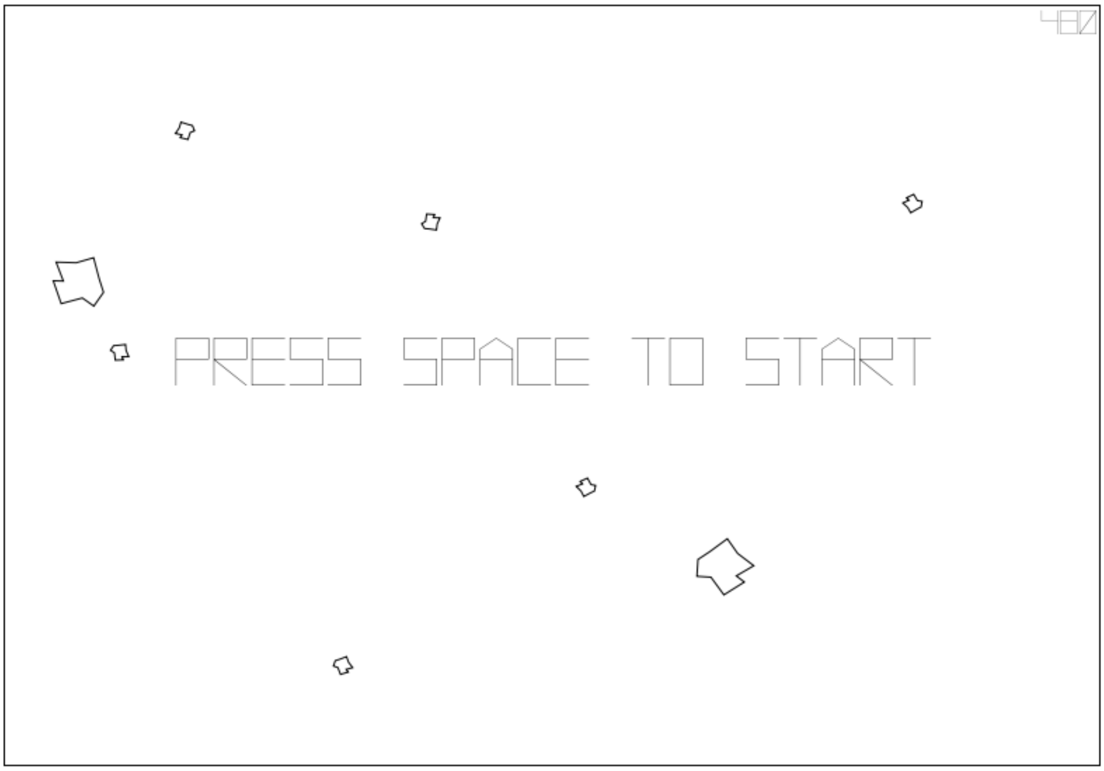


Figure 5: Text displayed after game over, enabling the user to see the score and start a new game

handles had dependencies on local variables in main.js so in order to extract the code into a separate class these local variables would have had to be made global, which would undermine the design by introducing even more global state. Based on this, the event handling code was left without unit tests, leaving it to integration and system testing to detect possible defects such as improper changes to global variables that were used in the event handler.

The major problems however were not of this technical sort, directly related to JS, but rather had to do with basic testing principles and how the tests were written. The original purpose of the testing was to provide a safety net for further improvement of the game, but it ended up as over-specification because many tests became implementation specific rather than based on what the code should achieve. Other tests, that were more behaviour and feature oriented, instead had issues with insufficient stubbing and cleanup, causing dependencies between tests so that they would fail or pass for the wrong reasons. The insufficient stubbing and cleanup was largely rectified by executing each set of specs in isolation with just the unit under test, without loading any dependencies before stubbing them, and ensuring that any side effects from running the specs affected only a certain namespace, which could then be reset between each test execution. The problem with the test locking the implementation through over-specification remained however.

A notable testability problem was to automate end-to-end testing. The game involved

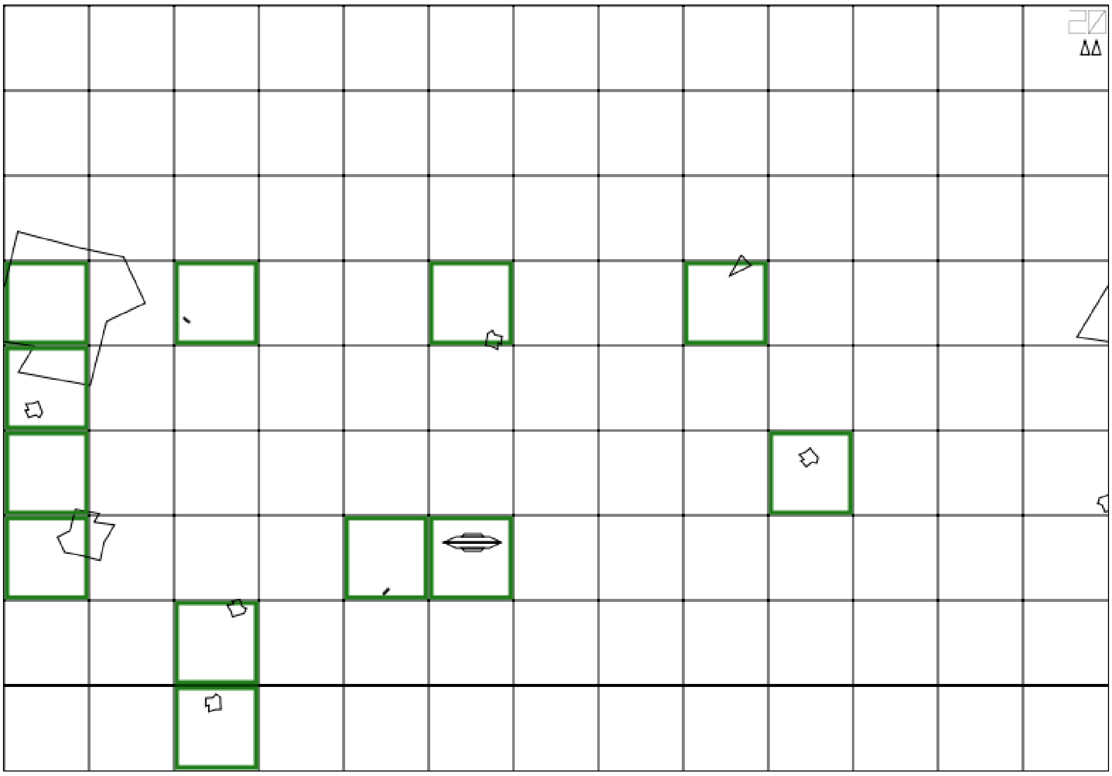


Figure 6: Draw grid function in action, highlighting cells with sprites

many nondeterministic aspects in how the asteroids moved that would be awkward to control in the tests, so it was decided not to attempt this kind of testing. If it had been attempted, there would probably have been technical challenges in controlling the game, and areas that would have required careful consideration such as what to base the assertions on in the tests. One tool that looked promising was js-imagediff[32], which includes a `toImageDiffEqual` Jasmine matcher for comparing two images as well as utility methods for producing images of an application that uses canvas. It could probably have been used to avoid having to manually construct fake objects for the canvas in unit tests as well. A similar useful module is Resemble.js[33], that has advanced image comparison features that are useful together with PhantomCSS[34].

5.3 (ML,JS)Conclusions

If the code has to be modified in a way that severely breaks the original design and exposes variables that should not be used from any other part of the application in order to make it testable, it can basically mean one of two things: either you should not test the code in the way you are trying to do, or you should rethink the responsibilities of that code. Perhaps it is doing more than one thing, and it is the secondary thing

[32] `https://github.com/HumbleSoftware/js-imagediff`
[33] `https://github.com/Huddle/Resemble.js`
[34] `https://github.com/Huddle/PhantomCSS`

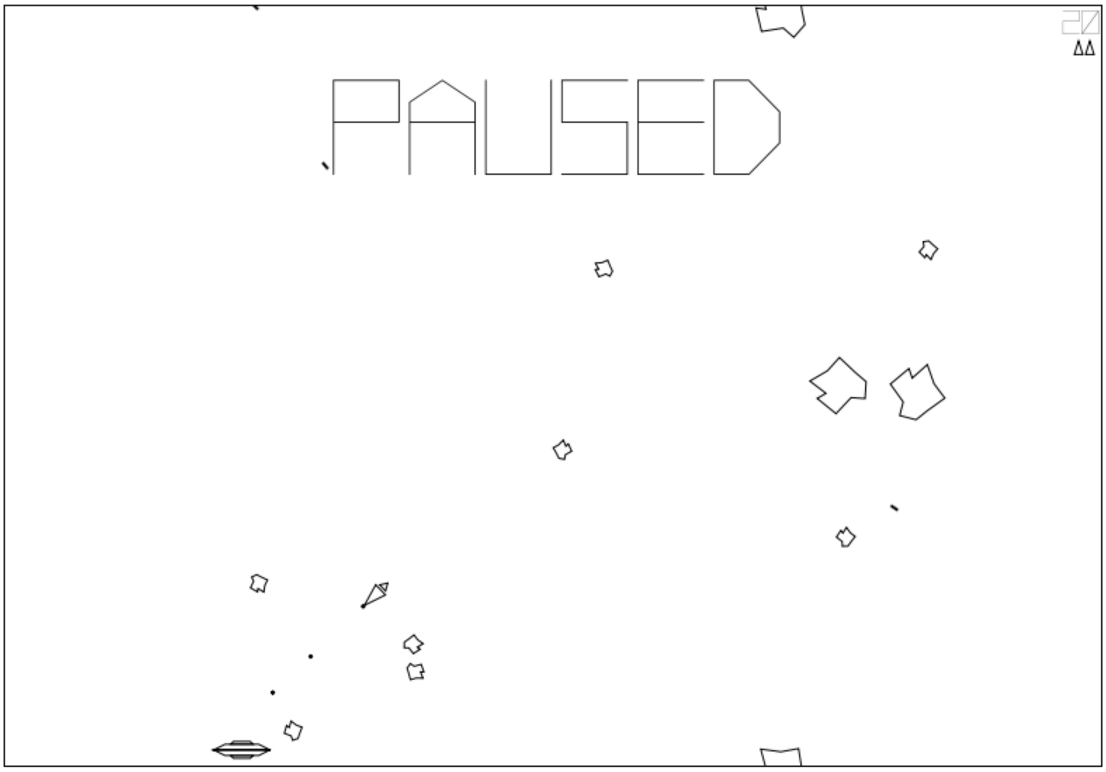


Figure 7: Game paused while accelerating and shooting towards an enemy ship

that you feel that you want to test. No wonder then that it feels wrong to expose it, it is not the primary purpose of that code segment, just an implementation detail. The key-press event handlers mentioned earlier is one example of this – the mechanism of event handling should probably have been kept separate from the settings and controls of the application.

The object orientation of the asteroids application provided useful structure and was a rather good fit since it really did contain natural abstractions in the form of objects. An interesting thought is whether the code would have been more testable if it had been written in a functional style. Certainly, there would have been less outer dependencies to locate and isolate in the tests, or less work to instantiate the objects. [10, question 26]

tools and testability

5.4 (ML0,JS)Testability issues with main.js in asteroids application

As mentioned in section 4.7, Selenium is currently the most popular tool for browser automation. Sometimes it is the only way to test a JS of an application without rewriting the code[22, question 43] (as mentioned in section5.9) but then tests tend to be brittle and provide little value. In general, since Selenium tests take such time to run they should only cover the most basic functionality in a smoke test fashion. [22, questions 16-17][10, question 21] 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. [2, question 44]

With Selenium IDE (again, see section 4.7), 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. [2, questions 42-43]

As a consequence of being unable to extract all code from main.js into testable classes, selenium tests were indeed considered. 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 JS 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 similar 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 optimisations 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.

5.5 (ML0,JS1)JsTestDriver evaluation

When setting up JsTestDriver[35] (JSTD) with the Jasmine adapter there are pitfalls in which version you're using. At the time of writing, the latest version of the Jasmine JSTD adapter (1.1) is not compatible with the latest version of Jasmine (1.3.1), so in order to use it you need to find an older version of Jasmine (such as 1.0.1 or 1.1.0) or figure out how to modify the adapter to make it compatible. Moreover, the latest version of JSTD (1.3.5) does not support relative paths to parent folders when referencing script files in jsTestDriver.conf although a few older versions do (such as 1.3.3d), which is a problem if you want to place the test driver separate from the system under test rather than in a parent folder, or if you want to reference another framework such as Jasmine if it is placed in another directory.

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

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 reproducible 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 [2].

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 [45].

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 JS, 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 JS) 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 optimisation.

Deficiencies such as these are important to note because they pose potential reasons to why JS 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. Realising 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.

5.6 (ML0,JS1)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 analysed with respect to what was considered to be its expected behaviour, such as adding something to a data structure or performing a call with a certain argument, and then a short sentence described that behaviour 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 behaviour 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 organisations (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 formalising this, as described in Introduction to Software Testing by Ammann and Offutt [46].

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[46, 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.

5.7 (ML,JS)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.

5.8 (ML,JS)Deciding when and what to test

Testing is especially important for large applications. It is extra valuable when new functionality is added because it helps to verify that old functionality is not broken and that the new code is structured appropriately. [22, questions 6-7]

You should focus on testing behaviour rather than appearance and implementation details. [2, question 10] Rather than testing that a certain class inherits from another, test that the methods of that class behaves as one would expect. Whether or not that is dependent on the inheritance patterns is mainly relevant for stubbing considerations - you may want to replace the prototype of an object in tests so that you can check that there are no unexpected dependencies. These are lessons learnt from working with the asteroids application, see section 5.4.

Another thing that came up during the interview with Edelstam was that when something feels like it is hard to test, it is likely that any test you write will become rather brittle as the code changes in the future. The proposed solution was to avoid testing it unless the code can be refactored so that testing becomes easier. [2, question 30] When writing the tests for the asteroids application, I deliberately chose to write tests even when it felt hard or felt like it provided little value, to see whether this made the application harder to test later and if people would remove the bad tests during the workshop.

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

Naturally, the more important a certain functionality is in relation to business value, the more effort should be put into e2e and similar tests related to it. Inherently complex parts of the code and code that is likely to change should be tested by unit tests to allow for refactoring and increase chance of discovering bugs, whereas simple code that is not expected to change can be tested manually. However, it is typically hard to know beforehand if the code you are writing is subject to change, so a compromise by writing a few simple tests for that code may pay off. [22, questions 28-29 and 33]

In the end, when deciding what tests to write, one of the most important things are to make sure that the tests are meaningful. They should in some way be connected to the problem that the system solves and describe something that matters rather than just testing methods for the sake of it. [10, questions 17-18]

5.9 (ML0,JS1)Meet-up open space discussion

During a talk at a meet-up 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 JS. 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 JS 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 JS 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 JS is good if done right, but also difficult.

5.10 (ML,JS1)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 JS 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 JS probably is so uncommon because people programming in JS often have a background as web graphic designers, without that much experience of automated testing. Another common conception is that JS 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.

6 (JS,ML0)Problems in JavaScript testing

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

In order to get started with testing JS, you need to understand why you should do it [2, question 38] and then practise with katas and begin to use testing in your projects. Coding katas focused on front-end JS development are not very common, but not impossible to do [10, question 47].

6.1 (JS,ML0)Asynchronous events

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

Two basic approaches to testing asynchronous code are to force the execution into a certain path by faking the asynchrony, and to set up waiting conditions for when assertions should be executed. When forcing the execution path, the code is not executed asynchronously but you can test different scenarios and thereby get a good feel of how the code will behave when run asynchronously. When using waiting conditions, timeouts need to be set for when tests should fail if waiting conditions are not met.

Both Mocha and Jasmine supports waiting conditions, with some differences in syntax and how it works.

A common concern is that the large number of possible combinations in which asynchronous code can be executed leads to a combinatorial explosion. This is true, but it is also true that if there are few dependencies between the asynchronous code and if they do not share resources, few things can go wrong.

Erlang message passing is also asynchronous, so how do they test their code? It has been used in system with high demands on stability, so there is probably a large culture for testing.

Blog posts (these should be in the bibliography and perhaps mentioned under previous work):

`http://lostechies.com/derickbailey/2012/08/18/jasmine-async-making-asynchronous-testing-w`

`http://lostechies.com/derickbailey/2012/08/17/asynchronous-unit-tests-with-mocha-promises`

`http://www.htmlgoodies.com/beyond/javascript/test-asynchronous-methods-using-the-jasmine-`

`http://www.htmlgoodies.com/beyond/javascript/how-mocha-makes-testing-asynchronous-javascr`

`http://blog.caplin.com/2012/01/17/testing-asynchronous-javascript-with-jasmine/`

6.2 (JS,ML0)DOM manipulation

A key to successful testing of JS with DOM manipulation is to separate the logic from the DOM manipulation as much as possible. One technique that was presented at the XP conference in Trondheim 2010[36] was to define a view layer of jQuery-related and similar code, and update the view as a separate task in the business logic. This allows for test driven development to be carried out much more effortlessly than if the logic is tightly coupled with the DOM manipulation. [10, question 4]

You can test your DOM manipulating code if you really need to, but be aware that there are many implementation differences depending on which browser or testing environment the tests are run in (on the other hand, this is one of the reasons why you might WANT to test the DOM manipulations) and that it can be hard to keep test fixtures up to date.

There are free cloud-based tools such as TestSwarm that are designed to help you with running the tests in many different browsers, or you can do it manually using JsTest-Driver, which also has built in support for setting up test fixture DOMs. One way is to use different Selenium drivers, then you can check almost anything, but beware that the tests become even slower so even for a medium sized application you might end up being unable to run the tests more than a couple of times per day.

6.3 (JS,ML0)Form validation

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

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

I have a real life example of a false positive in form validation. This year when I was about to buy a domain name, my personal number was interpreted wrongly so that I got an error message saying that I was not old enough. I reported the bug on twitter and it was quickly fixed so that I could finish my registration, but I imagine most people would have chosen another provider instead. Had there been proper testing of the form

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

validation, this would probably have been detected at an earlier stage, without losing potential customers. [47]

A similar bug manifested itself this year on a scandinavian internet bank. When entering the personal ID it got copied to the password field, and when entering the password it got copied to the personal ID field, so you had to use a cellphone identification service in order to log in. Such a thing would not happen if the bank had automated cross browser testing. [10, question 38]

So how to do this? You can of course use Selenium, to enter form data and submit and then check that the proper validation error was printed out at the right location. The problem with this is that if you have many validations that you need to test, these tests take a lot of time.

Depending on which framework you're using, you might have different options. If you are specifying the validation using regular expressions, hopefully you are able to expose these for isolated testing so that you can avoid testing all the validation in slow integration tests. These are still valuable, for instance to detect if there are any runtime errors or similar for the actual validation, but for complex validation such as testing which strings match a certain regular expression, you are typically better off with fast unit tests.

A common case is that you are using another programming language than JS for the back-end, and you specify the validation rules in that language and let a jquery plugin generate validations for the front-end. Here you can hopefully put regular expressions and the like as publicly available variables and access them directly in tests, whereas the plugin will have to be tested seperately, using perhaps a single test that just checks for existence of validation.

6.4 (JS,ML0)GUI testing

A Graphical User Interface (GUI) is usually the top layer of an application and is therefore a common target for system testing. It can consist of many components and typically provides a large number of possible interactions, which is why GUI testing is important, but also part of why it can be a time consuming activity both in terms of preparing and executing tests.

6.4.1 (JS,ML0)Application differences

A rather common opinion is that you should avoid automatic testing of an application's visual components. However, the value provided by GUI tests depends on how important the GUI is for the application. In web applications, the GUI and sequences in it are commonly more complex than the logic underneath, so testing the GUI makes more sense for such applications than for typical desktop or financial applications where backend logic tend to be more important. [10, question 21]

Before deciding on whether you want to do automated GUI testing or not, think carefully about how important the GUI and its cross browser compatibility is to you. If you feel that the exact appearance is irrelevant, then there is little sense in doing it. If you on the other hand feel that it would be valuable but have concerns about the technical aspects of it, you should not dismiss it too easily.

6.4.2 (JS,ML0)Regression testing of GUI

GUI testing is often confused with integration testing. The problem with this is that if you add testing of graphical elements on top of an integration test, it becomes harder to maintain and it can fail for more reasons. Instead of both executing a sequence and perform complicated assertions about graphical components of the expected result, you may benefit from keeping assertions simple in an integration test. The GUI testing can then be performed in a unit testing fashion as a separate activity, by creating a test fixture that mimics the state of the application the way it looks after a certain sequence has finished.

This approach has a fairly obvious risk. What if a test fixture fails to simulate an application state properly? You could detect this by comparing the application state with the corresponding test fixture in an integration test, deliberately introducing dependencies between tests. It might also be hard to construct such a test fixture to begin with, or you may feel that it introduces too much maintainability burden to have a test fixture that needs to be updated every time a relevant change occurs, in which case you have little choice but to either write bad GUI tests or skip writing them altogether.

Just as you might decide for or against implementing rounded corners that work in old browsers, based on the number of clients actually using such browsers and the maintenance cost of implementing it for every picture of a site, there is always a tradeoff regarding whether or not to test such graphical elements with automated regression tests or not. In the end, all that matters is cost versus gain, no matter what personal beliefs and preferences might be involved from the product owner or developers' personal point of view. [10, questions 40-44]

6.4.3 (JS,ML0)Automation of GUI testing

Manually testing a web page with all the targeted combination of browsers, versions and system platforms is usually not a viable option [48] so multi-platform automated testing is required. This is true in general for web application testing, but also holds for GUI testing in particular, regardless of whether the SUT is a web application or not. Since the appearance of a GUI can depend on factors such as configuration of hardware, operating system, version of third party software, etc., and because of the many possible sequences of interactions to test, it is rarely feasible to test all combinations and unless the testing is automated on multiple platforms, you will be unable to test even a small fraction of them.

GUI testing can be done in a semi-automated fashion. There are ways of producing snapshots of how websites look in different browsers and you should be able to have a program compare such snapshots to spot differences with previous versions, that can be reviewed by a human. This would reduce the need for manual testing of the GUI, since you get a good indication of which parts have changed.

Apart from writing tests in code using for instance TDD, there are techniques for automatically generating tests using crawlers or artificial intelligence. These techniques will by nature have more focus on coverage and detecting runtime errors than to simulate the behaviour and expectations of users. There have been quite a lot of research on this area.

Another technique is capture-replay, which is based on the concept that a user interacts with the SUT and records the actions. Selenium IDE provides this possibility, and there are other tools as well that can do this both for web and desktop applications.

A more modern approach is model based (FSM) GUI testing, where graphs are used to represent how the SUT can be interacted with and which states should be possible to reach. From such a graph, test cases can be generated.

6.5 (JS,ML0)Private APIs

Most problems with APIs, both external and private, are similar to problems with any external dependencies that you do not have full control over. The tests need to be kept up to date and in order to do so, integration tests are needed. Sometimes, testing towards a true implementation is problematic because it takes too long time, manipulates data that should not be touched or because the owner of the API does not accept large amounts of traffic or requires payment for it. Then you typically run the integration tests less often and stick to mocked APIs with fake objects to compensate. [22, questions 19-20]

Testing private APIs differs from testing calls to an external API in that the latter are often well documented, versioned and rarely changes. When using an external API, the main concerns are making sure that the correct version is used and that the API is called in a correct way with respect to that version. A private API on the other hand may change frequently which means that the fake objects representing requests and responses of the API in other tests need to change as well. Private APIs need to be tested because endpoints frequently change and may be used in several places. It is then important to have integration tests that can detect when the fakes need to change. [2, 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 characterises 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. [2, question 34]

Testing a private API is not the same thing as testing a private method. In JS 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. [2, questions 62-63]

7 (JS0,ML)Problems in testing of Standard ML

SML and JS have at least two things in common: they are both functional languages (although JS is perhaps more rarely used as such) and they both have poor testing cultures. This section relates how SML testing is carried out today to existing frameworks and formal verification, and introduces a new testing framework that was developed for SML as a part of this thesis, inspired by insights gained from looking at popular JS testing frameworks.

7.1 (JS0,ML1)Current Situation

The SML/NJ and Moscow ML language implementations[37][38], and other SML projects, have tests for large portions of the code, but there is no standard way of writing the tests. Each module typically "reinvents the wheel" by having its own scripts for testing, that is invoked differently and produces different output. If you search the web for "Standard ML testing framework" (with quotes, Nov 2013), you get zero results. Dig deeper, and you'll find QCheck/SML (henceforth referred to as QCheck) and a few small testing framework projects without any large user bases (see Appendix). It seems SML testing suffers from lack of motivation or knowledge among developers, or availability of good tools.

As a programmer, you should strive to do the simplest thing possible within the confines of given constraints, and to reuse as much as possible provided that any reuse candidates meets demands on quality and adaptability. YAGNI advocates this and is incorporated in many programmers' ways of thinking. When starting to write tests in a project, the pragmatic approach is to adhere to existing codebase conventions, or, in the absence of such, search the web for a suitable testing framework and use it in the project. Why is it that so many end up writing tests without using an existing testing framework? It seems that SML developers prefer writing their own testing framework rather than using an existing one.

As indicated by its 30 "stars" on github (Nov 2013), the currently most popular testing framework for SML is QCheck. However, it is somewhat cumbersome to use in scenarios where property testing is not the main objective. Some may decide against using it because they want something that is simpler, more suitable for TDD, or has a different

[37] `http://www.smlnj.org/svn.html`
[38] `https://github.com/kfl/mosml`

(e.g. BDD) style of defining tests and formulating failure and success reports. Without any production quality framework out there that satisfy these basic needs, it may feel like an easier task to do without a testing framework or write a new one than to familiarise with and adapt to an existing one that might turn out as disappointing anyway. The chance of developers deciding to use a framework relies on well prioritised features, active maintenance and a professional, practical or in other ways appealing presentation and reputation.

There are different driving forces behind testing different languages (such as JS and SML), because of the background and motives of people using them, and because of the diversity of the applications in which they are used. This partially explains why QCheck as the only truly popular testing framework for SML is based on generating test cases for properties rather than simple expectations/assertion based testing. The developer behind QCheck is Christopher League, a professor at LIU (Long Island University) Brooklyn, so his academic background in computer science is one of the answers. It could be that mathematicians are drawn to the language and are inclined to think in terms of properties and how they can be falsified rather than stating expectations on program state or output.

Another possible reason is that testing for expected output is easy even without the use of a testing framework, whereas QCheck solves problems that are not as easily solved ad hoc. Since SML has scripting capabilities, it is relatively easy to write and run tests without a testing framework, compared to doing it in languages that are more verbose and less often used in an interpreted setting, such as Java. Adding a testing framework becomes a bigger deal in SML for this reason. Possibly people simply do not find it worth their time to learn testing frameworks just to get readable tests and good failure reports, but instead are content with simply declaring tests as variables that evaluate to true or false, or similar. The consequences of doing this, however, are likely to include more debugging and risk of not noticing when tests fail, or ending up spending a lot of time scrolling through results of the ad hoc tests.

Yet another issue is the methods for distribution. In Node.js installing a package is very easy with the NPM and there is a central place to find popular node modules (`https://npmjs.org`). For SML there is the SML/NJ Compilation Manager (CM) and Smackage to choose between, and no central repository for modules except for github. Authors of testing frameworks might find it hard to decide which package manager (CM or Smackage) to use, figure out how to use it, or reaching out to users – which may also find learning how to use a package manager too cumbersome. Reputation, availability, ease of installation and use, and nice presentation may be essential for getting people to use a framework.

One area where testing is arguably very useful is for educational purposes, both for students when doing coding assignments and for teachers when grading. In order to encourage testing, the teacher may supply the students with small testing suites to build upon, although that requires emphasis on that the students should not consider an assignment to be done just because a test passes. A factor that makes testing especially fitting for programming assignments is that requirements are often relatively clear, and when they are not, testing can help in detecting those cases where additional information might have to be supplied or assumptions have to be made. Personal experience from using TDD as a student in the Programming Languages MOOC was that it greatly facilitated the coding process and was also useful for the peer reviews, both for being confident in refactoring code before handing it in and for checking the solutions of others and get direct feedback on which behaviour they might have missed to implement properly. I also have personal experience of working as a teaching assistant in an SML programming course, there too the testing suite that we used proved to be of great value to achieve objective evaluation of how well code handed in by the students satisfied the requirements, although we also evaluated style aspects and some non-functional requirements that were not feasible to evaluate using automated tests. Introducing testing early is an important step to encourage the understanding of testing principles for students and to improve efficiency for everyone in the long run. It is better that a student avoids faults through testing than being informed of them retrospectively in evaluation feedback.

Certain testing frameworks provide a watch capability, that runs tests whenever a change to the SUT is saved to the file system. In order for this to scale, only relevant tests should be run, or else the benefit of instant feedback is lost. Relevant tests are typically the ones that could be affected by changes in the specific files that have changed, but may also be selected based on speed or past failure rate. The idea is similar to having a build screen for a CI build, making not only the tests automated but also the task of running them frequently, and helps ensuring short cycles of writing a failing test, making it pass and refactor.

Mocking and stubbing may not be as important in a functional language like SML compared to more imperative languages, since dependencies can often be passed as arguments and functions tend to be behave the same way between calls given a certain input. Doing call assertions is as much about architecture as it is about state, and state is mainly kept in which data flows in and out of functions in a functional program, rather than in values of mutable variables as is often the case in imperative programs. However, there are certainly places where it would be appropriate to stub or mock a dependency in a function without having to redesign the function so that the dependency is passed as an argument, and in those cases the lexical scope rules and immutability of SML makes it hard to do so. It might be possible to implement a mocking framework that parses the code of a SUT and produce new bindings that have the desired dependencies and assertion capabilities, but such a framework would have to be quite sophisticated in order to not brake the code and is therefore outside of the scope for this thesis. A simpler approach would be to ensure that the any dependencies are bound to stub functions before the SUT is loaded, so that the stub definitions are used instead.

The strict type system of SML does not provide any generic toString method for the primitive types int, real, bool, etc. This means that in order to get good error reports from equality matchers, you either have to pass a type specific toString function, define matchers for each type: toEqualStr, toEqualInt, etc. or let the REPL (Read-Eval-Print-Loop) display the results (which is done differently depending on the SML implementation). You could also define the matchers in separate namespaces (structures), as is

	Formal verification	Testing
Finding bugs	medium	good
Proving correctness	good	useless
Cost	high	low

Table 1: Comparison between formal verification and testing

	Likely	Rare
Harmless	Testing	Unimportant
Catastrophic	Testing and FV	FV

Table 2: Verification strategies: Testing and FV (Formal Verification)

done in smell-spec[39].

QuickCheck, the Haskell predecessor of QCheck, resolves this by using the `where` keyword to specify the types of properties[40], which is elegant but unfortunately would not solve the problem in SML because types and structures can not be passed as arguments to functions.

7.2 (JS0, ML)Formal Verification

It has already been mentioned that QCheck is based on generating tests for properties defined in terms of the code. A similar but more rigorous approach is formal verification, which is performed by manually constructing mathematical proof of correctness, theorem proving, or deciding whether a model satisfies a given property (known as model or property checking). Model checking is most easily automated, either by specialised software such as CADP[41], UPPAAL[42] and many more[43] (a couple of them written in SML) or by using a constraint solver (see `http://www.cs.utah.edu/formal_verification/pdf/ligd_thesis.pdf`.

A problem with formal verification is its cost, because it can be so time consuming and requires special knowledge. This implies that unless bugs are potentially very severe, formal verification may not be worthwhile. On the other hand, if bugs have potential disastrous consequences, formal verification is the standard way of guaranteeing that certain properties hold, or prove the absence of certain defects. Testing on the other hand is mainly useful to detect bugs that are likely to occur. Tables 1 and 2 (pages 42 and 42) illustrate these differences between formal verification and testing. The tables are taken from lecture slides by Dr. Radek Pelánek[44], used with permission.

Kan formell verifiering ha liknande effekter på modularitet och utformning av gränssnitt som testdriven utveckling?

[39] `https://github.com/davidpdrsn/smell-spec/assertions.sml`
[40] `http://www.cse.chalmers.se/~rjmh/QuickCheck/manual_body.html#4`
[41] `http://cadp.inria.fr`
[42] `http://www.uppaal.org`
[43] `http://en.wikipedia.org/wiki/List_of_model_checking_tools`
[44] `http://www.fi.muni.cz/~xpelanek/IA158/slides/verification.pdf`

7.3 (JS0,ML)DescribeSML

By the time I found out about existing SML testing frameworks (apart from QCheck) I had already started to write my own. I wanted to use my insights from using JS testing frameworks such as Jasmine and Mocha to write a BDD framework which I would use in the Coursera MOOC course Programming Languages[45], as mentioned in section 3.2. The working title of it became DescribeSML.

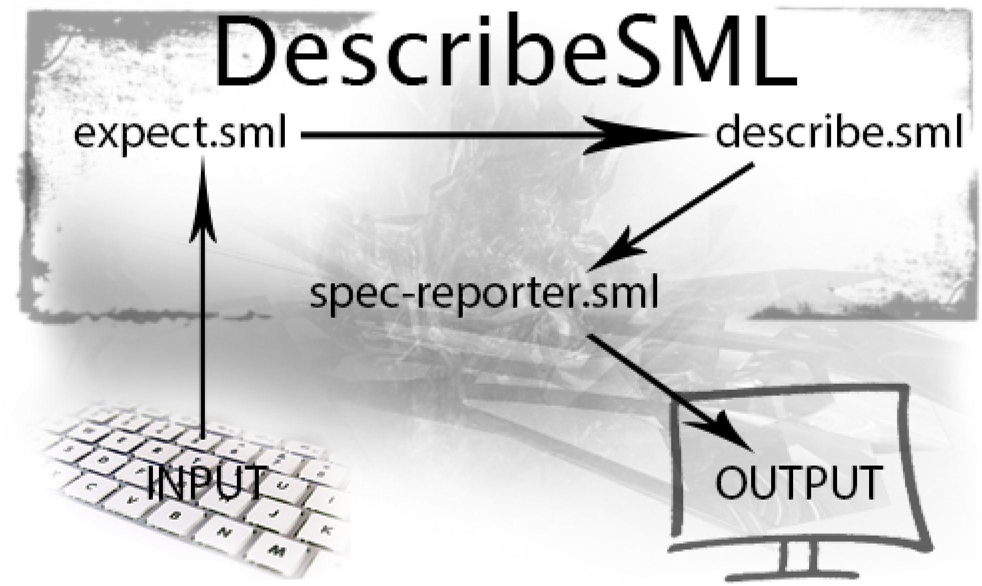


Figure 8: DescribeSML file structure and module architecture. Specs are written using Expect matchers, described and organised with Describe and should clauses, and the result of running them is reported by the SpecReporter. The arrows show how data flows through the application when the specs are executed.

Writing a testing framework in SML turned out to be an easier task than anticipated, it took only a few hours of development to get functionality such as matchers for checking equality, membership, regex matching and that certain exceptions are thrown. Nested describe-clauses were also implemented, to allow for neat organisation of specs, and the syntax and failure/success reports quickly became reasonably appealing.

Because of its modularity, DescribeSML could be used in many different ways. The `Expect` structure in expect.sml could be used on its own as an assertion framework, possibly wrapped or modified to signal success and failure differently than the `Describe` and `SpecReporter` structures do. However, the preliminary intention of usage is to define a test suite with the `suite` function, containing nested `describe` clauses, each containing specs defined with the `Describe.should`. The listing in figure 9 (page 44) shows an example of how it could look.

[45] `https://www.coursera.org/course/proglang`

```
CM.make "$/regexp-lib.cm";
use "spec-reporter.sml";
use "describe.sml";
val describe = Describe.describe;
val should = Describe.should;
use "expect.sml";
val expect = Expect.expect;
val toThrow = Expect.toThrow;
val toMatch = Expect.toMatch;
val toBeInt = Expect.toBeInt;
val toEqual = Expect.toEqual;

val foo = concat
    o (fn [] => raise Empty | lst => map (fn _ => "abc") lst)
    o explode;
datatype bar = Baz of int;

Describe.suite(describe "The SUT" [
describe "the function foo"
    [should("raise Empty when called with an empty string", fn _ =>
        expect (fn _ => foo "") toThrow Empty),

    should("return string containing 'abc' otherwise", fn _ =>
        expect (foo "string") toMatch "abc"),

    should("return string three times as long as argument", fn _ =>
        expect (size(foo "string")) toBeInt (3 * size "string"))],

describe "the datatype bar"
    [should("have constructor Baz, taking an integer", fn _ =>
        expect (Baz (7 + 35) : bar) toEqual (Baz 42))]
])
```

Figure 9: describe–sml–example.sml

An intriguing aspect of writing a testing framework is that you can use the framework to test itself. This may sound like circular reasoning, but actually worked quite well to detect defects immediately after they had been introduced and through TDD:ing ensuring a well thought through design and providing feedback about which parts are important and easy to use, and which parts are missing or deficient in some aspect.

Although the strict type system, functional style and static analysis of SML slightly reduces the need for testing compared to dynamic and imperative languages such as JS, it inevitably also results in more limiting testing frameworks. It is somewhat impractical in SML to state several expectations in a single spec or stub functions used by the SUT. However, this can also be a good thing, since it leads to more concise tests.

One way to compensate for this inflexibility could be to allow shared data between specs (see figure 10, page 45). Then it would not be such a problem that all specs have single expectations, since any value that you want to perform multiple assertions on can be computed once and then used in several specs. In an imperative language like JS this could encourage bad practises and lead to unwanted dependencies but since ML is (almost) purely functional, that is not an issue. Data could actually be shared between specs with a simple let-in-end expression already, but a future version of DescribeSML might implement a `given` construct to clearly demonstrate the possibility, and to better handle situations where an expression is raised by the expression producing the data.

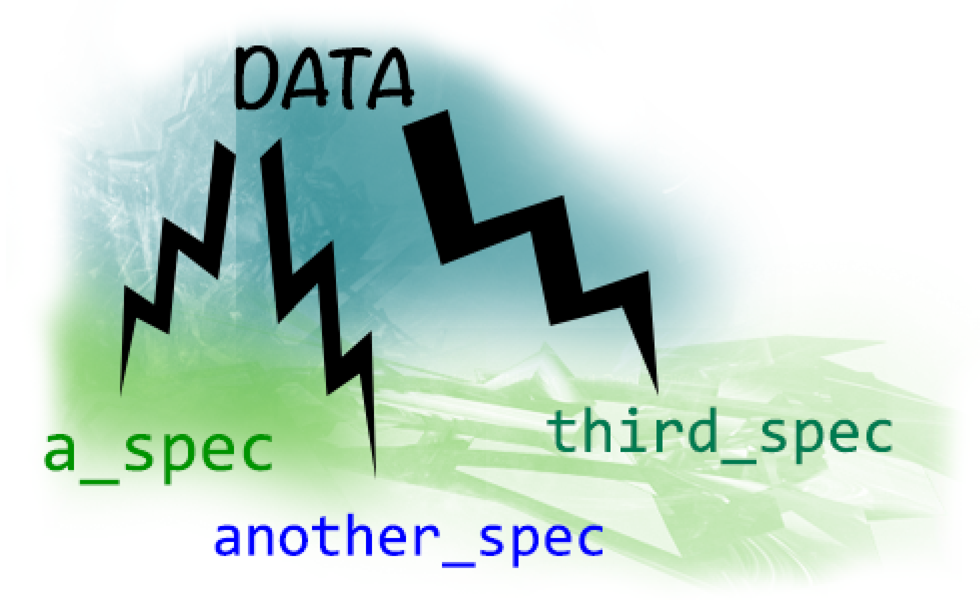


Figure 10: Sharing data between tests does not introduce dependencies between them in a functional language such as SML

Passing a toString function or letting the REPL display the results of an equality matcher are probably the only ways to print informative failure reports for user defined datatypes. The downsides are that if a toString function must be passed to each matcher, it imposes verbosity and makes the tests harder to read, and if the REPL is used the output can not as easily be written to a file or output as XML or other formats. It could be a good idea to use local definitions of matchers defined for a set of specs, for example:

```
describe "the function foo"
    let
        val toEqual = toBeInt
        val longInput = "string"
    in
        [should("return string three times as long as argument", fn _ =>
            expect (size(foo longInput)) toEqual (3 * size longInput))]
    end
```

However, this leads to increased indentation level and verbosity, and restricts the use of any local (possibly shadowing as in the example) matcher definition to the specified type, which is not always particularly convenient. The matchers that are otherwise hard coded in the framework due to the static typing are toEqual, toEqualStr, toEqualInt, etc. The worst part however is defining matchers for lists and tuples because you might end up needing things such as toContainStr, toEqualIntList or toEqualIntTuple5 (for instance, defined for int 5-tuples only) rather than toContain or toEqual which is what you really want. The real issue is getting informative and practical failure messages for all the types, user defined datatypes included, and to ensure that failure messages are output in a way so that you do not miss them by mistake.

Another consideration is that order may or may not matter for matchers of non-atomic data structures. This should be clear so that users of the testing framework are not surprised by the matchers' behaviour, and if one has to be picked over the other it should probably be that order does *not* matter, since the order of the elements can usually be checked anyway, in a way that makes it more explicit that it is supposed to matter. Not checking for a specific order unless necessary might increase the complexity of a matcher but will tend to make tests less implementation specific.

The naming of bindings is important in any language and a fundamental challenge that you face when learning to use a framework is what its functions are called and what they do. It is not always obvious for the people writing a framework what naming will come forth as the most natural and useful, for example there might be a tradeoff between conciseness and readability when choosing between naming a matcher toBeStr or toBeString. There is an option of binding both names to the same function, but that is considered bad practise because the user might expect that there are differences and the benefit of a uniform or idiomatic way of using the framework is lost.

Names generate assumptions regarding the semantics of a function. For example, in most languages there is a difference between two variables referencing the same object and two variables referencing objects that are equal. In SML, there is no notion of objects so this distinction is not as important. Moreover, there can be multiple references to the same value (as in `val a = ref v; val b = ref v`) and it is also possible to bind the same reference to several names (as in `val a = ref v; val b = a`), resulting in different results when checking if the references are equal although they do indeed point to the same value, depending on how the references were defined. This gave rise to the question whether there should be a difference between toEqual and toBe, and what the semantics should be. It was considered superfluous to define special matchers just for reference values, since they are used so sparsely and can be tested through dereferencing anyway. Based on that rationale, toEqual and toBe were bound to the same function in the first version of DescribeSML, which simply compared the arguments using the overloaded equality operator of SML. It can be argued that it would have been wiser to define just one of them, but it turned out that toBe felt more natural for primitive types such as booleans and numbers whereas toEqual felt more appropriate for composite data types such as tuples and lists.

A simple but important consideration of a testing framework is the phrasing of failure messages. In the first version of the DescribeSML reporter, some messages were formulated ad "expected "abc" to equal "def"". This made it somewhat unclear what was the expected and what was the actual value. Another bad example would be ""abc" does not equal "def"". Better alternatives include "expected "abc", but got "def"" or "was "def" but expected it to equal "abc"".

Är ordningen viktig? Kan vara bra att ha samma ordning som i testet. Är det viktigt om man skriver to equal eller to be? Om båda finns som matchers så kan det vara viktigt, men då tillkommer å andra sidan ytterligare komplexitet i testramverket.

Att skriva sitt eget testramverk är kul och inspirerar till att skriva mer tester. Om det redan finns bra testramverk, vilket oftast är fallet, så är det kanske i många fall dumt eftersom det kan vara svårt att motivera det extra arbetet inför kund, men jag vill ändå rekommendera det som hobbyprojekt, som en del av att lära sig TDD.

Första implementationen av toThrow visade sig strunta i vilket exception som kastades, däremot så rapporterade det vilket exception man förväntat sig skulle kastas om inget kastades öht. Detta är ett typexempel på något som man lätt upptäcker med testning, shame on me för att jag inte skrev tester för det. Det uppstår även hela tiden avvägningar om hur stringent man ska vara, om man till exempel ska nöja sig med att jämföra exnMessage eller om man även ska jämföra exnName.

Att skriva describe-funktionen var mer komplicerat än att skriva expect-funktionen. The matchers var av varierande svårighetsgrad, vilket var bra för det gjorde att jag kunde testa de mer avancerade med hjälp av de simpla.

Behovet av nästlade describes uppkom fort, så att man kan samla specar till flera funktioner som hör ihop utan att behöva slänga in alla i samma describe, och så att resultatet av samtliga describes visas längst ner, så att man inte missar att någon spec failar. Detta går in under "practical considerations" och är särskilt viktigt om man inte har tillräckligt stor skärm för att få plats med all output, men är egentligen alltid bra ur användbarhetsperspektiv.

Det finns ett behov av att fånga exceptions och representera dem som failures med passande felmeddelande. Detta kan göras av should-funktionen.

"The source code of the SML testing framework is supplied here, in the unlikely event that noone else has developed a better framework by the time you read this. It can also be found at github: "

When solving the first assignment, the instructions were often to use the "answer to the previous question". In this case, you would probably want to do a call assertion, which should be possible to do similar to how it is done in JavaScript, temporarily overwrite a definition of the function to stub, use side-effects to store information about calls to it, and then restore it (either automatically after the test has ended if the stubbing framework can be integrated with the testing framework, manually by calling a method, or super-manually by performing the stubbing using "vanilla-ML").

Möjlighet att specificera flera expectations per should känns som något många antagligen tar för givet, och som något jag själv känt behov av när jag använt det, men samtidigt så medför det vissa fördelar att alla ens tester är one-liners. Till exempel så blir det tydligare vilken/vilka expectation(s) som inte höll (men detta kanske skulle gå att få till ändå). En intressant aspekt är om expect-statements borde slänga exceptions, eller om should borde göra det, eller om det är bäst som det är nu, att ingen gör det. Det har även att göra med om själva anropen gör det (vilket i dagsläget hanteras av describes och toThrow).

Gör plugin för/till/integrera med qCheck: - Måste skicka med generatorer till checkGen

- Måste redirecta stdout och parsa utskrift från qCheck, eller använda qChecks egna reporter

setup/teardown/beforeEach/beforeAll/afterEach/afterAll-metoder heller inte lika viktiga (just pga att stubbning är en vanlig del av setup), men har fortfarande flera användningsområden

8 (JS,ML)Testing culture

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

8.1 (ML,JS1)State of events

The overall quality of JS code might have improved over the last few years partly as a consequence of testing methodologies entering the JS community. The new application frameworks has brought not only new technical possibilities (as mentioned in section 8.5) but also brought people with background from test-driven backend-programming, experienced with for example Ruby on Rails development, to JS. In fact, most of the new frameworks have been developed by people with such background. This has influenced the way JS code is written and given birth to a novel testing culture. [10, questions 12-15]

One may argue that TDD forces developers to write testable code which tends to be maintainable, and it may be true in some scenarios, but one has to bear in mind that JS is commonly used with many side-effects that are not so easily tested. More importantly, it is common to place all the JS code in a single file and hide the implementation using some variant of the module pattern[12, p. 40], which means that only a small subset of the code is exposed as globally accessible functions. In order to test the functions in any sensible way, the design typically will have to be improved so that no function has too many responsibilities. This conflicts with the eagerness of most developers to just get something that works without making it more complicated than necessary.

Many JS developers are used to manually test their JS in a browser. For someone not experiences with testing, this gives a relatively early feedback loop and although it does not come with the benefits of design, quality and automated testing that TDD does, it tends to give a feeling of not doing any extra work and getting the job done as fast as possible. Developers do not want to spend time on mocking dependencies when they are unsure if the solution they have in mind will even work. Once an implementation idea pops up, it can be tempting to just try it out rather than writing tests. If this approach is taken, it may feel like a superfluous task to add tests afterwards since that will typically require some refactoring in order to make the code testable. If the code seems to work good enough, the developer may not be willing to introduce this extra overhead, for good reasons [22, question 43].

As mentioned in section 1.1, tests should be maintainable and test the right thing. Otherwise, responding to changes is harder, and the tests will tend to cause frustration among the developers instead of detecting bugs and driving the understanding and development of the software [36]. Key points here are that tests can not substitute careful decision-making and that you also need to think of the maintainability of the tests themselves, they suffer from code rot just like production code so they need to be treated similarly [10, question 36][36, p. 123-133].

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 [18, p. 599]), be independent of implementations and have meaningful comments (if any). Structuring the code according to a testing pattern such as the Arrange-Act-Assert [49] and writing the code so that it reads like sentences will help in making the code more readable, in essence by honouring the communicate intent principle [18, p. 41].

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

Tests that depend on the DOM tend to be relatively slow, so it would be impractical to have hundreds of unit tests with DOM manipulation if you want to be able to run them often [22, questions 21-22]. Selenium tests are also known to be slow, the same principle applies there. Only running tests that are connected with changes that have been made is one way to get around this, there is support for this in many testing tools. An alternative is to separate the fast tests from the slow, and configure Autotest[46] or some tool based on Guard[47] such as guard-jasmine[48] to run the fast tests each time a file changes, and only run the slow tests before sending the code to a central source control repository or deploying the application.

8.2 (JS,ML)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[36], 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

[46] `http://autotest.github.io/`
[47] `https://github.com/guard/guard/`
[48] `https://github.com/netzpirat/guard-jasmine/`

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. [2, 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 behaviour is and coming up with examples that can be used as input and expected output in end-to-end tests. [2, questions 39-40][22, question 30]

The attitude towards and experience with testing varies heavily, so as a developer you can be forced to not test as much as you would otherwise want to, in order to avoid frustration and dips in productivity by testing other people's code. You should advocate methods that you believe in, especially in the beginning of a new project, but in the end adjusting to the culture is also essential since most benefits of testing appear only after a long term commitment by the whole team. One example is whether everyone uses their own database for testing, this is usually a bad idea but you have to decide whether it is worth the effort to create new databases for yourself, everyone, or switching to another, although these kinds of decisions might be limited by licenses and what the customer wants. [22, questions 31-32][10, question 36]

In order to understand the problems with testability that can be seen in JS applications, it is important to understand how many were first introduced to the language. When jQuery was released in 2006, a coding style evolved based on handlers with DOM manipulation through selectors and asynchronous callbacks. Since tutorials were written in a quick-and-dirty fashion, even experienced developers failed to apply principles for writing testable and maintainable code. Others lacked the background of software engineering altogether. [22, question 10]

An individual concern regarding testing is that it depends on being able to properly structure the code, which in turn tends to rely on experience with multiple programming languages and the ability to make pragmatic decisions, not just encapsulating and applying patterns for the sake of it, without understand why. Otherwise any efforts to produce good code or tests will fail On the other hand, as a beginner it might be better to rely on proven methods until you obtain proper contextual knowledge, rather than trying to get things exactly right from the start. Preferably, beginners should work with experienced developers to avoid risk of reinventing the wheel. [10, questions 23-24]

8.3 (ML0,JS0)Individual Motivation and Concerns

What defines successful testing is not which tools are used, the degree of coverage that is achieved or how strictly a certain set of principles or methodologies are followed. Successful testing is defined by how easy it is to make changes, find errors and understand the code. In order to get there, developers must feel that testing is meaningful and that they are allowed to spend time on testing as part of their professional tasks.

Some think that as long as the code works, testing is unnecessary [10, question 13]. Perhaps this is true for these people, testing is not for everyone, there are programmers out there that do not test their code but still manage to produce brilliant software. In most cases however, testing leads to better maintainability. The exception being when the tests themselves are not maintainable. [10, question 33][2, question 28]

It is important to remember that people tend to care most of things related to their guild, i.e. what they are good at and take pride in. The most embarrassing thing for a seasoned tester might be that a bug has leaked into production, and an art director may be more concerned that everything looks good and care less about the actual functionality. This can be seen in projects where, despite having the same economic risk, completely different approaches to quality control may be taken. In some projects it is ok to release a bug and roll back or update when it is discovered whereas others can spend an entire week performing manual tests, as part of a “hardening sprint”. Delaying a release by two weeks and having full time developers doing manual testing rather than implementing new functionality may give them a feeling similar to driving extremely slow in a sports car, not to mention the associated cost from a business perspective. This is why releasing and restoring previous versions should be as simple as possible. [10, question 38]

8.4 (ML,JS1)Project risks

Different projects have different fault tolerance. Sometimes a single bug can cause loss of millions of dollars or even lives, whereas in other projects the impact of a failure can be relatively small. The risk with untested JS is especially high when the code supports business critical operations. For instance, there have been several cases of banks and finance systems being fined for not reporting transactions to the government [50] or giving faulty advice to investors [51] due to application failure. A web-shop may lose orders and any website that is perceived as broken can harm trademarks associated with it and change people’s attitudes for the worse.

The connection between how much testing is carried out and how much risk is involved with a project is not as strong as one might think. The company culture and the background of the developers have great impact too, especially when it comes to economic risks. If the stakeholders are not used to testing and the developers are inexperienced with the tools of the trade, chances are that they will rely on manual testing or no testing at all, whereas utilising CI with good rollback capabilities can lead to qualitative and cost effective delivery without any manual testing. On the other hand, no company would rely on just CI if the application is safety critical, for example regulating nuclear coolant or flight navigation. In such scenarios there should be both automatic tests, rigorous manual testing routines and frequent validation reviews. [10, questions 14, 38]

8.5 (ML,JS)Frameworks

Programming languages in use today typically have frameworks to help with standard structure and other generic problems. JS is certainly no exception, the number of web application frameworks that focus on JS has increased a lot during the last few years alone. When application frameworks such as Backbone first appeared, they revolutionised JS the scalability and structure of applications [10, question 23][52, question 11]. The advent of application frameworks also gave new support for testing, for instance, 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" [53]

This likely contributes to increasing the testing of JS code by developers using such frameworks. This effect is accomplished both through tutorials and the way application frameworks are designed. For instance, the moustache templates of CanJS can be used to extract code from views, where it is easier to test and reduces view logic complexity [10, question 32].

There is a tendency of testing frameworks being tied to application frameworks and vice versa – Karma and protractor are for instance rather AngularJS specific, and angular-mocks is designed to work with Jasmine [52, question 7]. Several of the interviewees had experienced that some application frameworks are uncompromising when it comes to using only a subset of the framework or using another testing framework than what the application framework was designed for, although there are certainly exceptions to this rule as well, such as CanJS [10, question 22][52, question 41].

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 8.2. 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[49] is a popular framework that was designed for this in the Ruby programming language and there is a JS version called Cucumber.js[50]. Yadda,[51] Kyuri,[52] Cucumis[53] and Jasmine-given[54] are other examples of JS GTW-frameworks. [19, section 8.4]

Since different assertion frameworks have different syntax, they influence the readability of tests. This is presumably one of the main reasons why BDD frameworks such as Jasmine and Buster have become popular, despite lack of certain features such as `beforeAll` (which would be useful to construct test fixture setups) or sophisticated ways of selecting which specs to run based on more than just file names. There seems to be varying developer cultures behind these frameworks, that determine whether or not contributions from the community is allowed into the code base. [10, question 7][52, question 5-6]

Apart from testing frameworks, your ability to write tests is also influenced by what application framework you are using [10, question 8]. According to Edelstam, if you are able to choose, it is often preferable to have small rather than "magic bullet" frameworks for large projects because they tend to be less obtrusive. Exceptions may include when the framework is very well known to the developers or when there is a perfect fit between

[49] `http://cukes.info/`
[50] `https://github.com/cucumber/cucumber-js`
[51] `https://github.com/acuminous/yadda`
[52] `https://github.com/nodejitsu/kyuri`
[53] `https://github.com/noblesamurai/cucumis`
[54] `https://github.com/searls/jasmine-given`

what you want to achieve and what the framework is designed for, but one should bear in mind that requirements tend to evolve. [2, questions 48-50] Stenmark had another view on things and instead recommended small frameworks only for small projects, since the value of bindings and other features tend to outweigh adjustment problems. [22, questions 12-14]

Using a suitable application framework and doing test driven development can be mutually beneficial. The framework helps to create structure that simplifies testing and the testing process further improves the quality of the code, allowing you to make better use of the framework and providing incentive for even better structure. [22, question 15]

Aside from the increased awareness and knowledge of testing that Node.js has brought to the JS community from other communities such as Ruby on Rails, the runtime itself facilitates testing since it enables different testing tools to be distributed through npm (Node Package Manager) and run in the terminal. Previously, you had to load a html page in a browser in order to run your tests. [22, question 9]

9 (JS,ML)Conclusions

This section contains a summary of the main points of this thesis, a discussion about what could have been done differently, and suggestions for the future.

9.1 (JS,ML)Lessons learned

Both the interviews, the literature studies and the practical work indicated that one of the most crucial things for successful testing is finding suitable abstraction levels. If tests contain too much details they will be harder to read, and the same applies to functions in the production code, except that will also impact testability. It is therefore useful to introduce domain-specific language in tests [36, p. 127] and thereby encourage proper modularisation in the production code.

JS is used differently, it is not just another programming language. Because it is implemented in practically every browser, it is used by almost all web developers, regardless of their experience with programming. Frameworks such as jQuery also influence how people write their JavaScript code, often in a way that is negative to testing. On the other hand, other frameworks have the opposite effect and there is nothing inherently bad with using a powerful tool such as jQuery, you just need to be careful to separate code that does different things. If you don't know how the code works, as can be the case when copying code from an external source, then you can not do this. You also need to be aware of good practices.

The web is a difficult environment for testing because of many dependencies and events. This makes it harder to test, but also more important.

Striving for full coverage is not compatible with BDD and not being implementation specific. As a novice tester, it can be hard to find a good balance between writing many tests and writing good tests.

In a project without any testing routines, you may save time initially but there is no way of confirming that everything works as it should. Without automated tests or testing protocols, bugs will be harder to reproduce. Testing routines in the form of test plans can be hard to keep up to date and execute regularly since the testing activity depends on someone actually taking the time to do it. If something goes wrong you may have to repeat the procedure to ensure that it was not just an error in the testing. An absence of unit tests can lead to exorbitant amount of debugging in order to find the source of an error.

When automatic regression tests are missing, making changes to the code is error prone. Issues related to browser compatibility or subtle dependencies between functions and events are easier to overlook in the absence of tests, and a proper toolset is required to test multiple platforms.

10 (ML0,JS1)Insights from interviews

It seems that in the last couple of years, the number of people testing their JS has increased significantly[2, 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 JS is too user interface centred to recede, as people have realised 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. [2, 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. [2, 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. [2, 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. [2, questions 9-10]

In June this year, Kent Beck wrote the following tweet[55]:

> "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 [54]. What many probably fail to realise 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. [2, 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. [2, questions 11-12 and 20]

Error reports are useful feedback that tests provide. The quality of these reports vary depending on which testing frameworks you are using and how you write your tests. When using PhantomJS to run tests, some test failures require you to run the tests in a browser in order to get good error reports. [2, 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 JS 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. [2, questions 13-15]

One could argue that JS 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. [2, questions 20-21]

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

The problems that some people experience with testing front end 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. [2, 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. [2, 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 prfioritize 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. [2, 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. [2, 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 [2, questions 31-32]. Tests also help to give a clear image of how components fit together and can be a way to concretise a feature or bug.

10.1 (JS,ML)Future

It would be useful to have a fully automated solution for comparing snapshots in a smart fashion. This is possibly an area for image analysis, which could be integrated in testing tools to handle scenarios that are hard today.

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

11 (JS,ML)TODO/ideas

Översätta själv? Skriva tester på eget språk, domän/kommunicera med produktägare

JavaScript-människor skriver gärna egna ramverk, men vad krävs för att de faktiskt ska användas? (jmf MVC, MVVP, etc, där rätt många faktiskt används, vad krävs för att få samma genomslag)

Ta upp faux-bugg som exempel på problem som en crawler kunde löst, samt diskutera/spekulera kring varför folk tror att det är svårt

Olika drivkrafter/motivation till att skriva tester i JavaScript jämfört med ML - olika användningsområden

Description of work, referera fram i rapporten och använd som rubriker

API: mocka en nivå, kör tester, mocka nästa nivå, kör tester

Löser problemet direkt i webbläsaren, "visuell programmering", test first känns därför bakvänt för gränssnittare, som är problemlösningsorienterade

Teknisk aspekt: greenfield vs. existing code base (testability)

Förklaring av olika typer av tester: Enhetstester bra för att testa algoritmer. Exempel och liknelser: bilar och cykelhjul

Kolla upp coverage-verktyg för JavaScript, och om någon erbjuder rapporter för (cyklomatisk komplexitet)

Ställ frågor, fundera över vilka delar av rapporten som besvarar de frågorna, försök hitta frågor som alla delar är med i att besvara

References

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

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

[3] Ecma International. *ECMAScript Language Specification*. June 2011. URL: `http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf` (visited on 10/01/2013).

[4] Alexander McAuley et al. *The MOOC Model for Digital Practice.* 2010. URL: `https://oerknowledgecloud.org/sites/oerknowledgecloud.org/files/MOOC_Final_0.pdf` (visited on 10/30/2013).

[5] Robin Milner, Mads Tofte, and David Macqueen. *The Definition of Standard ML (Revised).* MIT Press Cambridge, 1997. ISBN: 0262631814.

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

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

[8] 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).

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

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

[11] 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.

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

[13] 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).

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

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

[16] 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).

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

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

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

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

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

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

[23] Jens Neubeck. "Testdriven Javascriptutveckling i webbapplikationer – En utvärdering av tekniker och möjligheter". MA thesis. Kungliga Tekniska Högskolan (KTH), 2010.

[24] Emden R. Gansner and John H. Reppy. *The Standard ML Basis Manual.* Cambridge University Press, 2004. ISBN: 0521794781.

[25] Jeffrey D. Ullman. *Elements of ML Programming.* 2nd ed. Prentice Hall, 1998. ISBN: 0137903871.

[26] Larry C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 1996. ISBN: 052156543X.

[27] Stephen Gilmore. *Programming in Standard ML '97: A Tutorial Introduction*. University of Edinburgh, 2003.

[28] Robert Harper. *Programming in Standard ML*. Carnegie Mellon University, 2011.

[29] Ute Schmid and Marieke Rohde. *Functional Programming with ML*. Universität Osnabrück, 2002.

[30] Riccardo Pucella. *Notes on Programming Standard ML of New Jersey*. Cornell University, 2001.

[31] Natalia Negara and Eleni Stroulia. "Automated Acceptance Testing of JavaScript Web Applications". In: *19th Working Conference on Reverse Engineering* (2012), pp. 318–322.

[32] Sebastien Salva and Patrice Laurencot. "Automatic Ajax application testing". In: *Fourth International Conference on Internet and Web Applications and Services* (2009), pp. 229–234.

[33] 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.

[34] 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.

[35] Phillip Heidegger and Peter Thiemann. "Contract-Driven Testing of JavaScript Code". In: *Objects, Models, Components, Patterns Lecture Notes in Computer Science?* (2010), pp. 154–172.

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

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

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

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

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

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

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

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

[44] 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).

[45] 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).

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

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

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

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

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

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

[52] Per Rovegård. *Interview.* Aug. 26, 2013.

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

[54] 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).