



UPPSALA
UNIVERSITET

UPTEC IT 14 002

Examensarbete 30 hp
Februari 2014

Rationales and Approaches for Automated Testing of JavaScript and Standard ML

Emil Wall



UPPSALA
UNIVERSITET

**Teknisk- naturvetenskaplig fakultet
UTH-enheten**

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
<http://www.teknat.uu.se/student>

Abstract

Rationales and Approaches for Automated Testing of JavaScript and Standard ML

Emil Wall

The ever increasing complexity of web applications has brought new demands on automated testing of JavaScript, requiring test-driven development to achieve maintainable code. A contrasting area is testing of Standard ML, another functional language but with important differences.

The aim of this thesis is to highlight the main problems with testing behaviour of applications written in these two programming languages, and how these problems relate to development tools and practises. This has been investigated based on the following research questions: What are the testability issues of client-side JavaScript and Standard ML? Which considerations need to be made in order to write stable and maintainable tests? How does testing culture affect productivity and quality of software?

Through quantitative interviews, implementation of the DescribeSML testing framework and development with tests in different scenarios, answers to these questions have been sought. The dynamic nature of JavaScript makes it more important to test whereas there are limitation on how Standard ML can be tested imposed by its static type system and immutability.

The conclusion was drawn that the main issues for testability are dependency management, how to test graphical interfaces, and maintaining separation of concerns. In order to write stable and maintainable tests, suitable tools and priorities are needed. The impact of testing culture depends on the ability to avoid time-consuming and unreliable tests. Harnessing the technological advancements, making continuous tradeoffs between rigour and simplicity, and applying pragmatism, ingenuity and persistence, are key to overcoming these challenges.

Handledare: Jimmy Larsson och Tobias Hasslebrant
Ämnesgranskare: Roland Bol
Examinator: Lars-Åke Nordén
ISSN: 1401-5749, UPTec IT 14 002
Tryckt av: Reprocentralen ITC

Populärvetenskaplig sammanfattning

När en webbsida skapas så innebär det ofta flera månader av kodande innan den kan börja användas, i vissa fall flera år. Ju mer avancerad funktionalitet sidan har desto mer tid tenderar utvecklingen att ta i anspråk och desto större blir risken för buggar, särskilt om det är många personer som arbetar med sidan. När sidan väl är lanserad så återstår vanligen en ännu längre tid av drift och underhåll. I det skedet upptäcks och åtgärdas de eventuella buggar som inte hann upptäckas under utvecklingsfasen och det tenderar att ske omfattande förändringar. Det kan vara en stor utmaning att göra detta utan att introducera nya buggar eller förstöra andra delar av sidan, och förändringarna tar ofta längre tid att genomföra än man tror.

En webbsidas skick kan idag vara avgörande för om det uppstår ett förtroende gentemot ett företag, eftersom den ofta utgör ett första intryck. Vissa företag sköter även en stor del av sin centrala verksamhet genom webbsidor så om en sida inte fungerar kan till exempel beställningar eller annan information gå förlorad. Företag kan alltså få både ökade kostnader och förlorad inkomst av buggar, vilket är varför det behövs noggrant förebyggande arbete.

Ett av de vanligaste sätten att minimera förekomsten av buggar är testning. En variant är att manuellt kontrollera funktioner, till exempel genom att lägga upp varje ny version av en sida på en testserver där man klickar sig runt på webbsidan och kontrollerar att saker fungerar som de ska innan man laddar upp den så att allmänheten kommer åt den. Detta är en billig metod på kort sikt, men det innebär ständig upprepning av tidskrävande procedurer så det kan visa sig ohållbart på längre sikt. Det medför även behov av manuell felsökning för att ta reda på varför något inte fungerar och det kan vara svårt att veta hur något är tänkt att fungera om kraven inte är tillräckligt formaliserade.

Ett alternativ är automatiserade tester, som kan testa specifika delar av koden eller simulera en användares beteende utan mänsklig inblandning. Det är den typen av tester som den här uppsatsen fokuserar på. Mycket av den teknik som används för att skapa hemsidor är förhållandevis ny och det tillkommer hela tiden nya sätt att arbeta på, så det finns ett stort behov av översikt och utvärdering av teknologin. Det finns även mycket att lära genom att jämföra denna teknik med testning i andra sammanhang och i programmeringsspråk som normalt inte används för webben. En fullständig kartläggning av alla tekniker som går att koppla till testning av webbsidor vore dock ett alltför stort projekt för detta format, så denna uppsats är avgränsad till att endast behandla JavaScript och Standard ML, två programmeringsspråk med intressanta likheter och skillnader.

JavaScript används inom webbutveckling för att skapa interaktiva sidor med minimal trafik mot servrar. Standard ML å andra sidan är uppbyggt annorlunda och används oftast i helt andra sammanhang, men stödjer i grunden en liknande programmeringsstil. Det har den senaste tiden skett framsteg inom testning för dessa programmeringsspråk, mycket tack vare att information om hur man går tillväga har blivit mer tillgänglig, nya ramverk gör att kod struktureras på ett annat sätt än tidigare och det har uppstått fler och stabilare sätt att köra koden på. Här ges en överblick över dessa framsteg, och de

många tekniker som idag används för testdriven utveckling.

Genom egen utveckling och intervjuer med andra utvecklare har problem och lösningar inom området undersökts. Några av de större svårigheter som identifierats är att vissa mekanismer (asynkront beteende, DOM-manipulation, grafiska komponenter) är svåra att testa utförligt utan att testerna blir opålitliga eller långsamma, att det sällan lönar sig att skriva tester i efterhand, att det kan vara svårt att hålla tester lättförståeliga och uppdaterade, att utvecklingsverktygen kan vara begränsande, och att programmeringsspråkens uppbyggnad och typen av applikation har viss inverkan på vad som är möjligt och lämpligt att testa. Det spelar även stor roll vilka erfarenhetsmässiga och kulturella förutsättningar utvecklarna har för att skriva tester.

Testning i sig är inte en lösning på alla problem, och olika former och nivåer av testning lämpar sig för olika situationer, men när det används rätt så kan det gynna både användare, organisationer och utvecklarna själva. I det här arbetet presenteras ett testningsramverk som producerats för Standard ML och en översikt av hur automatiserad testning kan gå till, med fokus på testdriven webbutveckling med JavaScript. En viktig slutsats är att testning behöver ses som ett verktyg och inte ett självändamål och att utvecklare behöver hitta de metoder och verktyg som fungerar bäst för dem.

Acknowledgment

Thanks goes to my supervisors Tobias Hasslebrant and Jimmy Larsson for providing me with valuable feedback and connections, to my reviewer Roland Bol for guiding me through the process and giving useful and constructive comments on my work, to the people I have had contact with and interviewed as part of this work, to all my wonderful colleagues at Valtech that never fail to surprise me with their helpfulness and expertise, and to my family and friends (and cats!) for all the little things that ultimately matters the most.

Contents

1	Introduction	3
1.1	Motivation	4
1.2	Background to Project	7
1.3	Scope and Delimitations	8
1.4	Organisation of Thesis	9
2	Previous Work	10
2.1	JavaScript Testing	10
2.2	Standard ML Testing	11
3	Methods	12
3.1	Literature Study	12
3.2	Programming	12
3.3	Interviews	13
3.3.1	Interview Considerations	13
3.3.2	The Interviewees	14
3.3.3	Further Contacts	15
4	Technical Background	15
4.1	Principles in Testing	15
4.2	Test-Driven Development	16
4.3	Behaviour-Driven Development	17
4.4	Spikes in TDD	17
4.5	Refactoring and Lint-like Tools	18
4.6	Mocking and Stubbing	20
4.7	Browser Automation	20
4.8	Build Tools	21
5	Testability – Real World Experiences	22
5.1	The Asteroids HTML5 Canvas Game	22
5.1.1	Getting Started	22
5.1.2	Attempts and Observations	24
5.1.3	Design Considerations	28
5.1.4	GUI Testing Considered	29
5.2	Tool Issues	30
5.2.1	JsTestDriver Evaluation	30
5.2.2	PhantomJS	32
5.2.3	Sinon.JS	33
5.3	Lessons Learned	34
5.3.1	General Issues With Adding Tests to an Existing Application . . .	34
5.3.2	Stubbing vs Refactoring	35
5.3.3	Deciding When and What to Test	36
5.3.4	A Related Experience	37
6	Problems in JavaScript Testing	38

6.1	Asynchronous Events	38
6.2	DOM Manipulation	39
6.3	Form Validation	40
6.4	GUI Testing	42
6.4.1	Application Differences	42
6.4.2	Regression Testing of GUI	42
6.4.3	Automation of GUI Testing	43
6.5	External and Internal APIs	44
7	Problems in Testing of Standard ML	45
7.1	Current Situation	45
7.2	Formal Verification	47
7.3	DescribeSML	48
7.3.1	The Framework	48
7.3.2	Alternatives Considered and Discarded	49
7.3.3	Using the Framework	54
7.4	Test-Driven Development of Programming Assignments	54
8	Testing Culture	56
8.1	State of Events for JavaScript	56
8.2	Culture, Collaboration and Consensus	58
8.3	Individual Motivation and Concerns	59
8.4	Project Risks	60
8.5	Frameworks	60
9	Conclusions	62
9.1	Lessons Learned	62
9.1.1	Testability Issues in JavaScript	63
9.1.2	Testability Issues in Standard ML	63
9.1.3	Stable and Maintainable Tests	63
9.1.4	Testing Culture	63
9.2	Future	64

Glossary

- Automated Deployment** Auto-deploy enables automatic release to test or production, possibly as part of a Continuous deployment strategy. 6
- BDD** Behaviour-Driven Development (BDD), see section 4.3 for a definition. 3, 9, 10, 15, 17, 18, 38, 45, 48, 55, 61, 63
- CI** Continuous Integration (CI) is a practice based on frequently merging new code with a main code repository, commonly using principles such as automated build, testing and deployment. 6, 16, 31, 46, 60
- CLI** A program with a command-line interface (CLI) is controlled by clients through successive lines of text (commands) input in a console. 21
- 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, 38–41, 57, 59
- 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 there is a clear picture about how alike the two scenarios actually are, if necessary. [2, questions 69-70]. 19
- GUI** A graphical user interface (GUI) provides, unlike a CLI, visual interactions and feedback for a client. 15, 38, 42–44, 48, 55
- 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. 3–15, 17, 19–22, 26, 29, 32, 33, 37–41, 44–46, 49, 50, 55–57, 59–65
- Matcher** BDD terminology for an assertion. Also commonly called expectation. 47–52
- 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, 45, 48, 54, 55

- 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]. 3, 4, 6–13, 19–21, 45–49, 51, 52, 54–56, 62, 63
- Spec (specification)** BDD terminology for a test. 17, 28, 49, 50, 52
- SUT** A system under test (SUT) is code intentionally exercised by a test. A unit under test (UUT) is a SUT that is clearly delimited from the rest of an application. 20, 30, 35, 36, 43, 44, 46, 47, 49
- TDD** Test-Driven Development (TDD), see section 4.2 for a definition. 10, 12, 14–18, 34, 35, 38, 40, 43, 45, 48, 53, 55–57, 59, 61
- 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 [6, question 5]. 40, 42, 43, 62
- 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. 3, 54
- YAGNI** The XP (Extreme Programming) principle YAGNI (You Ain’t Gonna Need It) is about not introducing anything except when entirely sure that it will be needed. 15, 45, 53

1 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 [7, 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 [8]:

“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 [8] (Boston Ruby Group presentation, 2012):

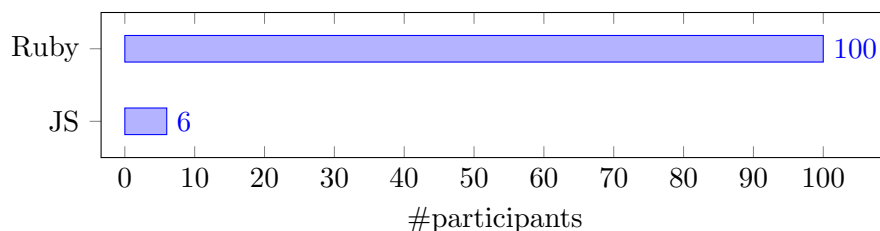


Figure 1: At a Ruby Group meeting in 2012, there was a clear distinction between how many participants were engaged in testing of JS compared to Ruby.

The goals of this thesis are to:

1. Highlight testability issues of client-side JS and SML
2. Describe practices for constructing stable and maintainable tests
3. Identify and discuss problems related to developer culture and technology

This section contains the background and scope, and an overview of the organisation of the thesis.

1.1 Motivation

Why testing, one may ask. Jack Franklin, a young JS blogger from the UK, gives three reasons:

1. It helps you to plan out your APIs
2. It allows you to refactor with confidence
3. It helps you to discover regression bugs (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. [9] Without tests, the ability to refactor (see section 4.5) 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 [10, 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 [11, p. 35-37].

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 can also serve as documentation and monitoring of the code, showing

how components fit together and concretising features and bugs. Provided that 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 [12, question 38].

Figure 2 shows 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.

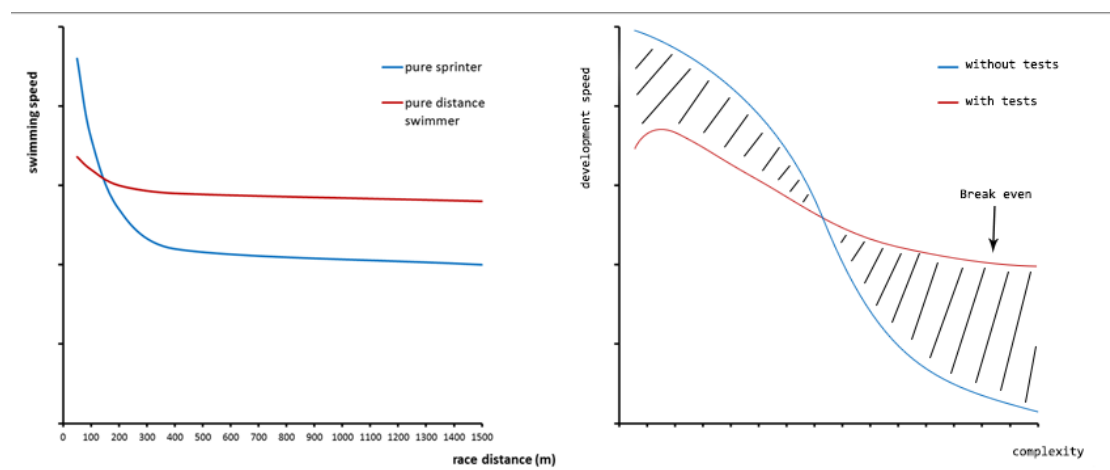


Figure 2: Long distance swimmers are initially slower but have better endurance than sprinters. A software development project displays similar properties, developing with tests is like training for long distances. If a project will be large enough to reach break even, testing will pay off. The graph to the right is not based on any exact data, but merely a sketch based on intuition. Image courtesy of graph to the left: Paul Newsome at www.swimsmooth.com

Implementations of the JS language differ, especially for host objects such as `document` and `XMLHttpRequest` 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 [7, ch. 10.2.3]. Automated testing can help discover bugs that appear due to such deviant behaviour and detect changes between versions. JS is also particularly important to test due to its dynamic properties [13] that complicate static analysis (however, there are lint tools, see section 4.5), 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 [14, 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 [13].

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 [15] and its applications have become increasingly complex [16, 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 more 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 harnessing 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 Background to Project

This thesis was written at Valtech AB, an IT consulting company based in Stockholm, with 185 employees (2013) specialised in digital strategy and full stack web and mobile development. The company has an interest in techniques for testing JS, but the subject of this thesis was left to the author to decide.

The first known JS testing framework JsUnit¹ was created in 2001 by Edward Hieatt [17] and since then several other test framework has appeared such as the testing framework for jQuery: QUnit², and JsUnits sequel Jasmine³. There are also tools for mocking such as Sinon.JS⁴ (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 [8] and thus, they lose the benefits, both short and long term, otherwise provided by testing.

There is only one production quality testing framework available for SML, namely QCheck⁵. 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, an increased use of SML testing in education (see section 7.4), or an indication that testing in general has been increasingly popular over the last couple of years, as can be seen 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 [16, question 3] although it has become better compared to a couple of years ago [12, 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 written examples 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 [10, 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 prob-

¹<https://github.com/pivotal/jsunit>

²<http://qunitjs.com/>

³<http://pivotal.github.com/jasmine/>

⁴<http://sinonjs.org/>

⁵<https://github.com/league/qcheck>

lem 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 Scope and Delimitations

The scope of this thesis is mainly limited to *automated* testing of SML and *client side* JS. As already mentioned in the beginning of this introduction, the goals are to investigate what the main problems within these two areas are and how they relate to development tools and practices. What are the testability issues in each respective language? Which considerations need to be made in order to write stable and maintainable tests? How does testing culture affect productivity and quality of software?

The impact of testing frameworks specialised for server side JS code (node.js) such as vows⁶ and cucumis⁷ was not considered during the project. Testing client side code is not necessarily more important than server side, but in many aspects client side testing 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 JsUnit⁸ and JSpec⁹ was deliberately left out of consideration. Others were left out because of a smaller user 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 it is outside the scope of test-driven development and 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 5.2.1 and 8.5)

⁶<http://vowsjs.org/>

⁷<https://github.com/noblesamurai/cucumis>

⁸<https://github.com/pivotal/jsunit>

⁹<https://github.com/liblime/jspec>

- Limitations in what can be tested (section 5)
- Complexity in setting up a test environment; installing frameworks, configuring build server, exposing functions to testing but not to users in production, etc. (section 5)

An important part of the scope has been to account for how to proceed conveniently 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 to introduce available tools and frameworks. Many tutorials for testing frameworks today tend 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 edge cases of how it can be used that might not be relevant to that many anyway. To provide guidance in how to set up a testing environment and how to write the tests, attention was 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 testing and application frameworks is not within the scope of this thesis, but others have done it [9][18]. 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 Chutzpah¹⁰ 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.

1.4 Organisation of Thesis

This thesis is about difficulties and experiences with JS and SML testing, and has been organised in favour of readers mainly interested in one of these programming languages. It contains a case study on testability aspects of adding tests to an existing JS application (section 5), problems specific to JS and SML testing (sections 6 and 7), considerations from writing and using a BDD framework in SML (section 7.3) and implications of testing culture that has been researched through interviews (section 8). These sections are preceded by an overview of what others have done in the fields of JS and SML testing (section 2), the methods that were used for writing this thesis (section 3) and a technical background that explains some of the concepts that appear in the rest of the thesis (section 4). The final section is conclusions (section 9), including summaries and proposals of future work.

Readers experienced with or uninterested in basics of testing and web application development may skip the technical background (section 4). Readers interested mainly in JS testing may skip sections 2.2, 7 and 9.1.2. Readers interested mainly in SML testing may skip sections 2.1, 4.7, 5, 6 and 9.1.1.

¹⁰<http://chutzpah.codeplex.com/>

The glossary, which is located before this introduction, contains definitions of abbreviations and technical terms used throughout the text.

2 Previous Work

This section contains an overview of what others have done within the fields of JS and SML testing. Particular interest is paid to research – lists of relevant frameworks and tools can be found in the Appendix.

2.1 JavaScript Testing

In 2010, a student at the Swedish royal institute of technology called Jens Neubeck wrote a master thesis about test-driven JS application development [19]. 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. Tools such as JsTestDriver and Jasmine were not considered and the results were based purely on original work, with no JS specific academic sources, so it has not been possible to build upon his findings here.

The main source of reference within the field of JS testing today is *Test-Driven JavaScript Development* [7] by Christian Johansen, which deals with JS testing from a Test-Driven Development (TDD) perspective. Johansen is the creator of Sinon.JS¹¹ 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 exercises. It is not very scientific but makes up for this with its pragmatism and roots in the software industry.

Today blog posts and books about JS are in abundance and examples can often be found in the documentation of frameworks. When it comes to examples of testing in general, there are several classics to refer to [20][21]. For examples of JS testing specifically the alternatives have been scarce historically, but recently a large number of books about JS testing has been published. *JavaScript Testing, Beginner's Guide* [22] is an introductory book about JS that covers some aspects of testing, *JavaScript Testing with Jasmine* [23] covers the Jasmine testing framework in detail, *Behaviour Driven Development with JavaScript* [24] presents a BDD perspective of JS testing, *JavaScript Unit Testing* [25] looks at the assertions and asynchronous testing capabilities of Jasmine, YUI Test, QUnit and JsTestDriver, *Using Node.js for UI Testing* [26] covers ways of automating testing of web applications with Zombie.js and Mocha, and *Testable JavaScript* [27] looks at ways of reducing complexity of JS code and discusses principles and tools (mainly YUI Test) for JS testing and maintainability in general. All of these were published this year (2013), except *JavaScript Testing, Beginner's Guide* which was published the same year as Johansen's book, in 2010.

¹¹<http://sinonjs.org/>

There are many academic articles about testing web applications available, and quite a few of them focus on JS specifically [13][28][29][30][31]. There is also a lot of material on testing patterns and how to write concise, useful and maintainable tests [21, part III][24, ch. 3-5][7, p. 461-474][27, p. 86-87][23, p. 13-14].

A Framework for Automated Testing of JavaScript Web Applications [13] 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 (see section 5.3.1 for further discussion). Automatically generated tests can be harder to maintain and will tend to fail for unintended reasons as the code changes, unless the tests are re-generated.

Automated Acceptance Testing of JavaScript Web Applications [29] 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 [31]. Ocariza et al. have investigated frequency of bugs in live web pages and applications [30]. 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 [28].

2.2 Standard ML Testing

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* [33] which describes the standard library, *Elements of ML Programming* [34] which cover the features of SML in a more comprehensible fashion, *ML for the Working Programmer* [35] which is somewhat outdated but comprehensive, and various lecture notes [36][37][38][39]. 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 [37, p. 16].

Articles on testing of SML are hard to find, none cover SML testing within a web applications context and most 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. However, the ideas of QuickCheck has been directly applied to SML in the QCheck testing framework, which is covered in *Random Testing of ML Programs* [40], a master thesis from 2010. In QCheck, tests are specified in terms of

properties 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 may be uncertainty in how well the properties are actually tested.

3 Methods

The methods that were used in this thesis comprise situational analysis, interviews, and programming activities in JS and SML. The work of this thesis began with an extensive literature study and an overview of existing technologies and frameworks. Interviews of JS developers of different background were performed and analysed. There were also hands on evaluation of tools and frameworks, assessment of testability and impact of adding tests to existing projects, and a small testing framework was developed in SML and used in a MOOC (Massive Open Online Course).

3.1 Literature Study

As mentioned in Previous Work (section 2), several new titles were published while writing this thesis, so the literature study continued to the very end. The books, articles and internet sources served both as reference to complement the interview material and as starting point for many of the experimental testing activities that were carried out (see next subsection).

Since there was an abundance of material on JS testing, a full review was not viable. For SML testing on the other hand, there were virtually no material on SML testing available, so any claims had to be based on practical observations and comparisons with formal verification instead. Since many relevant facts and principles hold for more programming languages than just these two, some classical works within the field of testing were included in the study as well.

3.2 Programming

In order to describe ways of writing tests for JS, the practical work involved adding tests to an existing JS application (see section 5.1), performing TDD exercises from Test-driven JavaScript Development [7, part III] 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.

An SML testing framework (see section 7.3) was developed and used to solve programming assignments in the Coursera Programming Languages MOOC. This provided experience of using TDD in SML (see section 7.4), allowed for application of insights from

looking at JS testing to SML dito, and made clear which problems within SML testing are specific to SML and which are not (see section 7).

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 testing frameworks that were part of the practical work are listed in the Appendix. Apart from the MOOC programming assignments, all code is publicly available on my Github account *emilwall*, together with the L^AT_EX code for this report.

3.3 Interviews

The JS community is undergoing more rapid changes than the SML community, so interviews were focused on JS to obtain up-to-date information about how it is currently used. They were first and foremost qualitative in nature, carried out as semi-structured case studies in order to prioritise insight into the problem domain and gather unique views and common experiences, which might not be picked up in a standardised survey or other efforts to quantitative research methods. The interviews were between 20 and 60 minutes long and were conducted both in person and via Internet video calls.

3.3.1 Interview Considerations

The preparations before the interviews included specifying purpose and which subjects to include, select interviewees, preparing questions and adjust the material to fit each interviewee. The chance of finding the true difficulties of JS testing was expected to increase with open questions. The interviews took place once 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.). Unfortunately no female candidate was available, due to the skewed gender representation among JS developers. There were five interviews and some email conversations, 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 agreed to be 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 were 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.

3.3.2 The Interviewees



Figure 3: The interviewees, in order of appearance. Clearly, there is a skewed gender representation among JS developers, since all candidates were men.

1. 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.
2. 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.
3. Marcus Ahnve, a senior developer and agile coach who 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.
4. 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 has given 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 August over Skype.
5. Henrik Ekelöf, a front-end developer who 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.
6. Fredrik Wendt, a software development teacher and consultant at Squeed AB, specialised in coding dojos and other skills development activities. Experienced in

Scrum, TDD, Java and web development. The interviews were held exclusively by mail.

3.3.3 Further Contacts

As can be seen at the end of the Appendix, there were some additional email conversations. Among those were: Fredrik Wendt, 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 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. The topics covered are principles in testing, TDD, BDD, spikes, refactoring, stubbing, mocking, browser automation and build tools.

4.1 Principles in 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 (see figure 2), 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 graphical user interface (GUI) to interact with [12, question 20]. Thinking in terms of APIs, there is usually a possibility of defining what each part should do, so tests can be written in an outside-in fashion, which reduces the risk of developing something that eventually turns out to be redundant or misdirected (remember, YAGNI (You Ain't Gonna Need It)) [12, question 29]. The downside of this approach is that the code required for such a high level test to pass might be complex, so intermediate tests need to be written. Such a test, that is in a failing state for an extended period of time, is sometimes called a pending test [12, question 31]. 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 [11, p. 132-133].

There are some desirable properties for unit tests, they should be fast, stable and to the point [12, questions 16-18][41, mail conversation][16, question 12]. To avoid slow or unstable unit tests and assure that they can be run without an internet connection or in parallel, outer dependencies such as databases, external APIs or libraries commonly have to be abstracted away through stubbing or mocking (see section 4.6). Unit testing suites that are difficult to set up, not frequently brought into a state where all tests pass, or take too long time to run, should be avoided [12, question 36][6, question 2][41, questions 21-22], whereas integration tests are typically slower and require more advanced setup, but they to should be automated and as stable as possible [41, question 37].

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 Test-Driven Development

Test-Driven Development (TDD) is “an iterative development process in which each iteration starts by writing a test that forms a part of the specification we are implementing” [7, p. 21].

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 TDD practitioner has written an insufficient number of 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 using TDD, instead relying 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 [6, 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 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 spec (specification)s) that helps readability and to make failure messages more helpful. [12, questions 17-18]

An advantage with BDD is how it encapsulates ideas about how tests can be organised, for instance through the Given, When, Then (GWT) form (covered in greater detail in section 8.5). In today's BDD frameworks there is often a possibility to separate the basic functionality from the special cases by organising specs in nested describes. This can provide an overview of what an application does just by looking at the spec output and is commonly seen in open source projects [41, question 42]. Providing a context for specs in this way 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, 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. [12, question 19]

Strictly speaking, a BDD framework is not required 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 to prioritise the parts that truly matters from a business value point of view over implementation details. [12, question 20]

4.4 Spikes in TDD

Although writing tests first is a recommended approach in most situations, there is a technique for trying something out before writing tests for it, without compromising testability. Dan North, the originator of BDD, came up with a name for this technique: spiking, which he confirmed on 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”.¹² The idea is to create a new branch in the version control repository, and hack away. Add anything that might solve the problem, don't care about maintainability, testability or anything of the sort. When not sure how to proceed, discard all changes in the branch and start over. As soon as an idea about how a solution could look like emerges, switch back to the previous branch and start coding in a test first fashion. [2, question 59]

¹²<https://twitter.com/tastapod/statuses/371352069812527105>, 2013-08-26

There is an ongoing discussion about whether or not to always 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 [42]. 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” [43]. It may seem like an elitist point of view and a sacrilege towards TDD principles but in the end, whatever maximises productivity 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 focus on what tests can be written 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. Write one or several acceptance tests for how the API should be used, then start to work with that feature in a TDD fashion [12, question 30]. Although not a perfect metaphor of Spike and Stabilise due to the lack of a stabilise step, this way of thinking and testing from the outside in can lead to a useful rationale regarding when to test – add tests whenever an external dependency is introduced, to make sure that the dependency is called correctly under certain circumstances, then if that dependency is something that already exists it can just be added, otherwise developed in a test-driven fashion.

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 an awareness of the bigger picture facilitates prioritisation and makes sure everything fits together in the end. There is the same risk with tests as with other pieces of code, sometimes pride generates an unwillingness to throw them away. In order to avoid that situation it is often better to think ahead and try things out rather than immediately spend time writing tests. [2, question 27]

4.5 Refactoring and Lint-like Tools

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 [10, p. 46]. As previously mentioned (section 1.1, Motivation), 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. Suitable situations to refactor are 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, for example in code reviews [10, p. 49-51].

There is a serious risk involved in refactoring untested code [10, 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.

A lint-like tool uses static analysis to detect syntax errors, risky programming styles and failure to comply to coding conventions. The use of lint-like tools can be beneficial when refactoring to avoid introducing errors, although it can not fully compensate for lack of tests. There are lint tools available for JS such as JSLint, JSHint, JavaScript Lint, JSure, the Closure compiler and PHP CodeSniffer. JSLint does provide some help to avoid common programming mistakes, but does not perform flow analysis [44] and type checking as a fully featured compiler would do, rendering proper testing routines the appropriate measure against programming mistakes. There is at least one lint tool available for SML, namely SML-Lint¹³, but since SML is statically typed the need for such a tool is not as great.

Apart from lint-like tools, there are also tools that can be very helpful to see which parts of the code are in most need of refactoring, and to automate certain refactoring actions by facilitating renaming, method extraction, etc. [45, ch. 5]. Refactoring tools help identifying sections of the code with high cyclomatic complexity, too many methods or with methods doing too many things (or having 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 [46]. There is some support for refactoring JS in IDEs such as Visual Studio (using JSLint.VS¹⁴, ReSharper¹⁵ and/or CodeRush¹⁶), WebStorm IDE¹⁷ (or IntelliJ Idea¹⁸ using a plugin) and NetBeans¹⁹. There are also standalone statistical tools for JS such as JSComplexity.org²⁰, kratko.js²¹ and jsmeter²² (not to be confused with the Microsoft Research project²³), and general source code analysis software that support JS such as Understand²⁴, SonarQube²⁵ and Yasca²⁶. There seems to be no refactoring tool for SML widely available.

¹³<https://github.com/nrnrr/SML-Lint>

¹⁴<http://jslint.codeplex.com/>

¹⁵<http://www.jetbrains.com/resharper/>

¹⁶<http://www.devexpress.com/Products/CodeRush/>

¹⁷<http://www.jetbrains.com/webstorm/features>

¹⁸http://www.jetbrains.com/editors/javascript_editor.jsp?ide=idea

¹⁹<https://netbeans.org/kb/docs/ide/javascript-editor.html>

²⁰<https://github.com/philbooth/complexity-report>

²¹<https://github.com/kangax/kratko.js>

²²<https://code.google.com/p/jsmeter/>

²³<http://research.microsoft.com/en-us/projects/jsmeter/>

²⁴<http://www.scitools.com>

²⁵<http://docs.codehaus.org/display/SONAR/Plugin+Library>

²⁶<http://www.scovetta.com/yasca.html>

4.6 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 [45, ch. 2]. A mock is different from a stub in that it has pre-programmed expectations and built-in behaviour verification [7, p. 453].

JS has no notion of interfaces. This makes stubbing and mocking harder, reduces the ability to write tests for an interface before there is an implementation and impedes the ability to write modular or testable code. This is both a reason why testing JS is hard, and a reason *for* doing it, since testing can compensate for the lack of interfaces by enforcing modularity.

In JS, tools for stubbing can be superfluous because of the possibility 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 could be avoided by using a stubbing tool, but the benefits include fewer dependencies and sometimes more readable code, as mentioned in section 5.2.3 [2, questions 54-55]. However, bear in mind that since JS has no notion of interfaces, it is easy to make the mistake of using the wrong method name or argument order when stubbing a function manually [7, p. 471].

Typical cases for using a stubbing or mocking framework rather than VanillaJS include when an assertion framework has support for it, as is the case for Jasmine, when there is 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 [41, 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.

In SML, stubbing and mocking can be problematic because of its immutability and lexical scoping. There are situations where replacing a definition with another is technically possible, such as if a dependency is located in another file (a fake version of the definitions could be imported in that file instead) or in the rare case where a mutable reference is used, but in most practical applications there is currently no way of stubbing or mocking in SML. Perhaps it would be possible to modify an SML implementation to allow it, or do something rash such as allowing tests to temporarily modify the source code containing the system under test (SUT), but that is somewhat far-fetched and error prone.

4.7 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. [47][48]

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 Build Tools

Build programs play an important role in automating testing, and they are often integrated with version control systems. 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 command-line interface (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, Grunt²⁷ 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 evergreen²⁸ that allows it to run Jasmine unit tests. [49][12, question 6]

SML has several compilers such as mosmlc and the MLton whole-program optimizer and a large number of interpreters, but they typically have to be manually integrated with other build tools such as GNU Make.

²⁷<http://gruntjs.com/>

²⁸<https://github.com/jnicklas/evergreen>

5 Testability – Real World Experiences

Adding tests for code that was written without testing in mind is challenging [7, p. 18]. In this section a case study for doing so in JS is described, in order to highlight problems and how some of them can be solved. After a short description of how the project was selected and set up, observations regarding the code and the tools used to test it are highlighted, followed by a more general discussion on what to think of when testing an existing application.

5.1 The Asteroids HTML5 Canvas Game

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 4), 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.

5.1.1 Getting Started

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 6 and 8). 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 [10, p. 17], but it turned out 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* [7] 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 5.

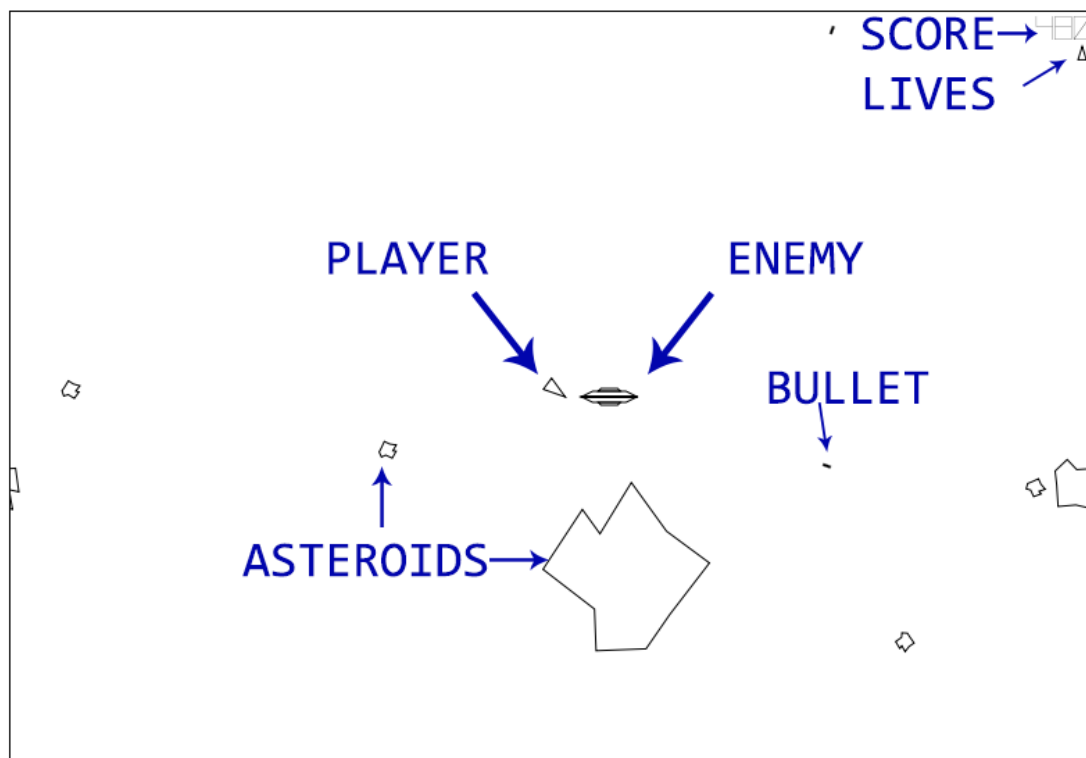


Figure 4: The asteroids HTML5 canvas game: here the player ship is about to collide with an enemy ship which will destroy them both (unlike what happens when asteroids collide, an example of desirable behaviour to test for)

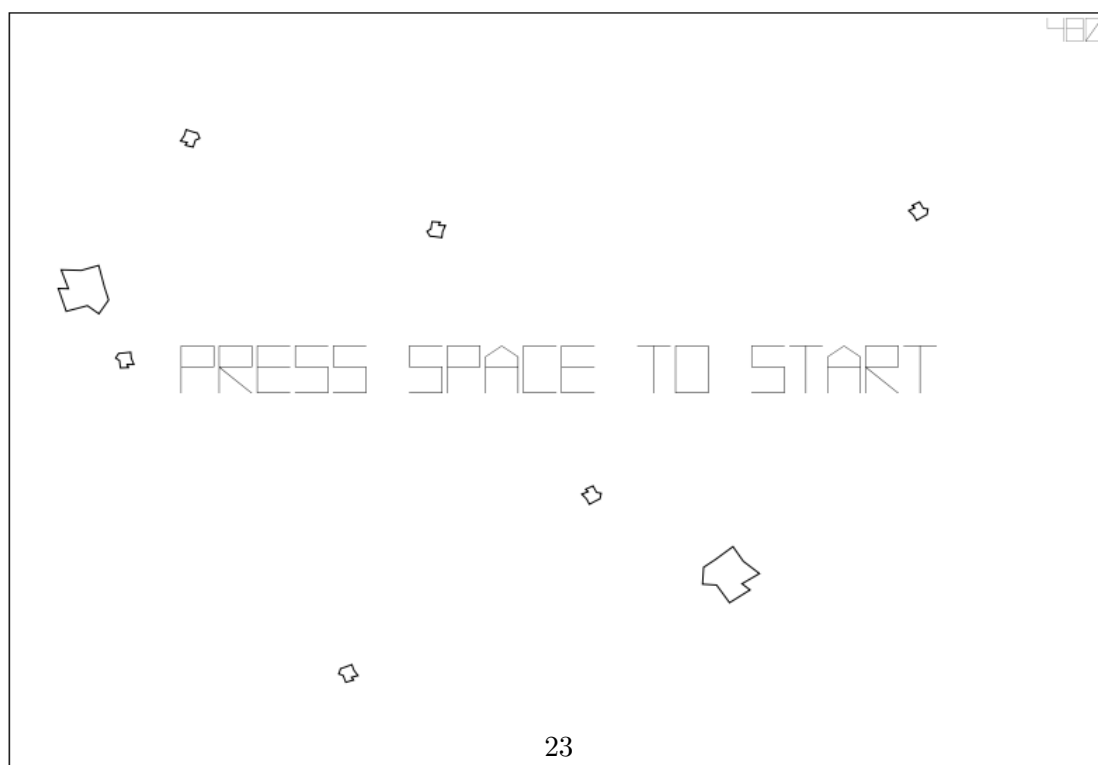
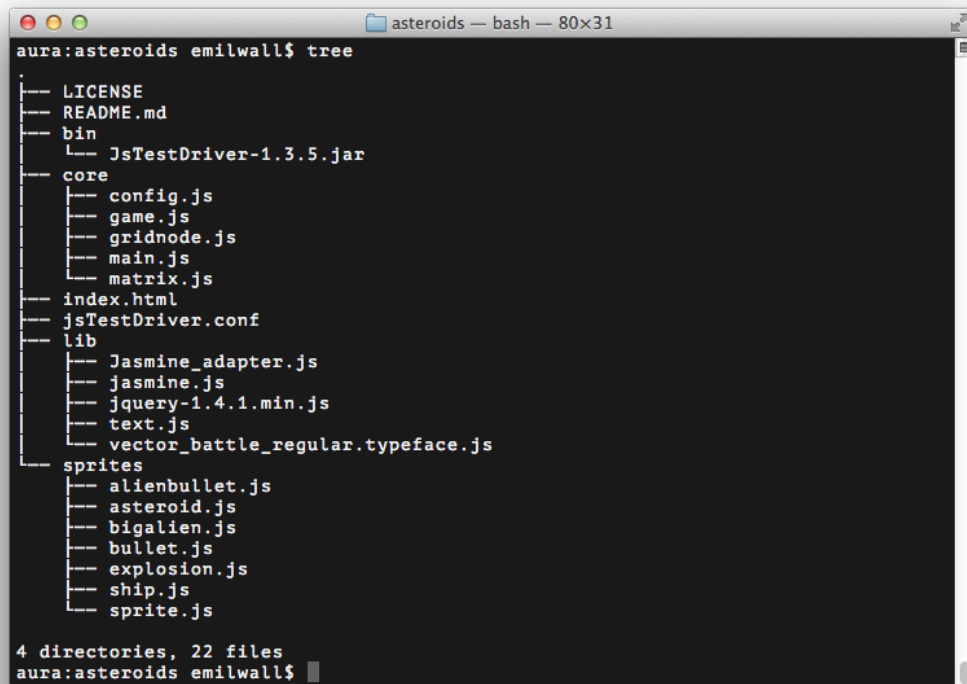


Figure 6: After a game over screen has been displayed for one second, the user sees this screen with score of previous game and the ability to start a new game

A terminal window titled 'asteroids — bash — 80x31' showing the output of the 'tree' command. The command is 'aura:asteroids emilwall\$ tree'. The output shows a directory structure with files like LICENSE, README.md, bin/JsTestDriver-1.3.5.jar, core/config.js, game.js, gridnode.js, main.js, matrix.js, index.html, jsTestDriver.conf, lib/Jasmine_adapter.js, jasmine.js, jquery-1.4.1.min.js, text.js, vector_battle_regular.typeface.js, and sprites/alienbullet.js, asteroid.js, bigalien.js, bullet.js, explosion.js, ship.js, and sprite.js. At the bottom, it says '4 directories, 22 files' and 'aura:asteroids emilwall\$' with a cursor.

```
aura:asteroids emilwall$ tree
.
├── LICENSE
├── README.md
├── bin
│   └── JsTestDriver-1.3.5.jar
├── core
│   ├── config.js
│   ├── game.js
│   ├── gridnode.js
│   ├── main.js
│   └── matrix.js
├── index.html
├── jsTestDriver.conf
├── lib
│   ├── Jasmine_adapter.js
│   ├── jasmine.js
│   ├── jquery-1.4.1.min.js
│   ├── text.js
│   └── vector_battle_regular.typeface.js
└── sprites
    ├── alienbullet.js
    ├── asteroid.js
    ├── bigalien.js
    ├── bullet.js
    ├── explosion.js
    ├── ship.js
    └── sprite.js

4 directories, 22 files
aura:asteroids emilwall$
```

Figure 5: 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 `core-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, containing just a single line:

```
var asteroids = {}
```

This line ensured that any old definitions in the `asteroids` namespace were overwritten before reading the files anew. This should not really be necessary, but was done in order to avoid emptying the cache in between each test run with JsTestDriver. An updated version of the application can be found at <https://github.com/emilwall/HTML5-Asteroids>.

5.1.2 Attempts and 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:

```
1 $(function () {
2   var canvas = $("#canvas");
```

```

3  Game.canvasWidth  = canvas.width();
4  Game.canvasHeight = canvas.height();
5
6  var context = canvas[0].getContext("2d");
7
8  ... // Omitted for brevity
9
10 window.requestAnimFrame = (function () {
11     return window.requestAnimationFrame      ||
12            window.webkitRequestAnimationFrame ||
13            window.mozRequestAnimationFrame  ||
14            window.oRequestAnimationFrame    ||
15            window.msRequestAnimationFrame   ||
16            function (/* function */ callback, /* DOMElement */ element)
17     {
18         window.setTimeout(callback, 1000 / 60);
19     });
20
21 var mainLoop = function () {
22     context.clearRect(0, 0, Game.canvasWidth, Game.canvasHeight);
23
24     ... // Omitted for brevity
25
26 };
27
28 mainLoop();
29
30 $(window).keydown(function (e) {
31     switch (KEY_CODES[e.keyCode]) {
32         case 'f': // show framerate
33             showFramerate = !showFramerate;
34             break;
35         case 'p': // pause
36             paused = !paused;
37             if (!paused) {
38                 // start up again
39                 lastFrame = Date.now();
40                 mainLoop();
41             }
42             break;
43         case 'm': // mute
44             SFX.muted = !SFX.muted;
45             break;
46     }
47 });
48 });

```

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 7) 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.

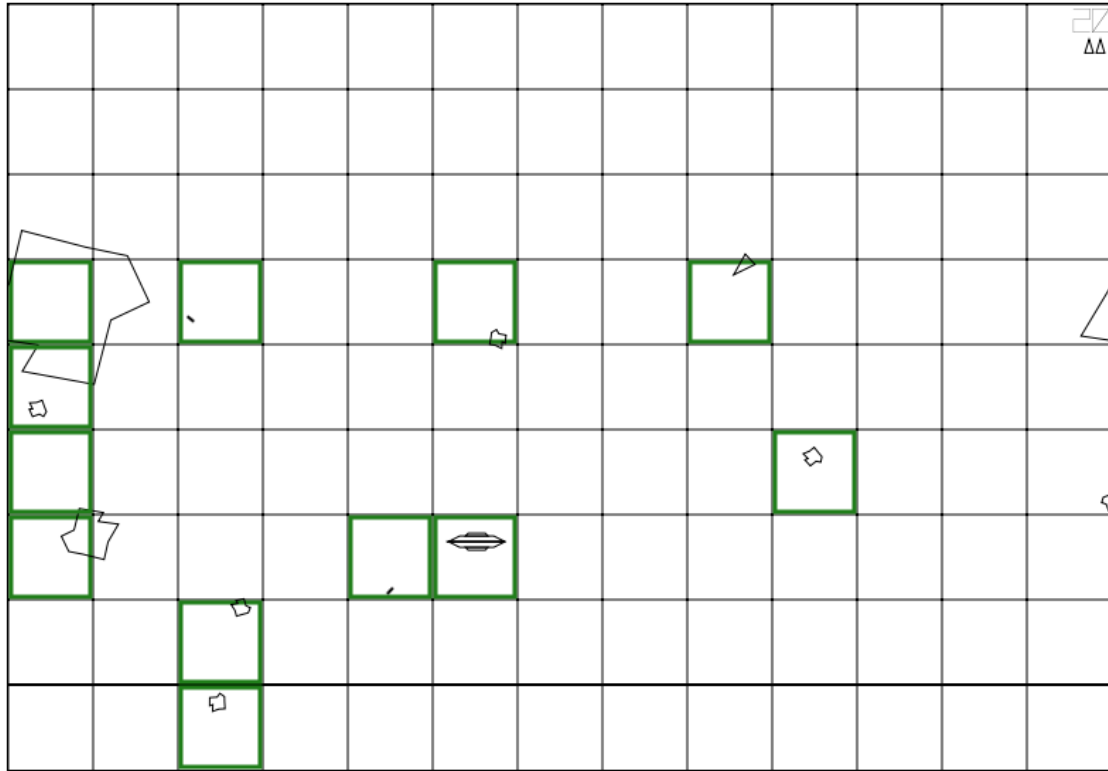


Figure 7: Draw grid function in action, highlighting cells with sprites

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 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

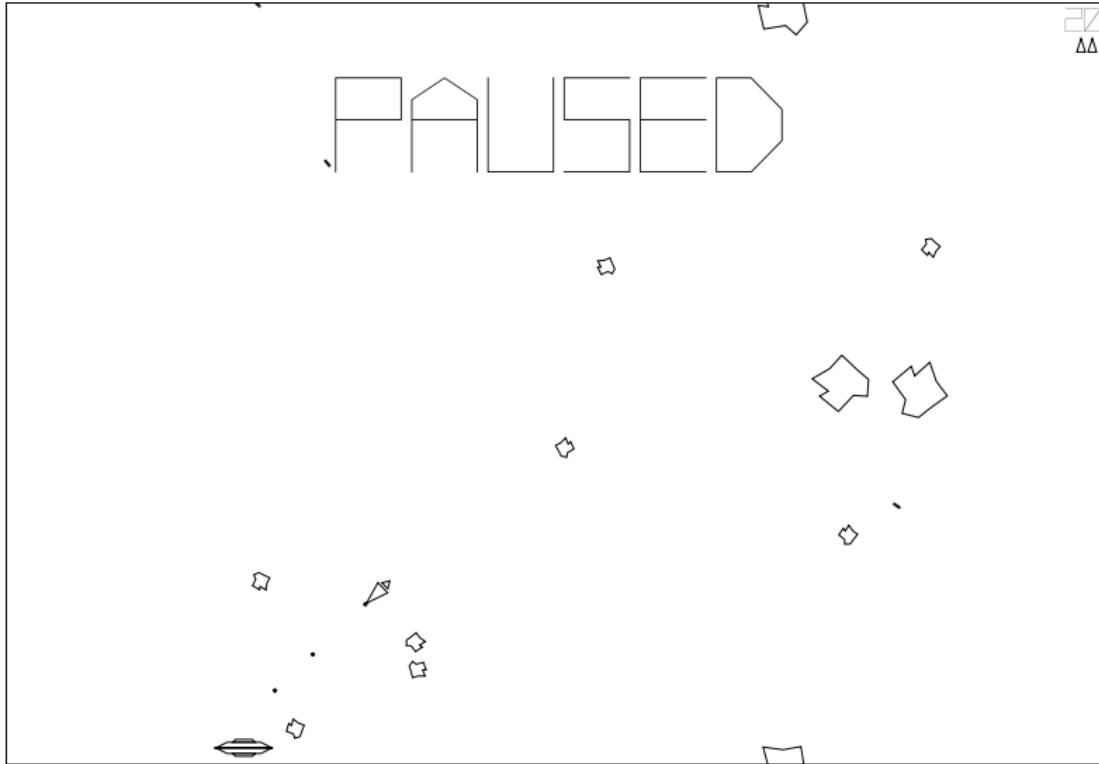


Figure 8: Game paused while accelerating and shooting towards an enemy ship

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 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`²⁹, 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 construct fake objects manually for the canvas in unit tests as well. A similar useful module is `Resemble.js`³⁰, that has advanced image comparison features that are useful together with `PhantomCSS`³¹.

²⁹<https://github.com/HumbleSoftware/js-imagediff>

³⁰<https://github.com/Huddle/Resemble.js>

³¹<https://github.com/Huddle/PhantomCSS>

5.1.3 Design Considerations

Because there was no documentation available and the application was sufficiently small, *test plans* were written in the form of source code comments about which tests that could be written, in the spec files for each class. Arguably, it could have been beneficial to write actual specs but with empty function bodys, but that might would have given an impression that they were passing rather than waiting to be written.

Each function was analysed with respect to 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 a comment is inevitably slightly faster to write than a complete test, since it makes up for fewer lines of code and the implementer avoids the risk of getting stuck with details about how to write the test.

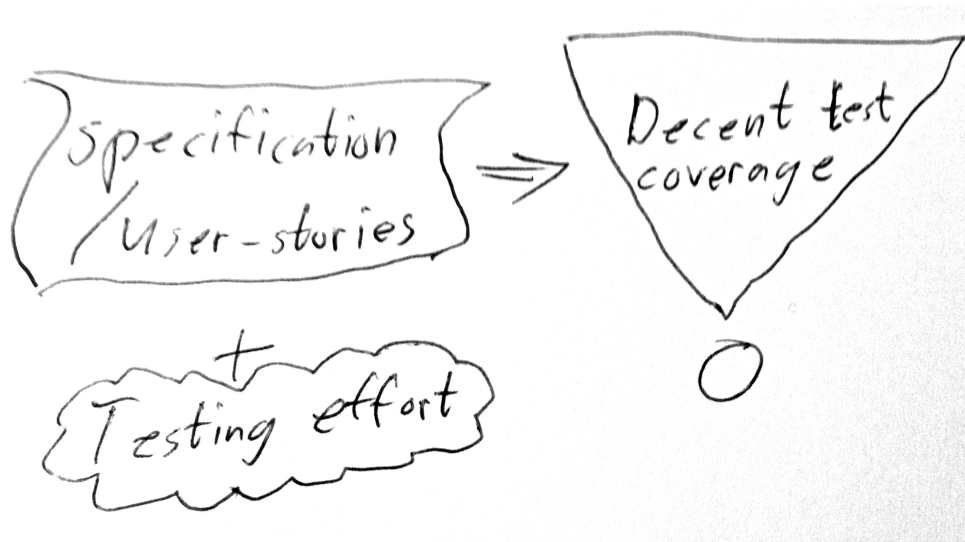


Figure 9: Tests should be based on desirable behaviour

Regardless of which tools are used, it is important to remember that testing strategies matter. For instance, coverage should not be a goal in itself because the risk of errors can never be eliminated that way [2, question 28], and I personally think that too much focus on coverage increases the risk of tests becoming more implementation specific than they would if instead focusing on what the application should do from a high level perspective. When writing tests for the finite state machine (FSM) in the asteroids.Game object of the asteroids application, it took almost 100 lines of test code to achieve clause coverage [50, p. 106] for 18 lines of production code (asteroids.Game.FSM.start), as can be seen in commit 61713c of <https://github.com/emilwall/HTML5-Asteroids>.

Even if attention is paid to what would be most valuable to test, modifying the application to enable these tests will often have an impact on the design. 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, one of two things apply: either the code should not be tested in that way [16, question 8][2, question 27], or the responsibilities of that code should be revised [41, question 34]. Perhaps the code is doing more than one thing, and it is the secondary thing that is hard to test. No wonder then that it feels wrong to expose certain aspects of it, they might not be the primary purpose of that code segment, just an implementation detail that either doesn't matter or should be moved into a separate location where it can be tested. 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 logic of the application.

My opinion after performing there experiments is that global state should not be modified by constructors. 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 their prototype), which meant that 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 and design.

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. Presumably, there would have been less outer dependencies to locate and isolate in the tests, or less work to instantiate the objects. [12, question 26]

5.1.4 GUI Testing Considered

As mentioned in section 4.7, Selenium is a popular tool for browser automation. Sometimes it is the only way to test a JS of an application without rewriting the code [41, question 43] but then tests tend to be brittle and provide little value, according to my own experiences and people I've talked with (see section 5.3.4 for further discussion). In general, since Selenium tests take such time to run they should only cover the most basic functionality in a smoke test fashion [41, questions 16-17][12, 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. In general, this has has potential of being 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 which aroused enthusiasm during the interview with Edelstam [2, questions 42-43], 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. In the case of the asteroids application, it would not work at all, since it does not record mouse movements but merely tracks which elements are interacted with and how.

As a consequence of being unable to extract all code from `main.js` into testable classes, selenium tests were indeed considered as a last resort of testing its behaviour. This would be a reasonably 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, provided the issues with non-determinism would somehow be dealt with. Selenium just recently included the most basic support for testing canvas elements [48, p. 165-166]. However, by the time this was considered there was no time for more experimental work with the application.

Using Mocha with Zombie.js (or Phantom.js) as described in *Using Node.js for UI Testing* [26] was also considered, but deemed as infeasible because that would require more fine grained control over the interaction with the canvas element than what is possible with these tools.

5.2 Tool Issues

Selecting which tools and frameworks to use is frequently done fast, based on availability or what people have prior experience with. The decision is nevertheless important, regardless of whether a project has just started or if it is under a maintenance phase. Tools can facilitate testing, so that people are not so deterred by the initial effort required to get started. There are many examples of frameworks and tools, such as Yeoman and Brunch, that can help in quickly getting started with a project, and structure it so that testing becomes easier [2, questions 11]. In this subsection the discussion will mainly be about tools used to write and run the actual tests.

5.2.1 JsTestDriver Evaluation

When setting up JsTestDriver³² (JSTD) with the Jasmine adapter there are pitfalls in which version is used. 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, an older version of Jasmine (such as 1.0.1 or 1.1.0) is needed or the adapter has to be modified for compatibility. 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 prevents placing the test driver separate from the SUT rather than in a parent folder, or referencing another framework such as Jasmine if it is placed in another directory.

One has to be careful when adding code to `beforeEach`, `setUp` and similar constructs in testing frameworks. If it fails, most testing frameworks report that all dependent tests fail, but the result can be unpredictable. When using Jasmine with JsTestDriver, there is no guarantee that all the supposedly affected tests fail despite `beforeEach` causing a failure, and subsequent test runs may produce false negatives. This is likely due to optimisations in the test driver or that the SUT contains globally defined objects (rather than constructors). `game.js` contained such a globally defined object and its tests commonly failed unexpectedly after some other test had failed, and continued to

³²<https://code.google.com/p/js-test-driver/>

do so in subsequent runs even after the failing test has been fixed. Restarting the test driver and emptying the cache in the captured browser temporarily solved the problem, but was time demanding.

When refreshing and clearing the cache of a captured browser, one might have to wait for a couple of seconds before running any tests or else the browser can hang and the server have to be restarted. To make matters worse, definitions from previous test runs might 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 this is far from trivial so a beginner will easily get frustrated with these issues, since the browser has to be refreshed quite frequently in the process of finding out. The fault is both due to the tool and due to the lack of discipline of the person writing the tests.

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, question 15].

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 any indication when these problems occur, especially when there is false positives. This was why `reset.js` was needed (see section 5.1.1).

If there is a syntax error in a test, JSTD 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 )
```

A developer 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 to pay close attention to the terminal output and check that the correct number of tests were run rather than just checking that there were no failures. This is impractical when running the tests in a CI build because the build screen would probably 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 [51].

Test run	Safari	PhantomJS	Firefox	Opera	Chrome
1	190	246	472	524	486
2	173	223	513	541	539
3	180	233	460	465	492
4	231	257	413	459	551
5	177	228	467	554	564
6	182	262	438	475	493
7	179	239	452	481	617
8	175	281	492	541	571
avg	185.9	246.1	463.4	505.0	539.1

Table 1: Time in ms to run 287 tests with JsTestDriver in headless and non-headless browsers. Average for PhantomJS was 246 ms, whereas Safari ran the tests in 186 ms on average – 24 % faster. The other non-headless browsers required roughly twice as long time as PhantomJS.

5.2.2 PhantomJS

The increased maturity of tools such as PhantomJS and Mocha can be truly helpful to get a smooth workflow when adding tests to already existing projects [2, questions 11-12 and 20]. An important difference between using a headless browser such as PhantomJS compared to running tests in ordinary browsers, using for instance JsTestDriver, is that a headless browser provides no information about how the code performs on different JS implementations. Reasons to do so anyway 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]

Error reports are useful feedback that tests provide. The quality of these reports vary depending on which testing frameworks are used and how the tests are writted. When using PhantomJS to run tests, some test failures requires running the tests in a browser in order to get good error reports. [2, question 12]

The unit tests for the asteroids application had an average execution time of roughly 1 ms per test, so there was little to gain from using a headless browser. However, it was done anyway for the sake of research, using <https://github.com/larrymyers/js-test-driver-phantomjs>. The results were somewhat surprising – the execution time increased by an average of 32 % (see table 1), rather than decreasing as would be expected.

The tests were run on OS X 10.9, using Safari 7.0 (537.71), PhantomJS 1.9.2 (534.34, installed using npm), Firefox 25.0.1, Opera 18.0.1284.63 and Chrome 31.0.1650.63. Nightly builds of Firefox (28.0a1) and Chrome (33.0.1734.0 canary) were also tested, with average runtimes of 718.5 ms (deterioration) and 482.3 ms (improvement), respectively. The tests were run with a single browser captured at a time, because the running times were longer and more unpredictable when capturing several browsers at once.

Since the browsers are not restarted between runs, the JS engine of PhantomJS was somewhat outdated compared to Safari (534.34 versus 537.71) and the tests contained

no dom manipulation that would trigger rendering activities in a non-headless browser, the Safari could perform well despite not being headless. Some efforts were made to run the tests in PhantomJS directly to eliminate the overhead of the JsTestDriver integration (using <https://github.com/jcarver989/phantom-jasmine>), but either no tests were run or uninformative error messages were given, although the same test fixture worked in Safari (with a negligible difference in execution time compared to running the tests with JsTestDriver).

5.2.3 Sinon.JS

Proper stubbing of dependencies can be a daunting task, and is closely connected to mocking and dependency injection. In JS, dependency injection is harder than in for instance C or Java because of the absence of formal class interfaces. The sinon.JS framework does make it easier to stub compared to if done manually (which on the other hand is exceptionally simple to do with JS) but there are still 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 and making them less readable.

Deficiencies such as these are important to note because they pose potential reasons to why JS developers don't test their code. I think that if using the tools and frameworks is perceived as cumbersome and demanding, fewer will, and those who do are likely to consider it worth doing so in fewer cases. However, there are also some notable advantages of using a framework like Sinon.JS, for instance it can provide a fake server for handling web requests [52].

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]

Having spent many hours debugging, a thorough check was done that no test depended on another test or part of the application that is not supposed to be tested by a specific test. The purpose was to ensure that the tests were truly unit tests. In order to do this, a copy of the application repository was created and all files in the copy except for one test and the corresponding part of the application were deleted. Then a JSTD server was configured with a browser to run only that test, and the process was repeated 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.

5.3 Lessons Learned

In this subsection, problems that are common to not just the asteroids application and the selected set of tools and frameworks, are highlighted. There are typically ways to work around at least some of these problems.

5.3.1 General Issues With Adding Tests to an Existing Application

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 writing the tests based on details about the implementation rather than basing them on what the system should do from an interface point of view. These kind of tests tend to be hard to understand and usually have to be changed whenever an improvement of the implementation is considered, but are unequivocally hard to avoid. This was both an experience from working with the asteroids application, and an opinion from the interviews [6, question 38].

A large number of tests is not necessarily a good thing either, since 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]

TDD can be applied even on an application that previously has no tests, by writing tests that document any changes made to the application, and isolates every bug that is found, a strategy proposed by Edelstam [2, question 30]. One should be careful not to break existing functionality, but if that happens then it can be useful to prevent the bug from reappearing by writing a test for it. This is best done before fixing the bug to watch the test fail, increasing the chance that the test is correct and ensuring that the same bug is not introduced again without anyone noticing. Writing tests for an application in its existing form before changing it will increase understanding of how it works and reduce risk of breaking existing functionality. However, sometimes code needs to be refactored in order to make it testable, as was described in section 5.1.2.

Traditionally, coverage criteria has been a central concept in software testing and is still today in many organisations because it is often relatively easy to measure. When doing TDD however, the need for thinking in terms of coverage is reduced as every small addition of functionality is tested beforehand. As Rovegård said, a decent code coverage comes for free [6, question 34]. The goal is generally not 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 not to test it more carefully to compensate, but to refactor in small steps and test new components separately [41, question 34], 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. There are various tools available for ensuring that

relevant parts of an application are exercised by tests and measure cyclomatic complexity to determine where it is most relevant to design tests based on edge cases and abnormal use. As a tester, it tends to pay 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 [50].

5.3.2 Stubbing vs Refactoring

In order to make code testable, the architecture may need to change. Sometimes introducing a new object or changing how some dependency is handled suffices. The important thing is to not continue in a direction that will cause the codebase to deteriorate over time [41, question 34]. If a lot of work goes into mocking dependencies (see section 4.6) 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. [41, question 42]

When a function is defined within a constructor it is hard to stub it unless there is an object created by the constructor available. In some cases the SUT creates an instance by itself and then there are few options except for stubbing the entire constructor, which produces a lot of code in the tests, or changing the SUT to increase testability, for instance by having the instance passed as an argument or defining the functions to stub on the prototype of the constructor instead of in the constructor. Often it is possible to increase testability without negative impacts on readability, performance, etc. of the SUT, but not always. Regardless, this requires a skilled programmer and effort is spent on achieving testability rather than implementing functionality which may feel unsatisfactory.

Sometimes it can be hard to know whether or not to stub library functions, such as those supplied by jQuery that have no side-effects, or trust that they behave as expected and steer the control flow via their input and the global state instead. 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. Aside from these technical considerations of how decisions regarding stubbing affects how the test works, there are readability benefits in reducing the number of test setup code lines on one hand and making the test context explicit on the other.

When an application is tightly coupled, stubbing easily becomes a daunting task. The decision is between keeping the level of stubbing to a minimum, refactor the code to reduce the amount of calls that needs to be stubbed, or engage in voluminous stubbing. 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 might not be worthwhile unless it significantly improves the design. Stubbing all dependencies might result in an impractical amount of setup code with complicated testing. A way to avoid this dilemma is TDD, which encourages and supports more aggressive refactoring. This tradeoff had

to be made when tests for classes that depended on the Sprite class were written. The Ship class used Sprite both for its “exhaust” attribute and for its prototype. A common principle is that constructors should not modify or depend upon global state. Luckily, the Sprite constructor adhered to that principle, so in this case not stubbing the Sprite class before parsing the Ship class was acceptable. However, all calls to methods defined in Sprite were stubbed, since they were tested separately.

Sometimes during the process of adding tests to the asteroids application, the tests accidentally ended up with incomplete stubbing. Detecting this sort of test smell was non-trivial, because there was no automated way of ensuring that a test accessed only a certain part of the application. It was not an option to manually organise the files to ensure that just the appropriated code was loaded, since attempting to stub a function which is not defined produces an exception in sinon.JS, in order to avoid making typing errors. This could be circumvented by manual stubbing, but abandoning sinon.JS for that reason felt inconvenient, especially considering that the separation from the code that should not be executed by the tests had to be done manually, so instead I opted towards simply being more careful not to miss any calls that should be stubbed. A semi-automated approach might be possible, using coverage reports to inspect that only the desired code is executed when a certain test is run, but that was not investigated.

A programmer has to be very methodical, principled and meticulous not to miss some detail and write an accidental integration test. Such mistakes results in 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 SUT.

5.3.3 Deciding When and What to Test

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, although the interviews showed that there is disagreement on this matter [12, question 21][53, question 5]. 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]

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 [41, questions 6-7]. It is a common belief that it is more important to 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, or better: test that the problem that the class is intended to solve is actually solved. Whether or not that is dependent on the inheritance patterns is mainly relevant for stubbing considerations – the prototype of an object may be replaced in tests to check that there are no unexpected dependencies.

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. A specification can be constructed, in order to be systematic about what tests to write. This can be done top-down by looking at user stories (if there are any), consulting a user experience expert, talking with the product owner and the users (if any) or by identifying features to test through normal use. 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, as was mentioned in section 5.1.3, which possibly contributed to more implementation specific tests. In the interviews, Ahnve proposed that unit tests should be implemented from a business value point of view (top-down) to avoid this problem [12, question 20].

Something that came up during the interview with Edelstam was that when a certain section of code feels hard to test, it is likely that all tests for that section 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], as discussed in the beginning of section 5.3.2. When writing the tests for the asteroids application, it was deliberately decided to write tests even when it felt hard, or when it felt like it provided little value, to see whether this made the application harder to test or modify later.

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. [41, 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 code is subject to change, so according to Stenmark a compromise by writing a few simple tests for that code may pay off. [41, 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. [12, questions 17-18]

5.3.4 A Related Experience

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 logic was too tightly coupled with the GUI. This meant 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, which he considered unstable (tests failing for the wrong reasons) and slow. He also sought advice regarding 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 BDD approach and at least one had used Karma (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 shared and appreciated. The consensus seemed to be that in general, testing JS is good if done right, but also difficult. Had I known about PhantomCSS³³ at the time, I would have recommended it.

6 Problems in JavaScript Testing

Considering the variety of available frameworks, one may be deceived into thinking that people do not test their JS because they are lazy or uninformed. However, there are respectable obstacles for TDD of JS, both in the process of fitting testing frameworks into an application and in writing testable JS. This section covers how to test asynchronous events, DOM manipulation, form validation, the GUI and APIs, in JS.

6.1 Asynchronous Events

JS is often loaded asynchronously in browsers to improve performance in page loads and minimise interference with the display of the page. Another asynchronous aspect of JS are events such as AJAX calls and timer dependent functions. Testing asynchronous events can be hard because execution order 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 different scenarios can be tested and thereby provide a good feel of how the code will behave when run asynchronously. When using waiting conditions, the tests may become nondeterministic and timeouts need to be set for when tests should fail if a condition is not met. Too long timeouts will lead to very long execution times whereas if they are not long enough, there will be false negatives.

³³<https://github.com/Huddle/PhantomCSS>

Most unit testing frameworks supports asynchronous testing through waiting conditions or similar constructs, with some differences in syntax and how it works. There are guides for how to use these aspects of the frameworks in both literature and blogs, for many major testing frameworks: Jasmine [23, p. 35][25, p. 45-49], Mocha [26, p. 59-65], YUI test (only explicit wait) [25, p. 78-79] and qUnit [25, p. 114-116]. There are also a large number of relevant blog posts for asynchronous JS testing ^{34 35 36 37 38}.

Test-driven JavaScript Development claims that JsTestDriver has no built in support for asynchronous testing so it proposes other approaches [7, p. 247-387], but the support was added to the development branch the same year (2010) and is now part of the release: *JavaScript Unit Testing* provides an example of using the `AsyncTestCase` object in JsTestDriver to extend test methods with a queue parameter [25, p. 143-145].

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 if there are few dependencies between the asynchronous code and they share few or no resources, the risk for data races and similar issues is smaller. Therefore, an important part in achieving testability of asynchronous code is to strive for such designs.

The issue of testing asynchronous code is not unique to JS, for example Erlang message passing is also asynchronous, and has been used in systems with high demands on stability. One may ask how Erlang programmers test their code, but answering that question and relating the answers to testing of JS is such a large project that it could make up for an interesting study of its own. Many other languages with support for multiple threads or processes arguably have this issue as well.

6.2 DOM Manipulation

During the interview with Ahnve, he proposed that 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 2010³⁹ 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. [12, question 4]

DOM manipulating code can be tested if necessary, but there are many implementation differences depending on which browser or testing environment the tests are run in which means that the tests have to run in different browser implementations to avoid a false sense of security (on the other hand, this is one of the reasons why one might WANT

³⁴<http://lostechies.com/derickbailey/2012/08/18/jasmine-async-making-asynchronous-testing-with-jasmine-su>

³⁵<http://lostechies.com/derickbailey/2012/08/17/asynchronous-unit-tests-with-mocha-promises-and-winjs/>

³⁶<http://www.htmlgoodies.com/beyond/javascript/test-asynchronous-methods-using-the-jasmine-runs-and-waitf>
html#fbid=eILq_Eu-N_I

³⁷<http://www.htmlgoodies.com/beyond/javascript/how-mocha-makes-testing-asynchronous-javascript-processes->
html

³⁸<http://blog.caplin.com/2012/01/17/testing-asynchronous-javascript-with-jasmine/>

³⁹Agile Processes in Software Engineering and Extreme Programming 11th International Conference, XP 2010, Trondheim, Norway, June 1-4, 2010, Proceedings

to test the DOM manipulations) and, as both Stenmark and Wendt mentioned in their emails (see Appendix), that it can be hard to keep test fixtures fast and up to date [41][53].

There are free cloud-based tools such as TestSwarm that are designed to help with running the tests in many different browsers [54], or it can be done manually using JsTestDriver, which also has built in support for setting up test fixture DOMs. This is covered in *Test-driven JavaScript Development*, which also shows how it can help to identify reusable components in views [7, p. 389-435]. The qUnit testing framework is also well suited for testing DOM manipulation⁴⁰, much due to its origins in the jQuery project.

Aside from unit testing, Selenium drivers can be used to check almost anything, but beware that such tests easily become slow so even for a medium sized application it may not be possible to run the tests more than a couple of times per day. Tests can be made to run faster if they run in a headless browser such as PhantomJS, which is supported by Selenium, or by using more specialised tools such as CasperJS which can produce very similar results as Selenium but executes faster and is more easily controlled using JS [27, p. 142-146].

6.3 Form Validation

A common case of user interface testing is form validation. According to Edelstam, a TDD extremist might argue that there should be tests for that the form exists, has a working submit button and other boilerplate things, but that is mostly irrelevant unless the form is written in a way that is unique or new 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]

Validation of forms is 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 in practise there is often a need to detect unexpected behaviour in the client side validation frameworks.

Tools for client side form validation include Parsley⁴¹, validate.js⁴² and the jQuery Validation Plugin⁴³. It is also possible to use the built in support in HTML5 (which however will not work in old browsers) or implement custom validation that is called with `onsubmit` or `onblur`. However, form validations should not just be implemented, they should be tested too.

⁴⁰<http://qunitjs.com/intro/#testing-the-dom-manipulation>

⁴¹<http://parsleyjs.org>, <https://github.com/guillaumepotier/Parsley.js>

⁴²<http://rickharrison.github.io/validate.js/>, <https://github.com/rickharrison/validate.js>

⁴³<http://jqueryvalidation.org>, <https://github.com/jzaefferer/jquery-validation>

It is typically a good idea to test for both type I and type II validation errors (false positives and false negatives). If forms are wrongly prevented from being submitted (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.

False positives in form validation can occur when input is interpreted wrongly. Proper testing of the form validation could detect this at an earlier stage, thereby avoiding loss of potential customers. A real life example of this is when I submitted my personal number to register a domain and the validation produced an error message falsely claiming that that I was not old enough. I alerted the service providers on twitter: *I get a validation error for my personal number, "Not old enough to buy .SE domain. Min age: 18" I'm born 890101...*⁴⁴. They immediately fixed the error, but presumably most people would choose another service provider in a case like this.

A similar bug was observed 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 a cellphone identification service had to be used in order to log in. Such a thing would not happen if the bank had automated cross browser testing of the forms. [12, question 38]

So how can form validation be tested? Selenium is one option, to enter form data and submit and then check that the proper validation error was printed out at the right location. A benefit with this approach is that a user interaction can be closely imitated. A problem is that if there are many validations to test, these tests tend to take several minutes, as was the case in my first project at Valtech. This is a general problem for tests that involve the DOM [41, questions 21-22].

Depending on which framework is used, there might be different options for testing. Just testing the validation rules without involving the forms at all can be of great value, for example if the validation is specified using regular expressions, hopefully they can be exposed for isolated testing to avoid testing all the validation in slow integration tests. This will not detect runtime errors or similar for the actual validation, but for complex validation such as testing which strings match a certain regular expression, it is typically better to have fast unit tests, which can be complemented by a smaller number of smoke tests for the integration.

A common case is to use another programming language than JS for the back-end, specifying the validation rules in that language and letting for instance a jQuery plugin generate validations for the front-end. Here regular expressions and the like can hopefully be put as publicly available variables and accessed directly in tests, whereas the plugin will have to be tested separately, using perhaps a single test that just checks for existence of validation.

⁴⁴<https://twitter.com/erif89/status/378536014337171456>, 2013-09-13

6.4 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 Application Differences

A rather common opinion is that automatic testing of an application's visual components should be avoided. However, the value provided by GUI tests depends on how important the GUI is for the application. According to Ahnve, 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. [12, question 21]

Before deciding on whether to do automated GUI testing or not, think carefully about how important the GUI and its cross browser compatibility is. If exact appearance is irrelevant, then there is little sense in doing it. If on the other hand it would be valuable but there are concerns about the technical aspects of it, it should not be dismissed so easily, since there are visual regression testing tools available as mentioned below, which can be used solely to assert a certain web page appearance, or in combination with ordinary unit and integration tests.

During the interview with Edelstam, we discussed the problems that some people experience with testing front end interfaces and that they sometimes have to do with poor modularity. For instance, the presentation layer should not be responsible for calling large APIs. In order to improve the modularity, tools such as RequireJS can be used to handle dependencies but if each part of the application has too many or large dependencies, mocking still 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]

6.4.2 Regression Testing of GUI

GUI testing should not be confused with integration testing. Testing graphical elements in tests that depend on system integration are hard to maintain and can fail for too many reasons. Instead of executing a sequence before making assertions about graphical components, there may be benefits in 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? This could be detected 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 a test fixture might need to be updated every time a relevant change occurs, which would introduce too much maintainability burden, in which case there is little choice but to either write bad GUI tests or skip writing them altogether.

Just as one 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 points of view. In the interview with Ahnve, we touched on the importance of estimations with regards to such matters. [12, questions 40-44]

6.4.3 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 [54] 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, it will not be possible 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 compare such snapshots to spot differences with previous versions, that can be reviewed by a human. This potentially reduces the need for manual testing of the GUI, providing a good indication of which parts have changed. Some of the most popular open source tools available for visual regression testing are PhantomCSS⁴⁵, Huxley⁴⁶, Depicted⁴⁷ and Wraith⁴⁸, and there are various blog posts available that cover these techniques⁴⁹⁵⁰⁵¹⁵². Among the problems that are often mentioned in such blog posts are that small changes can lead to many failing tests, so the baseline images all have to be updated and anti-aliasing of different operating systems cause tests that pass on some computers to fail on others.

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

⁴⁵<https://github.com/Huddle/PhantomCSS>

⁴⁶<https://github.com/facebook/huxley>

⁴⁷<https://github.com/bslatkin/dpxdt>

⁴⁸<https://github.com/BBC-News/wraith>

⁴⁹<http://piwik.org/blog/2013/10/our-latest-improvement-to-qa-screenshot-testing/>

⁵⁰<http://abhishek-tiwari.com/post/visual-regression>

⁵¹<http://csste.st/techniques/image-diff.html>

⁵²<http://codeutopia.net/blog/2014/02/05/tips-for-taking-screenshots-with-phantomjs-casperjs/>

by nature have more focus on coverage and detecting runtime errors than to simulate the behaviour and expectations of users. There has been quite a lot of research in this area [13][29][32].

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 External and Internal APIs

Stenmark brought up the importance of keeping tests involving mocked external API up to date, and how end-to-end tests are seldom an alternative since they can not be run as often [41, questions 19-20]. Testing towards a true implementation typically takes too long, manipulates data that should not be touched or the API owner may not accept large amounts of traffic or requires payment for it.

Testing towards internal APIs (that exist within the same application or organisation) differs from testing calls to an external API in that the latter are often well documented, versioned and rarely change. 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. An internal 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. Edelstam argued that this is a reason to test rather than not to [2, questions 34, 36]. It is important to have integration tests that can detect when fakes need to change, and preferably a way of changing the fakes all at once, for instance by using the object mother pattern.

Introducing versioning for internal APIs is not always feasible due to the organisational overhead it introduces, 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 new versions of the API to be released before they can be updated. However, there are 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 fake so that the tests passes will hopefully result in that any incompatibility is discovered given that the same fake is used in all other parts of the testing suite involving the API. [2, question 34]

Testing an internal 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 common practice not to test private methods. If a private method contains code that can not be tested through the public API, then that method is probably too complex to be private and its functionality should be moved into somewhere else. Private methods represent implementation details, tests should be focused on behaviour and functionality, not implementation. [2, questions 62-63]

7 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 covers existing frameworks, formal verification, and introduces a new SML testing framework that was developed as a part of this thesis, inspired by insights gained from looking at popular JS testing frameworks. In section 5, we discussed how adding tests to an application written without testing in mind can introduce testability issues. Later in this section (7.4) the challenges of writing tests first is described based on a TDD approach to solving SML programming exercises in a MOOC.

7.1 Current Situation

The SML/NJ and Moscow ML language implementations⁵³⁵⁴, 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. A search on the web for “Standard ML testing framework” (with quotes, Nov 2013) generates zero results. Dig deeper, and there is 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.

Programmers 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. The YAGNI principle 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. However, from looking at the existing testing frameworks, it seems like SML developers prefer writing their own or doing without 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 (for example BDD) style of defining tests and formulating failure and success reports.

⁵³<http://www.smlnj.org/svn.html>

⁵⁴<https://github.com/kfl/mosml>

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.

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 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 break the code and is therefore outside of the scope for this thesis. A simpler approach would be to ensure that 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, one either has 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). One could also define the matchers in separate namespaces (structures), as is done in `smell-spec`⁵⁵.

QuickCheck, the Haskell predecessor of QCheck, resolves this by using the `where` keyword to specify the types of properties⁵⁶, 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 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 CADP⁵⁷, UPPAAL⁵⁸ and many more⁵⁹ (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,

⁵⁵<https://github.com/davidpdrsn/smell-spec/assertions.sml>

⁵⁶http://www.cse.chalmers.se/~rjmh/QuickCheck/manual_body.html#4

⁵⁷<http://cadp.inria.fr>

⁵⁸<http://www.uppaal.org>

⁵⁹http://en.wikipedia.org/wiki/List_of_model_checking_tools

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

Table 2: Formal verification is expensive and less useful for finding bugs than testing, but can prove correctness

	Likely	Rare
Harmless	Testing	Unimportant
Catastrophic	Testing and FV	FV

Table 3: Formal verification (FV) has a sweet spot when bugs are critical and rare, whereas testing works best when bugs are common

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 2 and 3 (pages 48 and 48) illustrate these differences between formal verification and testing. The tables are taken from lecture slides by Dr. Radek Pelánek⁶⁰, used with permission.

An interesting discussion is whether formal verification can have similar effects on modularity and interface design as TDD. Isolated units, for example short methods with few dependencies and side effects, are easier to construct proofs for, so modularity can certainly be positively affected by formal verification. It also favours simplicity, for good or ill. Whether it has benefits for interface design depends on what we mean by interface. Formal verification is unsuitable for GUIs but can help to keep APIs terse.

7.3 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 Languages⁶¹, as mentioned in section 3.2. The working title of it became DescribeSML.

7.3.1 The Framework

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

⁶⁰<http://www.fi.muni.cz/~xpelanek/IA158/slides/verification.pdf>

⁶¹<https://www.coursera.org/course/proglang>

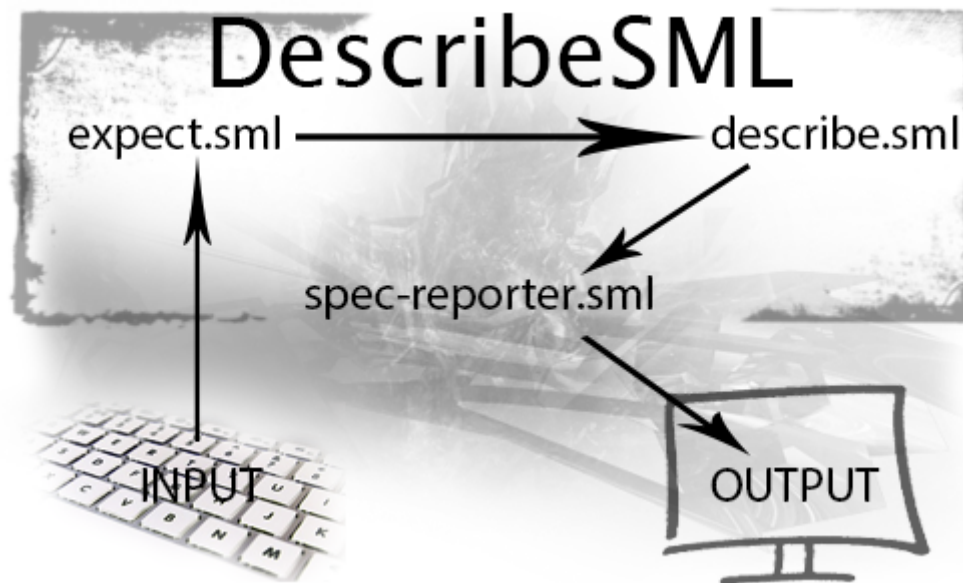


Figure 10: DescribeSML file structure and module architecture. Specs are written using Expect matchers, described and organised with Describe and should clauses, and the result of running them is reported by the SpecReporter. The arrows show how data flows through the application when the specs are executed.

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 11 (page 50) shows an example of how it could look.

The source code of the SML testing framework is supplied in the appendix. It can also be found at <https://github.com/emilwall/DescribeSML>.

7.3.2 Alternatives Considered and Discarded

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

```

1 CM.make "$/regexp-lib.cm";
2 use "spec-reporter.sml";
3 use "describe.sml";
4 val describe = Describe.describe;
5 val should = Describe.should;
6 use "expect.sml";
7 val expect = Expect.expect;
8 val toThrow = Expect.toThrow;
9 val toMatch = Expect.toMatch;
10 val toBeInt = Expect.toBeInt;
11 val toEqual = Expect.toEqual;
12
13 val foo = concat
14   o (fn [] => raise Empty | lst => map (fn _ => "abc") lst)
15   o explode;
16 datatype bar = Baz of int;
17
18 Describe.suite(describe "The SUT" [
19   describe "the function foo"
20     [should("raise Empty when called with an empty string", fn _ =>
21       expect (fn _ => foo "") toThrow Empty),
22
23       should("return string containing 'abc' otherwise", fn _ =>
24         expect (foo "string") toMatch "abc"),
25
26       should("return string three times as long as argument", fn _ =>
27         expect (size(foo "string")) toBeInt (3 * size "string"))],
28
29   describe "the datatype bar"
30     [should("have constructor Baz, taking an integer", fn _ =>
31       expect (Baz (7 + 35) : bar) toEqual (Baz 42))]]
32 ])

```

Figure 11: describe-sml-example.sml

(see figure 12, page 51). Then it would not be such a problem that all specs have single expectations, since any value can be computed once and then used in several specs to perform multiple assertions. In an imperative language like JS this could encourage bad practices and lead to unwanted dependencies but since ML is (almost) purely functional, this is not an issue. Data could actually be shared between specs with a let-in-end expression already, but a future version of DescribeSML might implement a **given** construct to clearly demonstrate the possibility, improve readability and to better handle situations where an expression is raised by the expression producing the data.

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"
```



Figure 12: Sharing data between tests does not introduce dependencies between them in a functional language such as SML

```
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 there might be a need for matchers such as `toContainStr`, `toEqualIntList` or `toEqualIntTuple5` (for instance, defined for int 5-tuples only) rather than `toContain` or `toEqual`. 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 they are not missed 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. A similar issue

is whether a `toThrow` matcher should check the exception message or just the type or name of the exception. `DescribeSML` checks if the exception message begins with the specified message, but does not require the messages to be exactly equal, in order to allow for a lenient specification of the exception to be thrown.

The naming of bindings is important in any language and a fundamental challenge of 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 practice 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 as “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””. Here, another consideration is whether the failure reports should indicate the kind of matcher used in the test. Sometimes this will make the failure report read better and reveal if the test is using the wrong matcher, but it also adds complexity to the testing framework that is probably not worth it if the testing framework becomes harder to use.

Failure messages should be displayed in the order that the specs are executed, and the specs should be executed in the order that they are defined, because even though specs should be written so that the order in which they are executed does not matter it can be confusing for the user if the execution order is not clear.

Writing a testing framework can be entertaining, inspires to write more tests, and is a

good way to hone TDD skills. However, if existing frameworks are good enough, writing a completely new one may not be justifiable towards a customer. Since open source is increasingly common, a better approach may be to improve an existing framework.

An intriguing aspect of writing a testing framework is that the framework can be used to test itself. This may sound absurd, but can work both to detect defects early and to improve the design since there is automatic feedback about which parts are important and easy to use, and which parts are missing or deficient in some aspect. An early version of DescribeSML had a `toThrow` matcher that reported which exception was expected to be thrown when no exception was thrown, but did not check that a thrown exception was the correct one. This is sometimes known as The Silent Catcher anti-pattern, although it is more commonly introduced by the user of a testing framework than by the framework itself. The `toThrow` matcher was implemented in an early stage, before the framework was used for TDD of itself, so it serves as a good representative of bugs that are easily introduced in the absence of tests.

Implementing the nested describes was perhaps the hardest part of writing DescribeSML since it had to be both recursive and simple to use. The `expect`-function was written as a one-liner: `fun expect it f = f(it)`, in order to emulate infix behaviour of the matchers, which were written as curried functions. A curried function evaluates its arguments in turn: `fun toEqual x y = if x = y then "pass" else "fail"`. An alternative could have been to define `expect` as the identity function `fun expect x = x` and then define all matchers as infix: `infix toEqual; fun op toEqual (x, y) = if x = y then "pass" else "fail"` – an advantage of this would be that expressions such as `expect 1 + 1 toEqual 2` would be valid syntax, instead of having to add parentheses around the addition.

Aside from these considerations, there were not that many issues with writing most of the matchers. A benefit of this was that the simple ones were so simple that they required virtually no testing, and could safely be used to test the more complex ones.

The first versions of DescribeSML did not have nested describes, in favour of the YAGNI principle, but the need quickly became apparent when using the framework in practice. Collecting specs in different describes would have made it harder to run them all at once and inspect the results, since the failure/success reports would get scattered instead of collected at the end of the output. Having all specs in a single describe was not desirable either, because that reduced the ability to specify their contexts. Nested describes and other ways of improving the usability are especially important when the tester has a small screen so only a small section of failure/success reports can be viewed at a time.

When a test throws an exception, the `should` function handles it and displays it as a failure (unless the `expect toThrow` matcher is used), so that the other specs can still be executed. This feature could be misused by writing tests that throws an exception instead of returning a failure message, but this would be bad practice since it might throw another exception than anticipated. The matcher `toNotThrow` is better to use in most situations, to make the test more precise and ensure that it fails for the right reason.

7.3.3 Using the Framework

When solving the first assignment, the instructions were often to “use the answer to the previous question”. In this case, a call assertion would probably have been a good idea, 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. However, automating such a stubbing mechanism in SML appears to be non-trivial because function bindings within a scope can not be changed after definition.

As DescribeSML is currently written, there is no standard way of defining several expectations in a single should. When using the framework, one might feel the need of doing so rather than writing an almost identical spec just for the sake of stating an extra expectation. This could easily be implemented by changing the type of the should function so it accepts a list of expectations instead of a single one, but that would make the syntax of the framework more complicated and it would be harder to determine what the cause of a failure is.

An interesting discussion regarding SML testing is whether current frameworks could be merged and unified into new, improved ones with collaborators joining efforts of maintaining them. Possibly, small testing frameworks such as DescribeSML could be made into plugins for others such as qCheck, in order to make them easier to use and provide additional functionality. Getting people aware that the plugin exists and finding ways of integrating might be difficult though. Since qCheck writes its output to stdout, changing the way results are reported is non-trivial. Therefore, modifying both frameworks might be necessary and then it could make more sense to merge small testing frameworks such as sml-testing and smell-spec, and keep them simple.

7.4 Test-Driven Development of Programming Assignments

Testing is arguably very useful for educational purposes, both for students when solving coding assignments and for teachers when grading. A teacher may supply students with small testing suites to build upon, or encourage testing in other ways. The value for teachers lies in that the students can work more independently and that tests can be written specifically to evaluate solutions. This can provide a fast and objective evaluation of requirements satisfaction, although style aspects and non-functional requirements are typically not possible to evaluate without manual evaluation. 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.

A factor that makes testing especially fitting for programming assignments is that it can help students to understand requirements better. When requirements are clear and unambiguous, testing will guide the student in implementing them in a structured fashion whereas when they are not, testing can help in detecting where additional information is needed or assumptions have to be made. It is better that a student finds errors through testing than being informed of them retrospectively in evaluation feedback.

A real world example of SML testing in an educational context is the Coursera MOOC

Programming Languages⁶². In that course (and others, not listed here), students are encouraged to test their code and required to hand in their tests together with their solutions. The handed in tests are not graded, and not likely to be read by the course administration to any large extent, but the requirement still has an effect on the amount of testing done by the students.

The tests that students write facilitates the coding process and gives confidence to refactor. They are also useful for the peer reviews, because they provide instant feedback on what behaviour other students might have missed. For testing to be truly valuable for a student, he or she should be careful not to consider an assignment as complete based solely on a passing test, especially if the test is provided by the teacher. A complex problem might be harder to test, but also more important.

As previously mentioned, there has been a recent increase in the number of testing frameworks available for SML. This might be explained by the need for such frameworks for MOOC students.

“I wrote sml-testing for two reasons, first because I was taking the Coursera course on Programming Languages and didn’t find anything else in which to test my code. Secondly because I wanted to explore testing concepts with the use of currying and polymorphic types.”

– Kjetil Valle, author of sml-testing

Tests were required not only for SML, but also for the two other programming languages in the course, namely Racket and Ruby. Racket has rackunit as its built in testing framework, which aims to support both novice and advanced programmers but lacks BDD support. Although feature-rich, compared to JS testing frameworks such as Jasmine and Mocha, rackunit is hard to write readable tests in. Alternative Racket testing frameworks such as test-engine does not have BDD support either, but there has been efforts to implement it: <https://github.com/damienklinnert/lisp/tree/master/bdd>

Ruby has a great testing culture so finding suitable testing tools was no problem for that part, however in the introduction of the first Ruby assignment it said:

“Because the primary challenge in this assignment is reading, understanding, and extending a program in a new-to-you programming language, we [have] not provided any example tests. We also acknowledge that testing this sort of graphical program is difficult.”

Indeed it turned out to be difficult to test certain things, for instance that key bindings in the Tk GUI invoked the correct functions, since no way of triggering the events was found. The second Ruby assignment also had an SML part. Due to time constraints only the SML part was test driven, and the result was that it got full points whereas the Ruby part only got 54 out of 70 points. However, virtually no time was spent debugging and there were some provided tests that could be relied upon, so in this case the failure to reach a high grade was probably more due to the limited amount of time spent solving the assignment than whether TDD was used or not. In conclusion, TDD had benefits in terms of structure and solution robustness for all of the assignments regardless of

⁶²<https://www.coursera.org/course/proglang>

programming language, although the benefits would arguably have been greater if the assignments had been bigger.

8 Testing Culture

Different people have different opinions about testing: how it should be done, when it pays off and how exciting it is. Some see testing as a tool that helps to provide structure and improve the design of code, whereas others see it as a way to detect bugs and prevent others from braking the code. Most people agree that testing is hard, but many struggle to do it anyway. Attitudes towards testing are essential to whether or not it is carried out or not, and to how worthwhile it turns out to be in the end. Economic and technical circumstances, team culture and background of the developers have great impact on the level of testing, and in this section the impact of such aspects and viable ways forward are discussed.

Although it would certainly be interesting to discuss the extent to which SML testing is carried out, and reasons for it, no statistical or qualitative research was carried out within this area as part of this thesis. Supposedly, many who engage in SML programming are doing it for academic purposes rather than in true production environments, so the routines and cultures around testing are different than in typical server side languages such as Java and C#, but in this section we will focus on JS testing, except in the parts that are general to any programming language.

8.1 State of Events for JavaScript

Testing of JS might be uncommon because JS programmers are often web graphic designers, without that much experience of automated testing [41, question 10], or that many are new to JS [16, question 16]. A common conception is that JS in practice 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. However, lately there seem to have been a turn of events.

The overall quality of JS code might have improved over the last few years due to improved toolsets and testing methodologies entering the JS community. The new application frameworks has brought not only technical possibilities (as mentioned in section 8.5) but has also brought people with background from test-driven backend-programming, experienced with for example Ruby on Rails development. 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. [12, 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[14, 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 experienced 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 [41, question 43]. This tendency is closely related to the concept of spikes in TDD, see section 4.4.

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 [11]. Key points here are that tests can not substitute careful decision-making and that the maintainability of the tests themselves is also important, they suffer from code rot just like production code so they need to be treated similarly [12, question 36][11, 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 [21, 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 [55] 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 [21, p. 41].

Testing the right thing means focusing on the specifications and behaviour that provides true business value rather than for example 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, or else refactoring will be even harder later on [10]. A bad test on the other hand is one that is hard to understand, fails for the wrong reasons or makes the code hard to change.

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 one wants to be able to run them often [41, 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 Autotest⁶³ or some tool based on Guard⁶⁴ such as guard-jasmine⁶⁵ 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 Culture, Collaboration and Consensus

In June this year, Kent Beck wrote the following tweet⁶⁶:

“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”⁶⁷. What many probably fail to realise about why testing can be time consuming and hard, is that when writing tests the problems that are encountered would most of the time have to be solved anyway. The difference is that they are solved by formulating tests rather than staring at production code for the same amount of time. [2, question 11]

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[11], and to work together with experienced programmers and testers. Without this understanding there is a risk of frustration, inability to decide which tools to use and difficulties in motivating choices related to testing. Collaboration problems may occur when members of a team have different opinions about the purpose of testing. [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][41, question 30]

The attitude towards and experience with testing varies heavily, so developers with positive attitudes towards testing working with developers that do not have that attitude might be forced to not test as much as they otherwise would, in order to avoid frustration and dips in productivity by testing other people’s code. Testing may be advocated, especially in the beginning of a new project, but in the end adjusting to existing culture is also essential since most benefits of testing appear only after a long term commitment

⁶³<http://autotest.github.io/>

⁶⁴<https://github.com/guard/guard/>

⁶⁵<https://github.com/netzpirat/guard-jasmine/>

⁶⁶A tweet is a message sent to many using the social media www.twitter.com

⁶⁷<https://twitter.com/KentBeck/status/350039069646004224>, 2013-07-26

by the whole team [41, questions 31-32]. Ahnve brought up an example of whether everyone uses their own database for testing or not [12, question 36]. Doing so is usually a bad idea but it might still be worth considering whether or not it is worth the effort to create new databases for everyone, or switching to another, although these kinds of decisions might be limited by licenses and what the customer wants.

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. [41, question 10]

As a novice within testing it might be better to rely on proven methods until proper contextual knowledge has been obtained, 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. [12, questions 23-24]

8.3 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 [12, 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. [12, question 33][2, question 28]

A common feeling when writing tests is that a lot of effort is put into the tests and not as much in writing production code, but according to Edelstam that can be a good thing. When spending more time thinking about the problem and possible abstractions, elegant solutions tend to emerge. When writing the tests before the code, many of the problems that have to be solved would have to be solved anyway if the code had been written first, the difference is that the design is considered at an early stage rather than staring at incomplete code when solving a 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, an illusion of being more productive than is actually the case may manifest itself. One of the positive effects of TDD is that it can maintain track of direction, and help 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 good ideas regarding direction, but once such an idea has appeared, testing tends to be easier and helps to discover new aspects and scenarios which might have been left unnoticed without tests. [2, question 8]

It is important to remember that people tend to care most of things related to their guild, 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. [12, question 38]

8.4 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 [56] or giving faulty advice to investors [57] 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. [12, questions 14, 38]

8.5 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 scalability and structure of JS applications [12, question 23][6, 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”
[58]

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 [12, 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 [6, 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 [12, question 22][6, question 41].

In the last couple of years, the number of people testing their JS has increased – at a tech talk two years ago only a few participants raised their hands when asked such a question whereas now almost every single one does it. [2, question 1] 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 have ever been against TDD or testing in general. Positive experiences from testing in Ruby on Rails projects 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]

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. Cucumber⁶⁸ is a popular framework that was designed for this in the Ruby programming language and there is a JS version called Cucumber.js⁶⁹. Yadda,⁷⁰ Kyuri,⁷¹ Cucumis⁷² and Jasmine-given⁷³ are other examples of JS GTW-frameworks. [24, 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

⁶⁸<http://cukes.info/>

⁶⁹<https://github.com/cucumber/cucumber-js>

⁷⁰<https://github.com/acuminous/yadda>

⁷¹<https://github.com/nodejitsu/kyuri>

⁷²<https://github.com/noblesamurai/cucumis>

⁷³<https://github.com/searls/jasmine-given>

`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. [12, question 7][6, question 5-6]

Apart from testing frameworks, the ability to write tests is also influenced by what application framework are used [12, question 8]. According to Edelstam, given a choice, 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 with 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. [41, questions 12-14]

Employing a suitable application framework and 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, enabling better use of the framework and providing incentive for even better structure. [41, 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, a HTML page had to load in a browser in order to run tests. [41, question 9]

9 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 Lessons Learned

The goal of this thesis was to investigate the main problems with automated testing of SML and client side JS, and how they relate to development tools and practices.

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 [11, p. 127] and thereby encourage proper modularisation in the production code.

9.1.1 Testability Issues in JavaScript

What are the testability issues of client-side JS?

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

JS is often used in the front end layer of applications, an environment with many dependencies and very specific behaviour that can be difficult to handle in terms of code. Traditionally, browser automation tools have been used to cope with these challenges, and the tests often serve multiple purposes such as combining smoke testing with integration testing and system testing, with little or no mocking of dependencies. The main problems are that such tests take long to run, even with the introduction of headless browsers such as PhantomJS, and are typically brittle. Alternatives such as comparing screen shots automatically with diff tools and then manually inspect the ones that differ should probably be considered in many situations, despite the difficulty in making them self-checking. Under certain circumstances it is also possible to break out logic from the front-end code and test it with unit tests.

9.1.2 Testability Issues in Standard ML

What are the testability issues of SML?

First off, people do test their SML code, but not always, and it seems there are no standard ways of doing it. There is a need for a simple but good testing framework, QCheck has great benefits but is too hard to use for the average programmer or in scenarios where the process of writing tests need to be quick. Other frameworks, including DescribeSML, are too immature. The way forward could be to make QCheck easier to use, with new BDD syntax options, customisable failure reports and autotest capabilities, or to merge and collaboratively develop emerging testing frameworks further.

9.1.3 Stable and Maintainable Tests

Which considerations need to be made in order to write stable and maintainable tests?

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.

9.1.4 Testing Culture

How does testing culture affect productivity and quality of software?

In order to get started with testing, one has to understand why it should be done [2, question 38] and then practise with katas and begin to use testing in projects. Coding katas focused on front-end JS development are not very common, but not impossible to do [12, question 47].

One reason for that JS has suffered from a poor testing culture could be that it 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 JS 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, as long as care is taken to separate code that does different things. If there is uncertainty in how the code works, as can be the case when copying code from an external source, then this is not possible.

In a project without any testing routines, time may be saved 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 the procedure may have to be repeated 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.

Many people have never set up a testing framework in a project and overestimate how hard it is. Automatic test generation or integration with monitoring services can be hard, but installing a test framework and write the first test is typically not. 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.

9.2 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. This has already been done to some extent, see <http://www.imagemagick.org/> and <http://huddle.github.io/Resemble.js/>, but has not quite reached the industry yet.

Selenium IDE, offers the possibility of recording interactions and generate code for them, which in theory could be used to help reproduce bugs by letting users record the actions that led to it (see section 5.1.4). It would be an interesting future work to investigate how this could function in practise – what training and incentives would be needed for the users and how the tool would have to be modified to fit different situations and needs.

In section 5.3.2 it was mentioned that enforcing proper stubbing in JS can be hard, because JS offers little support for selective encapsulation. The possibility of introducing notions of protected variables and more elaborate namespace mechanisms could therefore

be an interesting area of investigation, or other means of automating the detection of incomplete stubbing.

As mentioned in section 6.1, the issue of testing asynchronous code is not unique to JS. Researching testing of asynchronous code in other programming languages, such as Erlang, Haskell and Clojure, and applying the same methods to JS could be an interesting study. The common denominators are the message passing of Erlang and the multiple thread and multiprocess programming of other languages. It is a common belief that parallel computing will become more important in the future, although this depends on programmers being able to write efficient parallel programs and the ability to capture race conditions through testing.

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 (see Appendix for transcript)*. 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] Per Rovegård. *Interview (see Appendix for transcript)*. Aug. 26, 2013.
- [7] Christian Johansen. *Test-Driven JavaScript Development*. ADDISON-WESLEY, 2010. ISBN: 9780321683915.
- [8] Mark Bates. *Testing Your JavaScript/CoffeeScript, Part 1 of 2*. URL: <http://www.informit.com/articles/article.aspx?p=1925618> (visited on 04/09/2013).
- [9] 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).
- [10] Martin Fowler. *Refactoring - Improving the Design of Existing Code*. ADDISON-WESLEY, 1999. ISBN: 0201485672.
- [11] Robert C. Martin. *Clean Code - A Handbook of Agile Software Craftsmanship*. PRENTICE HALL, 2009. ISBN: 9780132350884.
- [12] Marcus Ahnve. *Interview (see Appendix for transcript)*. Aug. 12, 2013.
- [13] 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.
- [14] Douglas Crockford. *JavaScript: The Good Parts*. O'Reilly Media, Inc., 2008. ISBN: 9780596517748.
- [15] 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).

- [16] Henrik Ekelöf. *Interview (see Appendix for transcript)*. Aug. 28, 2013.
- [17] Edward Heatt and Robert Mee. “Going Faster: Testing The Web Application”. In: *IEEE Software* (Mar. 2002), p. 63.
- [18] 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).
- [19] Jens Neubeck. “Testdriven Javascriptutveckling i webbapplikationer – En utvärdering av tekniker och möjligheter”. MA thesis. Kungliga Tekniska Högskolan (KTH), 2010.
- [20] Kent Beck. *Test-Driven Development By Example*. ADDISON-WESLEY, 2002. ISBN: 9780321146533.
- [21] Gerard Meszaros. *xUnit Test Patterns - refactoring test code*. ADDISON-WESLEY, 2007. ISBN: 9780131495050.
- [22] Liang Yuxian Eugene. *JavaScript Testing Beginner’s Guide*. Packt Publishing, 2010. ISBN: 9781849510004.
- [23] Evan Hahn. *JavaScript Testing with Jasmine*. O’Reilly Media, 2013. ISBN: 9781449356378.
- [24] Marco Emrich. *Behaviour Driven Development with JavaScript*. Developer.Press, 2013. ISBN: 9781909264113.
- [25] Hazem Saleh. *JavaScript Unit Testing*. Packt Publishing, 2013. ISBN: 9781782160625.
- [26] Pedro Teixeira. *Using Node.js for UI Testing*. Packt Publishing, 2013. ISBN: 9781782160526.
- [27] Mark Ethan Trostler. *Testable JavaScript*. O’Reilly Media, 2013. ISBN: 9781449323394.
- [28] 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.
- [29] Natalia Negara and Eleni Stroulia. “Automated Acceptance Testing of JavaScript Web Applications”. In: *19th Working Conference on Reverse Engineering* (2012), pp. 318–322.
- [30] Frodin 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.
- [31] 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.
- [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] Emden R. Gansner and John H. Reppy. *The Standard ML Basis Manual*. Cambridge University Press, 2004. ISBN: 0521794781.
- [34] Jeffrey D. Ullman. *Elements of ML Programming*. 2nd ed. Prentice Hall, 1998. ISBN: 0137903871.
- [35] Larry C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 1996. ISBN: 052156543X.
- [36] Stephen Gilmore. *Programming in Standard ML ’97: A Tutorial Introduction*. University of Edinburgh, 2003.
- [37] Robert Harper. *Programming in Standard ML*. Carnegie Mellon University, 2011.

- [38] Ute Schmid and Marieke Rohde. *Functional Programming with ML*. Universität Osnabrück, 2002.
- [39] Riccardo Pucella. *Notes on Programming Standard ML of New Jersey*. Cornell University, 2001.
- [40] Louis Morgan. “Random Testing of ML Programs”. MA thesis. University of Edinburgh, 2010.
- [41] Patrik Stenmark. *Interview (see Appendix for transcript)*. Aug. 7, 2013.
- [42] 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).
- [43] 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).
- [44] Douglas Crockford. *JSLint - The JavaScript Code Quality Tool*. URL: <http://www.jshint.com/lint.html> (visited on 05/01/2013).
- [45] Michael C. Feathers. *Working Effectively with Legacy Code*. Prentice Hall PTR, 2004. ISBN: 0131177052.
- [46] Juriy Zaytsev. *Refactoring Javascript with kratko.js*. URL: <http://perfectionkills.com/refactoring-javascript-with-kratko-js/> (visited on 09/11/2013).
- [47] SeleniumHQ. *Platforms Supported by Selenium*. URL: <http://docs.seleniumhq.org/about/platforms.jsp> (visited on 08/09/2013).
- [48] David Burns. *Selenium 2 Testing Tools Beginner’s Guide*. Packt Publishing, 2012. ISBN: 9781849518307.
- [49] 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).
- [50] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge university press, 2008. ISBN: 9780521880381.
- [51] 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).
- [52] Mark Bates. *Testing Your JavaScript/CoffeeScript, Part 2 of 2*. URL: <http://www.informit.com/articles/article.aspx?p=1930036&seqNum=5> (visited on 09/06/2013).
- [53] Fredrik Wendt. *Email conversation (see Appendix)*. Aug. 18, 2013.
- [54] John Resig. *JavaScript testing does not scale*. URL: <http://ejohn.org/blog/javascript-testing-does-not-scale/> (visited on 04/09/2013).
- [55] Cunningham & Cunningham Inc. *Arrange Act Assert*. URL: <http://c2.com/cgi/wiki?ArrangeActAssert> (visited on 04/24/2013).
- [56] 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).
- [57] 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).

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

Appendix

See <https://github.com/emilwall/exjobb>