Master's thesis

Automated testing of a dynamic web application

Niclas Olofsson

May 13, 2014

Technical supervisor: Mattias Ekberg, GOLI AB

Supervisor: Anders Fröberg, IDA, Linköping University

Examiner: Erik Berglund, IDA, Linköping University

Linköping University
Department of Computer and Information Science

Abstract

Software testing plays an important role in the process of verifying software functionality and preventing bugs in production code. By writing automated tests using code instead of conducting manual tests, the amount of tedious work during the development process can be reduced and the software quality can be improved.

This thesis presents the results of a conducted case study on how automated testing can be used when implementing new functionality in a Ruby on Rails web application. Different frameworks for automated software testing is used as well as test-driven development methodology, with the purpose of getting a broad perspective on the subject. We study common issues with testing in these kinds of applications, and discuss drawbacks and advantages of different testing approaches. We also look into quality factors which are applicable for tests, and analyze how these can be measured.

Acknowledgments

This final thesis was conducted at GOLI AB, on the business incubator LEAD during the spring of 2014. I would like to thank my dear colleagues Lina, Malin and Madeleine, as well as all other people at LEAD for good fellowship and encouragement during this period. I specially want to thank my technical supervisor Mattias for help, support, interesting discussions and ideas.

I would also like to give thanks to my supervisor Anders Fröberg for help with finding and narrowing down the problem formulation, and my examiner Erik Berglund for overall help support as well as feedback on the final report.

Contents

1 Introduction

1.1 Background

During code refactoring or implementation of new features in software, errors often occur in existing parts. This may have a serious impact on the reliability of the system, thus jeopardizing user's confidence for the system. Automatic testing is utilized to verify the functionality of software in order to detect bugs and errors before they end up in a production environment.

Starting new web application companies often means rapid product development in order to create the product itself, while maintenance levels are low and the quality of the application is still easy to assure by manual testing. As the application and the number of users grows, maintenance and bug fixing becomes an increasing part of the development. The size of the application might make it implausible to test in a satisfying way by manual testing.

The commissioner body of this project, GOLI, is a startup company developing a web application for production planning called GOLI Kapacitetsplanering (ECP). Due to requirements from customers, the company wishes to extend the application to include new features for handling staff manning. The current system uses automatic testing to some extent, but these tests are cumbersome to write and takes long time to run. The purpose of the thesis is to analyze how this application can begin using tests in a good way whilst the application is still quite small. The goal is to determine a solid way of implementing new features and bug fixes in order for the product to be able to grow effortlessly.

1.2 Problem formulation

The goal of this final thesis is to analyze how automated tests can be introduced in an existing web application, in order to detect software bugs and errors before they end up in a production environment. In order to do this, a case study is conducted focused on how this can be done in the GOLI production planning system. We use the results from this case study in order to discuss how testing can be applied to dynamic web applications in general.

The main research question is thus to determine how testing can be introduced in the scope of the GOLI ECP application, and how tests can be applied when implementing new functionality. We focus on techniques which are relevant this specific application, i.e. techniques relevant to web applications that uses Ruby on Rails [1] and KnockoutJS [2] with a MongoDB[3] database system for data storage. We investigate which kind of problems that are specific to this kind of environment, and how these kinds of problems can be solved.

1.3 Scope and limitations

There exists several different categories of software testing, for example performance testing and security testing. The scope of this thesis is tests in which the purpose is to verify the functionality of a part of the system rather than measuring its characteristics. This thesis also only covers automatic testing, as

[1] Ruby on Rails framework, `http://rubyonrails.org/`
[2] KnockoutJS framework, `http://knockoutjs.com/`
[3] MongoDB document database, `https://www.mongodb.org/`

opposed to manual testing where the execution and result evaluation of the test is done by a human. The term *testing* will hereby refer to automatic software testing unless specified otherwise.

We will also not cover testing static views or any issues related to the deployment of a dynamic web application, but rather testing of the dynamic application itself.

1.4 Conventions and intended audience

The intended audience of this report is primarily people with some or good knowledge of programming and software development. A great deal of knowledge in the area of software testing or test methodologies should however not be required. The report can probably also be of interest for people without programming knowledge which is interested in the area of software testing and development.

Code examples are written in Ruby[4] 2.1.1 since that is the primary language of this thesis, but with emphasis on being understandable by people without knowledge of this language rather than using Ruby-specific tools and practices. For example, implicit return statements are avoided since these are nontrivial to understand for people used to languages without this feature, such as Python or Java.

The built-in Ruby module *Test::Unit* is used for general test code examples in order to preserve independence of a specific testing framework to as wide extent as possible, although other testing frameworks are also mentioned and exemplified in the report.

The area of software development contains several terminologies which are similar or exactly the same. In cases where multiple different terminologies exists for a certain concept, we have chosen the term with most hits on Google. The purpose of this was to choose the most widely- used term, and the number of search results seemed like a good measure for this. Footnotes with alternative terminologies is present where applicable.

[4] `https://www.ruby-lang.org`

2 Methodology

This chapter outlines the general research methodology of this thesis. Readers which are not interested in the academic properties of this report may safely skip this chapter.

2.1 Introduction

The methodology of this thesis is generally based on the guidelines proposed by Runeson and Höst [34] for conducting a case study. An objective is defined and a literature study is conducted in order to establish a theoretical base. The theory is then evaluated by applying it in a real-life application context, and the result is analyzed in order to draw conclusions about the theory.

2.2 Literature study

The literature study was based on the problem formulation, and therefore focuses on web application testing overall and how it can be automated. In order to get a diverse and comprehensive view on these topics, multiple different kinds of sources were consulted. As a complement to a traditional literature study of peer-reviewed articles and books, we chosen to also consider blogs and video recordings of conference talks.

While blogs are not either published nor peer-reviewed, they often express interesting thoughts and ideas, and may often give readers a chance to leave comments and discuss its contents. This might not qualify as a review for a scientific publication, but it also gives greater possibilities of leaving feedback on outdated information and is more open for discussion than traditional articles. Conference talks has similar properties.

Blogs and conference talks does have another benefit over articles and books since they can be published instantly. The review- and publication process for articles is long and may take several months, and also might not be available in online databases until after their embargo period has passed [10, 35]. This can make it hard to publish up-to-date scientific articles about some web development topics, since the most recent releases of well-used frameworks are less than a year old [4, 9, 12].

Utilized alternative sources are mainly relied upon recognized people in the open-source software community. One main reason for this is that large parts of the web development community as well as the Ruby community are pretty oriented around open-source software and agile approaches. This is also the case for several test-driven techniques and methodologies. Due to this, one might notice a skew in this report towards agile approaches and best-practices used by the open-source community.

2.3 Choices

The choice of frameworks for development was mainly given by the constituent, since the existing software was written in Ruby on Rails with KnockoutJS and a MongoDB database. The main server-side language was thus Ruby, and the client-side code was written in CoffeeScript. CoffeeScript is a scripting language which compiles into Javascript.

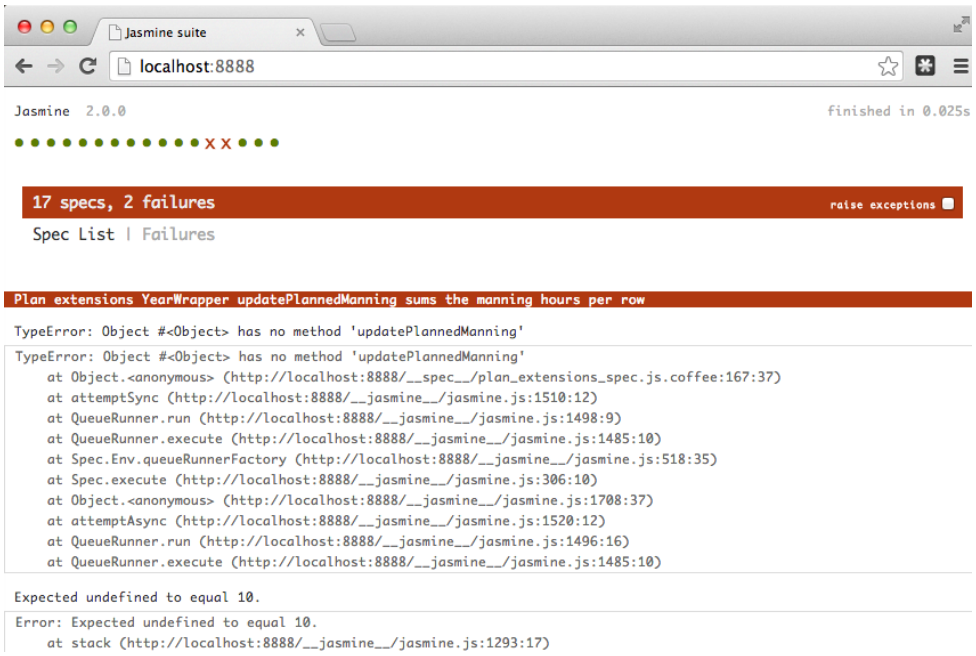

Figure 2.1: The test runner bundled with Jasmine.

For the choice of testing-related frameworks, we choose to look for frequently used and active developed open source frameworks. Technologies that are used by many people intuitively often has more resources on how they are used, and also has the advantage of being more likely to be recognized by future developers. Active development of used frameworks is crucial, most importantly since they are likely to be incompatible with future versions of other frameworks (such as Rails). Another benefit is that new features and bug fixes are released.

The Ruby Toolbox website [1], which uses information from the Github and RubyGems websites was consulted in order to find frameworks with mentioned qualities.

2.3.1 Ruby test frameworks

RSpec, RSpec-mocks, Cucumber, Capybara, FactoryGirl, Timecop, site_prism. (wip)

2.3.2 Javascript test frameworks

There exists a few different frameworks for testing Javascript or CoffeeScript code. We had previously good experiences from working with Jasmine[2]. We also found that this framework seemed to be very popular, actively developed and had good documentation[3]. Its syntax is inspired by RSpec, which also is an advantage since RSpec is used for the server side tests.

The Jasmine framework provides a way of writing tests, but a test runner is also required in order to run the tests. Jasmine ships with a basic test runner which was used initially, which worked out-of-the box using the Jasmine Ruby gem. A screenshot of this test runner is shown in figure 2.1.

The Jasmine test runner did however have multiple issues. First of all, it runs completely in the browser. Switching to the browser and reload in order to run the tests are not excessively problematic, but might be a little bit tiresome in the long run when using a test-driven development methodology. A larger problem was the fact that the asset handling, i.e. the compilation of CoffeeScript into Javascript, is handled by Rails upon each reload. Rails currently re-compiles all assets upon page load if any file has

[1] `https://www.ruby-toolbox.com/`
[2] `http://jasmine.github.io/`
[3] It is worth to mention that the Jasmine documentation is basically its own test suite with some additional comments. This works incredibly good in this case, presumably since it is a test framework.

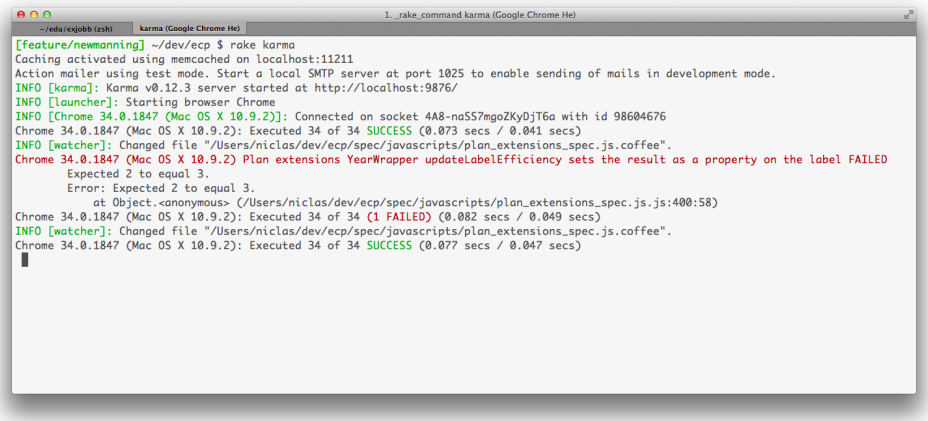

Figure 2.2: The Karma test runner.

been changed. This process takes a while, which means that each test run could take up to 10-15 seconds even though the actual tests only takes a fraction of a second to run. The asset compilation also got stuck for for apparently no reason once in a while. Since the server port on which the test runner runs cannot be specified, it is also impossible to restart the test runner without manually killing its process.

Due to these issues, we switched to the Karma[4] test runner. Karma originates from a master's thesis by Jína [22], and this thesis also covers several problems with other test runners (such as the mentioned Jasmine test runner). Karma was designed to solve several of these issues and to be used with test-driven software methodologies. Test are run in a browser as soon as a file is changed, and the results are reported back to the terminal and displayed as shown in figure 2.2.

2.3.3 Test coverage

Ruby test coverage

There are multiple different ways of analyzing test coverage, and the properties and conditions for each different kind of test coverage are discoursed in section 3.3.1. However, we were unable to find any test coverage tools for Ruby which analyzed anything else than statement coverage, which is the weakest test coverage metric. Quite much effort was spent on finding such tool, but without any success. Several websites and Stack Overflow-answers indicates that no such tool exists for Ruby at the time of this writing [1, 2, 3, 5]. On one hand, some of these sources are rather old and might be outdated, which would indicate that such tool could have been created recently. On the other hand would at least some of these sources probably been updated if such tool became available.

We ended up using the SimpleCov[5] tool for Ruby test coverage metrics. At the time of this writing, it is the most used Ruby tool for test coverage. It is also actively developed, works with recent Ruby versions and RSpec versions, and produces pretty and easy- to-read coverage reports in HTML (see figure 2.3).[5]

CoffeeScript test coverage

For the client-side CoffeeScript, we used a plug-in for the Karma test runner called karma-coverage[6]. This tool basically integrates Karma with Ibrik[7], which is a tool developed by Yahoo! for measuring test coverage of CoffeeScript code. We did initially have some problems with getting this tool to work correctly, since Ibrik internally uses another CoffeeScript compiler; CoffeeScriptRedux, than the compiler used when tests itself are run. CoffeeScriptRedux is more strict and yielded syntax errors in some of our files, which could be compiled correctly in the production code. The latest available release of Ibrik

[4] http://karma-runner.github.io/
[5] https://github.com/colszowka/simplecov
[6] https://github.com/karma-runner/karma-coverage
[7] https://github.com/Constellation/ibrik

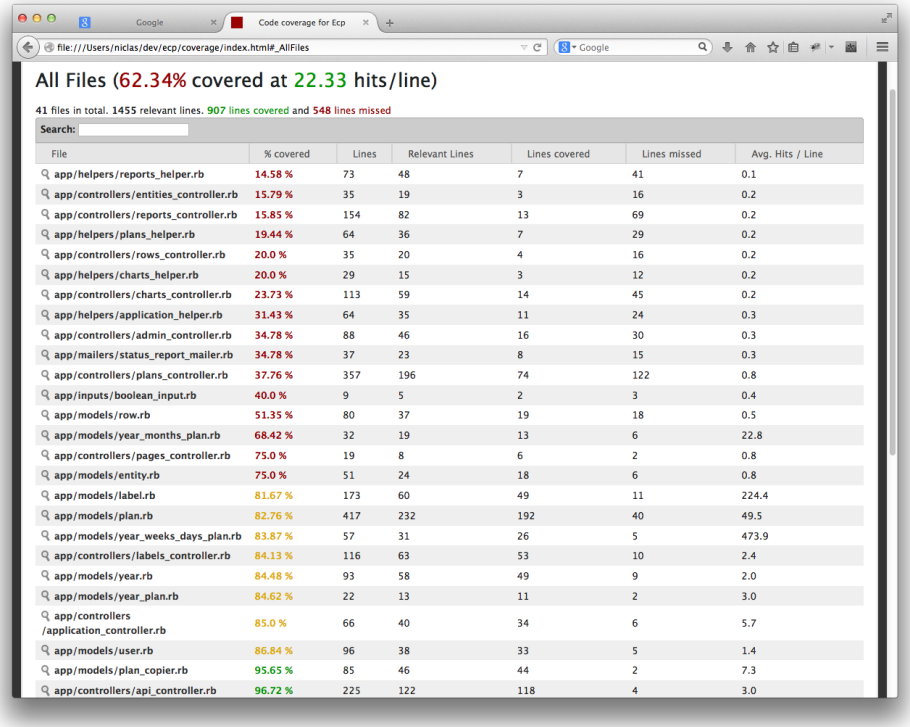

Figure 2.3: A coverage report generated by SimpleCov

(version 1.1.1) also had major issues with certain constructs in CoffeeScript, which made the files impossible to analyze. This issues was however fixed in the development version. Ibrik was first released in December 2013, which may explain its immaturity. Ibrik internally uses istanbul-js for the coverage analysis and report generation.

The chosen solution worked very well after sorting out the issues. Statement coverage as well as branch coverage is supported, and it yields useful reports. As with SimpleCov, the coverage reports are produced as an interactive HTML report (see figure 2.4).

Test coverage issues

One issue with using test coverage in this particular project is the fact that very few files were added. Most of the changes was made in existing classes and files, either as new functions or as changes to existing functions. Since the overall test coverage is measured per file, it is impossible to get an exact measure of how well-tested the new and re-factored code is, since old and completely untested code lower the test coverage.

We have tried to mitigate this by looking at the coverage reports by hand, and try to determine a subjective measure of how good the test coverage is for new and re-factored code. Figure 2.5 shows an excerpt of a coverage report. In this case, our subjective measure would say that the function `exports.getProduction()` is completely untested. The function `exports.getLabelId()` is well tested and has full statement- as well as branch coverage. The function `exports.formatValues()` has full statement coverage, but non-optimal branch coverage since the cases where `x` or `y` is not set are not covered.

2.3.4 Mutation analysis

There exists a few different tools for mutation analysis of Javascript code. The ones we have found originates from academic research papers. Mirshokraie et al. [27] proposes a solution which has been

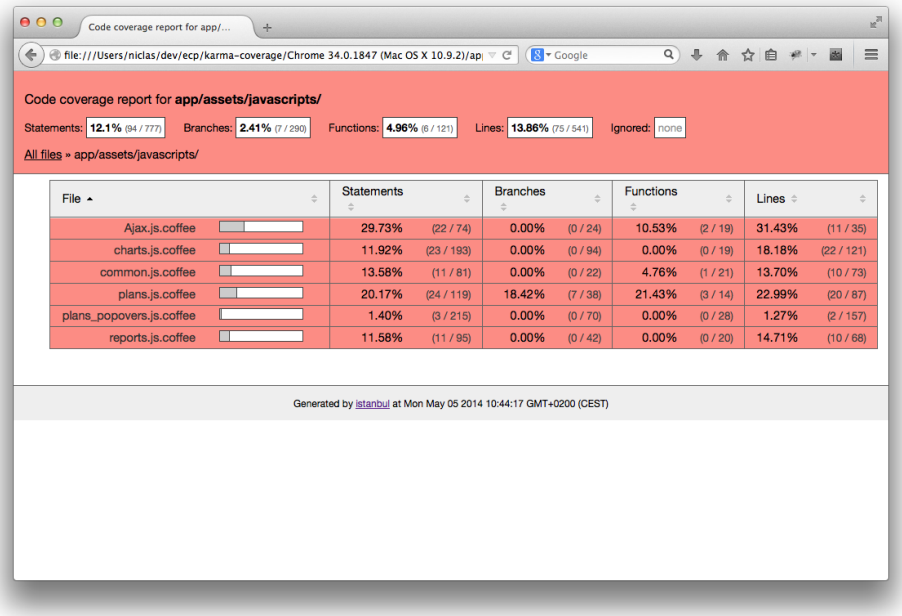

Figure 2.4: A coverage report generated by istanbul-js using karma-coverage

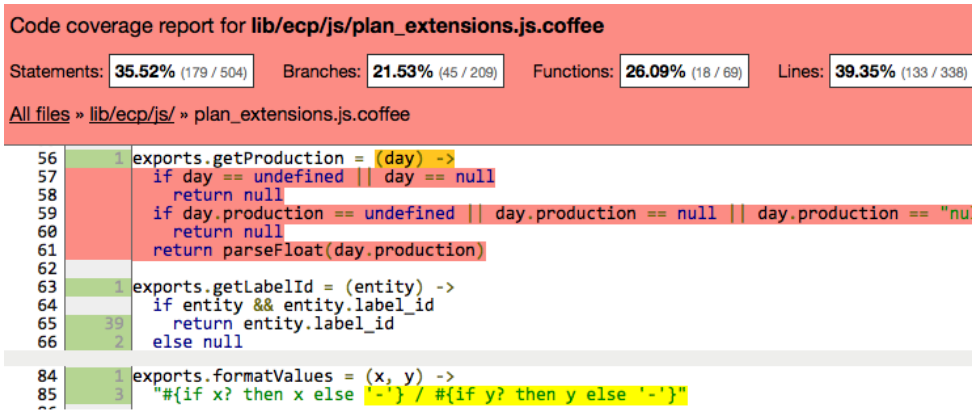

Figure 2.5: An excerpt from a coverage report generated using karma-coverage which demonstrates three different types of tested functions.

implemented as a tool called Mutandis [8]. Nishiura et al. [29] presents another approach which has been released as AjaxMutator [9]. Praphamontripong and Offutt [32] proposes a system-level mutation testing approach called webMuJava.

Mutandis is based on website crawling tests. Although Mirshokraie et al. mentions that pure Javascript frameworks have been tested using this tool, its implementation showed to be too specific to be considered in our context. webMuJava does not seem to be publicly available, and also seems to be too tightly integrated with a specific back-end technique to be useful for Javascript-testing only.
AjaxMutator is in our opinion the most mature of all the considered frameworks. It provides some basic documentation and installation instructions, and the amount of implementation needed in order to begin using it in a new project is reasonable. However, AjaxMutator currently only supports Javascript tests written in Java using JUnit[10]. Using it for mutation testing tests written in CoffeeScript using Jasmine would probably be possible, but the effort of doing this is outside the scope of this thesis.

[8] https://github.com/saltlab/mutandis/
[9] https://github.com/knishiura-lab/AjaxMutator
[10] http://junit.org/

3 Theory

3.1 Levels of testing

One fundamental part of all software development is the concept of abstraction. Abstraction can be described as a way of decomposing an application into different levels, with different level of detail. This allows us to forget about certain details of the software, and instead focus on other details. Consider the development of a simple game with basic 2D graphics. On a very low level, this requires a tremendous amount of work in order to shuffle data between hardware buses, perform memory accesses and CPU operations. By using higher abstraction levels, we can let third-party frameworks take care of stuff such as drawing graphics to the screen, let the operating system and programming language take care of bus and memory accesses. This allows us to focus on designing the game logic itself, and just forget all about the necessary stuff that just works.[23]

In the same way, testing can be performed at several different levels. There are several ways of defining these levels, but one way of describing it is like a pyramid (figure 3.1). We can imagine testing at different levels as holding a flashlight at different levels of the pyramid. If we hold the flashlight at the top of the pyramid, the flashlight will illuminate a large part of the pyramid. If the flashlight is hold at the bottom of the pyramid, a much smaller piece of the pyramid will be illuminated. Similar to this, testing at a high level lets us forget about a lot of details, and a large part of the code must be run in order for the test to complete. Testing at a lower level requires a much smaller part of the code to be run. Different levels of testing has different advantages, drawbacks and uses, which will be covered in the following subsections.

3.1.1 Unit-testing

Unit testing[1] refers to testing of small parts of a software. In practice, this often means testing a specific class or method of a software. The purpose of unit tests is to verify the implementation, i.e. to make sure that the tested unit works correct [31].

Since each unit test only covers a small part of the software, it is easy to find the cause for a failing test. Writing unit tests alone does not give a sufficient test coverage for the whole system, since unit tests only assures that each single tested module works as expected. A well- tested function for validating a 12-digit personal identification number is worth nothing if the module that uses it passes a 10-digit number as input [13].

Testability

Writing unit tests might be really hard or really easy depending on the implementation of the tested unit. Hevery [20] claims that it is impossible to apply tests as some kind of magic after the implementation is done. He demonstrates this by showing an example of code written without tests in mind, and points out which parts of the implementation that makes it hard to unit-test. Hevery mentions global state variables, violations of the Law of Demeter, global time references and hard-coded dependencies as some causes for making implementations hard to test.

Global states infers a requirement on the order the tests must be run in, which is bad since it is often non-deterministic and therefore might change between test runs. Global time references is bad since it

[1] Also called low-level testing, module testing or component testing

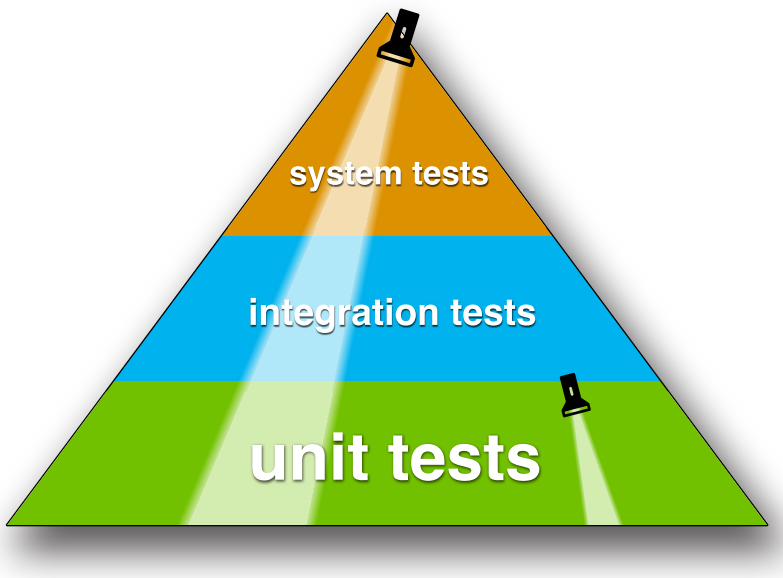

Figure 3.1: The software testing pyramid, with two flashlights at different levels illustrating how the level of testing affects the amount of tested code.

depends on the time when the tests are run, which means that a test might pass if it is run today, but fail if it is run tomorrow.

The Law of Demeter[2] means that one unit should only have limited knowledge of other units, and only communicate with related modules [8]. If this principle is not followed, it is hard to create an isolated test for a unit which does not depend on unrelated changes in some other module. The same thing also applies to the usage of hard-coded dependencies. This makes the unit dependent on other modules, and makes it impossible to replace the other unit in order to make it easier to test.

Hevery shows how code with these issues can be solved by using dependency injection and method parameters instead of global states and hard-coded dependencies. This makes testing of the unit much easier.

Stubs, mocks and fakes

Another way of dealing with dependencies on other modules is to use some kind of object that replaces the other module. The replacement object has a known value that is specified in the test, which means that changes to the real object will not affect the test. The reasons for using replacement objects is often to make it more robust to code changes outside the tested unit. It might also be used instead of calls to external services such as web-based API:s in order to make the tests run when the service is unavailable, or to be able to test implementations that depends on classes that has not been yet.

The naming of different kinds of replacement objects may differ, but two often used concepts are *stubs* and *mocks*. Both these replacement objects are used by the tested unit instead of some other module, but mocks sets expectations on how it can be used by the tested module on beforehand. [18]

Another type of replacement object are *fakes* or *factory objects*. This kind of object is typically provides a real implementation of the object that it replaces, as opposed to a stub or mock which only has just as many methods or properties that is needed for the test to run. The difference between a fake object and the real object is typically that fake objects uses some shortcut which does not work in production. One example is objects that are stored in memory instead of in a real database, in order to gain performance. [18]

[2] Also called the principle of least knowledge

Code listing 3.1: Example of how mocking might make tests unaware of changes which breaks functionality

```
1  class Cookie
2      def eat_cookie
3          return "Cookie is being eaten"
4      end
5  end
6
7  class CookieJar
8      def take_cookie
9          self.cookies.first.eat_cookie()
10         return "One cookie in the jar was eaten"
11     end
12 end
13
14 def test_cookie_jar
15     Cookie.eat_cookie = Mock
16     assert CookieJar.new.take_cookie() == "One cookie in the jar was eaten"
17 end
```

Bernhardt [16] mentions some of the drawbacks with using mocks and stubs. If the interface of the replaced unit changes, this might not be noticed in our test. Consider the scenario given in 3.1. In this example we have written a test for the `take_cookie()` method of the `CookieJar` class, which replaces the `eat_cookie()` method with a stub in order to make the `CookieJar` class independent of the `Cookie` class.

If we rename the `eat_cookie()` method to `eat()` without changing the test or the implementation of `take_cookie()`, the test might still pass although the code will fail in a production environment. This is since we have mocked an object which no longer exists in the `Cookie` class.

Some testing frameworks and plug-ins detects replacement of non-existing methods and warns or makes the test fail if such mocks are created [16]. Another possible solution is to re-factor the code to avoid the need for mocks or stubs.

3.1.2 Integration-testing

Since units tests only assures that a single unit works as expected, faults may still reside in how the units works together. The purpose of integration testing is to test several individual units together, in order to see if they still work together as intended.

Huizinga and Kolawa [21] states that integration tests should be built incrementally by extending unit tests so that they span over multiple units. The scope of the tests is increased gradually, and both valid and invalid data is given into the integration tested system unit in order to test the interfaces between smaller units. Since this process is done gradually, it is possible to see which parts of the integrated unit that is faulty by examining which parts have been integrated since the latest test run.

Pfleeger and Atlee [31] refers to this type of integration testing as *bottom-up testing*, since several low-level modules (modules at the bottom level of the software) are integrated into higher-level modules. Multiple other integration approaches such as *top-down testing* and *sandwich testing* are also mentioned. The difference between the approaches is in which order the units are integrated.

Rainsberger [33] criticizes one kind of integration tests, which he refers to as *integrated tests*. This refers to integration tests that are testing the functionality of multiple units in the same way as unit tests, i.e. by input data and examine the output. When testing multiple units in this way, one loses the ability to see which part of all the tested units that are actually failing. As the number of tested units rises, the number of possible paths will grow exponentially. This makes it hard to see the reason for a failed test, but also makes it very hard to decide which of all this paths that needs to be testes. Integrated tests therefore puts much less pressure on the tested functions compared to unit tests.

Rainsberger claims that this fact makes developers more sloppy, which increases the risk of introducing mistakes that goes unnoticed through the test suite. If this is solved by writing even more integration tests, developers have less time to do proper unit tests and instead introduce more sloppy designs, forming an infinite feedback-loop.

One may argue that this argument is based on the fact that integrated tests to a large part are used instead of unit tests. The purpose of integration tests are not to verify special cases of individual modules but to make sure that they work together as intended. However, it would be very hard to assure that all modules are working together by just using a few test cases due to the number of possible execution paths. The running time would also be long since large parts of the code base needs to be run.

Instead of integrated tests, another type of integration tests are proposed by Rainsberger called contract- and collaboration tests. The purpose of these tests is to verify the interface between all unit-tested modules by using mocks to test that Unit A tries to invoke the expected methods on Unit B (contract test). In order to avoid errors due to mocking, tests are also needed to make sure that Unit B really responds to the calls that are expected to be performed by Unit A in the contract test. The idea of this is to build a chain of trust inside our own software via transitivity. This means that if Unit A and Unit B works together as expected, and Unit B and Unit C also works together as expected, Unit A and Unit C will also work together as expected.

3.1.3 System-testing

System testing is conducted on the whole, integrated software system. Its purpose is to test if the end product fulfills specified requirements, which includes determining whether all software units (and hardware units, if any) are properly integrated with each other. In some situations, parameters such as reliability, security and usability is tested.[21]

The most relevant part of system testing for the scope of this thesis is functional testing. The purpose of this is to verify the functional requirements of the application at the highest level of abstraction. In other words, one wants to make sure that the functionality used by the end-users works as expected. This might be the first time where all system components are tested together, and also the first time the system is tested on multiple platforms. Because of this, some types of software defects may never show up until system testing is performed.[21]

Huizinga and Kolawa proposes that system testing should be performed as black box tests corresponding to different application use-cases. An example could be testing the functionality of an online library catalog by adding a new users to the system, log in and perform different types of searches in the catalog. Domain-testing techniques such as boundary- value testing are used in order to narrow down the number of test cases.

3.2 Software development methodologies

During the ages of computers and software development, several software development methodologies have been proposed. A software development methodology defines different activities and models which can be followed during software development. Such activities can for instance be to define software requirements or writing software implementations, and the methodology typically defines a process with a plan for how and in which order the activities should be done. The waterfall model and the V model are two classical examples of software development methodologies.[36]

Extreme programming (XP) is a software methodology created by Kent Beck, who published a book on the subject in 1999 [14]. This methodology was probably the first to propose testing as a central part of the development process, as opposed to being seen as a separate and distinct stage [36]. These ideas was later further developed, and the concept of *testing methodologies* was founded. A testing methodology defines how testing should be used within the scope of a software development methodology, and the

following sections will focus on the two most prominent ones.

3.2.1 Test-driven development

Test-Driven Development (TDD) originates from the Test First principle in the Extreme Programming methodology, and is said to be one of the most controversial and influential agile practices [25]. It should be noticed that the phrase *test driven development* is often used in several other contexts where discusses general practices for testable code are discussed. In this section, we consider the basics of TDD in its original meaning.

Madeyski [25] describes two types of software development principles; Test First and Test Last. When following the Test Last methodology, functionality is implemented in the system directly based on user stories. When the functionality is implemented, tests are written in order to verify the implementation. Tests are run and the code is modified until there seems to be enough tests and all tests passes.

Following the Test First methodology basically means doing these things in reversed order. A test is written based on some part of a user story. The functionality is implemented in order to make the test pass, and more tests and implementation is added as needed until the user story in completed [25].

The Test First principle is a central theme in TDD. Beck [15] describes the basics of TDD in a "mantra" called *Red/green/refactor*. The color references refers to colors used by often test runners to indicate failing or passing tests, and the three words refers to the basic steps of TDD.

- Red - a small, failing test is written.
- Green - functionality is implemented in order to get the test to pass as fast as possible.
- Refactor - duplications and other quick fixes introduces during the previous stage is removed.

According to Beck, TDD is a way of managing fear during programming. This fear makes you more careful, less willing to communicate with others, and makes you avoid feedback. A more ideal situation would be one there developers instead try to learn fast, communicates much with others and searches out constructive feedback.

Some arrangements are required in order to practice TDD in an efficient way, which are listed below.

- Developers needs to write tests themselves, instead of relying on some test department writing all tests afterwards. It would simply not be practical to wait for someone else all the time.
- The development environment must provide fast response to changes. In practice this means that small code changes must compile fast, and tests need to run fast. Since we make a lot of small changes often and run the tests each time, the overhead would be overwhelming otherwise.
- Designs must consist of modules with high cohesion and loose coupling. It is very impractical to write tests for modules with many of unrelated input and output parameters.

3.2.2 Behavior-driven development

Behavior-Driven Development (BDD) is claimed to originate from an article written by North [30], and is based on Test-Driven Development (TDD) [6], which was discussed in section 3.2.1. This section is based upon the original article by North [30].

North describes that several confusions and misunderstandings often appeared when he taught TDD and other agile practices in projects. Programmers had trouble to understand what to test and what not to test, how much to test at the same time, naming their tests and understanding why their tests failed. North thought that it must be possible to introduce TDD in a way that avoids these confusions.

Instead of focusing on what test cases to write for a specific feature, BDD instead focuses on behaviors that the feature should imply. Instead of using regular function names, each test is described by a string (typically starting with the word *should*). For a function calculating the number of days left until a given

Code listing 3.2: A small example program for explaining different test coverage concepts.

```
1  def foo(a, b, x)
2    if a > 1 and b == 0
3      x = x / a
4    end
5    if a == 2 or x > 1
6      x = x + 1
7    end
8  end
```

date, this could for example be "should return 1 if tomorrow is given" or "should raise an exception if the date is in the past".

Using strings instead of function names solves the problem of naming tests - the string describing the behavior is used instead of a traditional function name. It also makes it possible to give a human- readable error if the module fails to fulfill some behavior, which can make it easier to understand why a test fails. It also sets a natural barrier for how large the test should be, since it must be possible to describe in one sentence.

After coming up with these ideas, North realized that writing behavior-oriented descriptions about a system had much in common with software analysis. They came up with scenarios on the following form to represent the purpose and preconditions for behaviors:

Given some initial context (the givens),
When an event occurs,
Then ensure some outcomes.

By using this pattern, analysts, developers and testers can all use the same language, and is hence called a *ubiquitous language*. Multiple scenarios are written by analysts to specify the properties of the system, which can be used by developers as functional requirements, and as desired behaviors when writing tests.

3.3 Evaluating test efficiency

One key property of tests is that they must be able to detect bugs and errors in the code, since that is typically the very main reason for writing tests at all. Several measures exists for discovering how well tests satisfies this property.

3.3.1 Test coverage

Test coverage[3] is a measure to describe to which degree the program source code is tested by a certain test suite. If the test coverage is high, the program has been more thoroughly tested and probably has a lower chance of containing software bugs.[7]

Multiple different categories of coverage criterion exists. The following subsections are based on the chapter on test-case designs in Myers et al. [28] unless mentioned otherwise.

Statement coverage

One basic requirement of the tests for a program could be that each statement should be executed at least once. This requirement is called full *statement coverage*[4]. We illustrate this by using an example from Myers et al. [28]. For the program given in 3.2, we could achieve full statement coverage by setting

[3] Also known as *code coverage*
[4] Some websites, mostly in the Ruby community, refers to this as C0 coverage

`a = 2`, `b = 0` and `x = 3`.

This requirement alone does however not verify the correctness of the code. Maybe the first comparison `a > 1` really should be `a > 0`. Such bugs would go unnoticed. The program also does nothing if x and a are less than zero. If this was an error, it would also go unnoticed.

Myers et al. finds that these properties makes the statement coverage criterion so weak that it often is useless. One could however argue that it does fill some functionality in interpreted programming languages. Syntax errors could go unnoticed in programs written in such languages since they are executed line-by-line, and syntax errors therefore does not show up until the interpreter tries to execute the erroneous statement. Full statement coverage at least makes sure that no syntax errors exists.

Branch coverage

In order to achieve full branch coverage[5], tests must make sure that each path in a branch statement is executed at least once. This means for example that an if-statement that depends on a boolean variable must be tested with both `true` and `false` as input. Loops and switch-statements are other examples of code that typically contains branch statements. In order to achieve this for the code in 3.2, we could create one test where `a = 3`, `b = 0`, `x = 3` and another test where `a = 2`, `b = 1`, `x = 1`. Each of these tests fulfills one of the two if-conditions each, so that all branches are evaluated.

Branch coverage often implicates statement coverage, unless there are no branches or the program has multiple entry points. There is however still possible that we do not discover errors in our branch conditions even with full branch coverage.

Condition coverage

Condition coverage[6] means that each condition for a decision in a program takes all possible values at least once. If we have an if- statement that depends on two boolean variables, we must make sure that each of these variables are tested with both `true` and `false` as value. This can be achieved in 3.2 with a combination of the input values `a = 2`, `b = 0`, `x = 4` and `a = 1`, `b = 1`, `x = 1`.

One interesting this about the example above it that condition coverage does not require more test cases than branch coverage, although the former is often considered superior to branch coverage. Condition coverage does however not necessarily imply branch coverage, even if that is sometimes the case. A combination of the two conditions, *decision coverage*, can be used in order to solve make sure that the implication holds.

Multiple-condition coverage

There is however still a possibility that some conditions mask other conditions, which causes some outcomes not to be run. This problem can be covered by the *multiple-condition coverage* criterion, which means that for each decision, all combinations of condition outcomes must be tested.

For the code given in 3.2, this requires the code to be tested with $2^2 = 4$ combinations for each of the two decisions to be fulfilled, eight combinations in total. This can be achieved with four tests for this particular case. One example of variable values for such test cases are `x = 2`, `a = 0`, `b = 4` and `x = 2`, `a = b = 1` and `x = 1`, `a = 0`, `b = 2` and `x = a = b = 1`.

Myers et al. shows that a simple 20-statement program consisting of a single loop with a couple of nested if- statements can have 100 trillion different logical paths. While real world programs might not have such extreme amounts of logical paths, they are typically much larger and more complex than the simple example presented in 3.2. In other words can it is typically impossible to achieve full multiple-condition

[5] Sometimes called *decision coverage* and sometimes also referred to as C1 coverage
[6] Also called *predicate coverage* or *logical coverage*

Code listing 3.3: Example of a piece of code before mutation

```
1  def odd?(x, y)
2      return (x % 2) && (y % 2)
3  end
```

Code listing 3.4: Mutated versions of 3.3

```
1  def odd?(x, y)
2      return (x % 2) && (x % 2)
3  end
4
5  def odd?(x, y)
6      return (x % 2) || (y % 2)
7  end
8
9  def odd?(x, y)
10     return (x % 2) && (0 % 2)
11 end
12
13 def odd?(x, y)
14     return (x % 2)
15 end
```

test coverage.

3.3.2 Mutation testing

An alternative to draw conclusions from which paths of the code that is run by a test, as done when using test coverage, is to draw conclusions from what happens when we modify the code. The idea is that if the code is incorrect, the test should fail. Thus, we can modify the code so it becomes incorrect and then look at whether the test fails or not.

Mutation testing is done by creating several versions of the tested code, where each version contains a slight modification. Each such version containing a mutated version of the original source code is called a *mutant*. A mutant only differs at one location compared to the original program, which means that each mutant should represent exactly one bug.[11, 24]

There are numerous ways of creating mutations to be used in mutants. One could for example delete a method call or a variable, exchange an operator for another, negate an if-statement, replace a variable with zero or null-values, or something else. Code listing 3.3 shows an example of a function which should return true if both arguments are odd. Several mutated versions of this example is shown in code listing 3.4. The goal of each mutation is to introduce a modification similar to a bug introduced by a programmer.[24]

All tests which we want to evaluate are run for the original program as well as for each mutant. If the test results differ, the mutant is considered to be *killed*, which means that the test suite has discovered the bug. Some mutants may however have a change that does not change the functionality of the program. An example of this can be seen in 3.5, where two variables are equal and therefore not affected by replacing one of them with the other. This is called an *equivalent mutant*. The goal is to kill all mutants which is not equivalent mutants.[11, 24]

Lacanienta et al. [24] presents the results of an experiment where mutation testing was used in a web application with an automatically generated test suite. Over 4500 mutants was generated, with a test suite of 38 test cases. Running each test case for each mutant would require over 170000 test runs. A large part of the program was therefore discarded and the evaluation was focused on a specific part of the software, which left 441 mutants. 223 of these was killed, 216 was equivalent and 2 was not killed.

Code listing 3.5: Example of a program with an equivalent mutant

```
1  def some_function(x)
2      i = 0
3      while i != 2 do
4          x += 1
5      end
6      return x
7  end
8
9  def equivalent_mutant(x)
10     i = 0
11     while i < 2 do
12         x += 1
13     end
14     return x
15 end
```

The article by Lacanienta et al. exemplifies two challenges with mutation testing; a large amount of possible mutants, and a possibly large amount of equivalent mutants. In order for mutant testing to be efficient, the scope of testing must be narrow enough, the test suite must be fast enough, and equivalent mutants must be possible to detect or not be generated at all. Lacanienta et al. uses manual evaluation to detect equivalent mutants, which is probably impracticable in practice. Madeyski et al. [26] presents an overview of multiple ways of dealing with equivalent mutants, but concludes that even though some approaches looks promising, there is still much work to be done in this field.

3.4 Other test quality factors

3.4.1 Execution time

Performance of the developed software is often considered to be of great importance in software development. Some people mean that the performance of tests are just as important.

Bernhardt [17] talks about the problem with depending too much on large tests which are slow to run. He also mentions how the execution time of a test can increase radically as the code base grows bigger, even if the test itself is not changed. If the system is small when the test is written, the test will run pretty fast even if it uses a large part of the total system. As the system gets bigger, so does the number of functions invoked by the test, thus increasing the execution time.

One of the main purposes of a fast test suite is the possibility to use test-driven software development methodologies. As discussed in section 3.2.1, a fast response to changes is required in order to make it practically possible to write tests in small iterations.

Even without using test-driven approaches, a fast test suite is beneficial since it means that the tests can be run often. If all tests can be run in a couple of seconds, they can easily be run every time a source file in the system is changed. This gives the developer instant feedback if something breaks.

In order to achieve fast tests, Bernhardt proposes writing a large amount of low-level unit tests which is focused on a small testing part of the system, rather than many system tests that integrates with large parts of the system.

Haines [19] also emphasizes the importance of fast tests, and proposes a way of achieving this in a Ruby on Rails application. The basic idea is the same as proposed by Bernhardt, namely separating business logic so it is independent from Rails and other frameworks. This makes it possible to write small unit tests which only tests an isolated part of the system, independent from any third-party classes.

3.4.2 Readability and ease of writing

wip

4 Approach

Based on the theory discussed in chapter 3, this chapter outlines a hypothesis about how testing of the ECP system should be implemented. We also constitute activities that should be done during the case study, and thus forms the base for the results presented in chapter 5. In chapter 6, we compare the outlined hypothesis described in this chapter to the actual results in order to see how well this hypothesis worked in practice.

4.1 Hypothesis

- A combination of behavioral-driven and test-driven development is used during development. Tests are written before implementation and is described using sentences.
- Tests are written in code using RSpec rather than using an ubiquitous language as used by Cucumber.
- The major part of all written tests are low-level unit tests.
- A few of system level tests are written in order to test the integration of all units together.
- System level tests are run with Selenium in order for them to detect cross-browser system bugs.
- Test coverage is analyzed continuously and full branch coverage should be striven for.

4.2 Case study

The case study is divided into three sub parts. The purpose of each sub part is to evaluate some aspects of the testing approach, in order to compose a good evaluation of the chosen testing approach when combined.

4.2.1 Refactoring of old tests

There have been attempts to introduce testing of the ECP system a while back, but this has stopped since the chosen approaches was found to be very cumbersome. At the start of this project, the implemented tests had not been maintained for a very long time, which resulted in that many tests failed although the system itself worked fine.

As mentioned in section 4.1, TDD methodology is used during the case study. This methodology is based on the fact that tests are written before implementation of new features and then run iteratively during development. The test suite should pass, then fail after a new test has been implemented, and then pass again after the new feature has been implemented. This of course presupposes that existing tests can be run and give predictable results.

The first part of the case study is therefore to make all old tests run. Apart from being a prerequisite for new tests and features to be implemented, it also gives a view on how tests are affected as new functionality is implemented. This is especially interesting since it otherwise would be impossible to evaluate such factors in the scope of a master's thesis. It also gives a perspective on some of the advantages and drawbacks of the old testing approach.

Another drawback of the old tests are the fact that they run too slow in order to be continuously in a TDD manner. Another objective of this part of the case study is therefore to make them faster, so at least some of the tests can be run continuously.

4.2.2 Implementation of new functionality

As mentioned in section 1.1, the commissioner body of this project wishes to implement support for staff manning in the ECP system. This functionality is implemented and tests are written for new parts of the system as well as for re-factored code.

The purpose of this part is, besides implementing the new feature itself, to evaluate test-driven development and how tests and implementation code can be written together by using TDD methodology and an iterative development process. We also gain more experience of writing unit tests in order to evaluate how different kinds of tests serves different purposes in the development process.

4.2.3 Analyzing quality metrics

In order to evaluate the tests written in previous parts of the case study, code coverage is used as a measure. The last part of the case study focuses on analyzing quality metrics of the application in general as well as for newly implemented functionality. We evaluate the tests written in previous parts of the case study and complement them if needed.

The purpose of this part is to get a more solid experience of using test coverage as a measure for code quality, and to produce a measurable output of the case study.

5 Results

This chapter presents the results of the case study. (wip)

5.1 Tools and frameworks

5.2 Chosen test approaches

5.3 Test efficiency

As discussed in section 3.3 and 3.4, there are several measures for evaluating the quality of tests. This section presents the results of these different metrics for the ECP software before and after different parts of the case study.

5.3.1 Test coverage

A quick overview of the results are presented in table 5.1. More information is presented in the following sections. As discussed in section 2.3.3, we were unable to find any tool for analyzing anything else than statement test coverage for Ruby.

Before the case study

Before the start of this master's thesis, the average statement coverage of all RSpec unit- and integration tests was 42 %.

For the client-side code, no unit-tests existed. The test coverage was thus zero at this state.

After the first part of the case study

The first part of the case study (described in section 4.2.1) was focused on rewriting broken tests, since some of the existing tests was not functional. During this period, a large part of the existing test suite was either fixed or completely rewritten. Many of the existing unit- and integration tests was rewritten and a few large acceptance-level test was replaced by more fine-grained integration tests. The statement coverage of the increased to 53 % after this part of the case study.

There were still no client-side unit-tests after this part, since the focus was fixing the existing server-side tests. The client-side test coverage was still zero at this state

After the second part of the case study

The second part of the case study (described in section 4.2.2) was focused on implementing new functionality while re-factoring old parts as needed and write tests for new as well as re-factored code. The first sprint of this part was focused on basic functionality, while the second sprint was focused on extending and generalizing the new functionality.

Phase	No. of tests	Test coverage
Before case-study	59	42.24 %
After first part	58	55.25 %
After second part	81	62.34 %

Table 5.1: Statement test coverage of RSpec unit- and integration tests at different phases.

Description	No. of tests	Test coverage
RSpec browser tests	3	46.3 %
Cucumber browser tests	8	61.2 %
All RSpec tests	84	66.8 %

Table 5.2: Statement test coverage including browser tests after the second part of the case study.

When the implementation of the new functionality was finished, statement coverage of unit- and integration tests for the server side was 62 %. A subjective measure indicated that new and re-factored functions in general has high statement coverage. In cases where full statement coverage is not achieved, the reason is generally special cases. The most common example is when error-messages are given when request parameters are missing or invalid.

Statement test coverage for the client-side was 21 % and the corresponding branch coverage was 10 %. A subjective measure indicates that almost all of the newly implemented functions achieves full statement coverage. The branch coverage is in general also high for newly implemented functions, but conditionals for special cases (such as when variables are zero or not set) are sometimes not covered.

Coverage including browser tests

All metrics in the previous sections refer to test coverage for unit- and integration tests. As one of the last steps of the second part of the case study, browser tests was added in order to do system-level testing.

The total statement test coverage for the RSpec browser tests alone was 46 %, and the statement test coverage for all Ruby tests was 67 %. With the tools used, it was not possible to measure Javascript test coverage for the test suite including browser tests.

The total statement test coverage for the Cucumber browser tests alone was 61 %. It was not possible to calculate the total coverage for Cucumber and RSPec tests combined.

5.3.2 Mutation testing

As discussed in section 2.3, we were unable to find any working frameworks for mutation testing our code. We did however do a manual test, where ten mutants was manually created for a single part of the code.

The tests for the evaluated Ruby controller killed six of the ten generated mutants (leaving four alive mutants). Three of the alive mutants mutated the same line in different ways.

For the client-side tests, eight out of ten mutants was killed. The two alive mutants however showed to be equivalent due to the occasionally erratic behavior of Javascript. For instance, the expressions `0 + 1` and `null + 1` both evaluates to `1`[1].

[1]There is a talk by Gary Bernhardt on this topic which is well worth watching. `https://www.destroyallsoftware.com/talks/wat`

Phase	No. of tests	Total time	Time per test	Tests per second
Before case-study	59	19.9 s	340 ms	3.0
After first part	58	8.3 s	140 ms	6.9
After second part	81	7.6 s	153 ms	10.7

Table 5.3: Execution times of RSpec integration- and unit tests at different phases

Phase	No. of tests	Total time	Time per test	Tests per second
Before case-study	12	43 s	3.6 s	0.27
After first part	10	160 s	18 s	0.056
After second part	8	155 s	17 s	0.056

Table 5.4: Execution times of Cucumber feature tests at different phases

5.4 Execution time

All execution times below are mean values of multiple runs and excluding start-up time. Execution times are those reported by each testing tool when run on a 13" mid-2012 Macbook Air[2] with 1.83 GHz Intel Core i5 processor and Mac OS 10.9.2. Google Chrome 34.0.1847 was used as browser for execution of browser tests and Jasmine unit tests. The browser window was focused in order to neutralize effects of Mac OS X power saving features.

An quick overview of the results are presented in table 5.3, 5.4, 5.6, and 5.5. The same results are also presented as text in the following sections.

5.4.1 Before the case study

The total running time of the 59 RSpec integration- and unit tests before the case study was 23.0 seconds. Average test execution time was 340 ms per test, which means an execution rate of 3.0 tests per second.

Twelve Cucumber browser tests existed at this time (with 178 steps), but none of these tests passed at this stage. After fixing the most critical issues with these tests, 36 steps passed in 42.7 seconds. The average time per test was 3.6 seconds.

5.4.2 After refactoring old tests

Several of the previous tests was removed and replaced by more efficient tests during the second part of the case study. 58 RSpec integration- and unit tests existed after this stage, an the total execution time was 8.3 seconds.

The running time of the ten Cucumber tests (with 118 steps) was 160 seconds, and the average time per test was 18 seconds.

5.4.3 After implementation of new functionality

In total, 81 RSpec integration- and unit tests existed at this stage, and the execution time of these was 7.6 seconds. Average test execution time was 94 ms per test, and the rate was 10.7 tests per second.

At this phase, 34 Jasmine unit tests had been written and the total running time of these was 0.043 seconds. Average execution time was 1.3 ms per test and the rate was 791 tests per second.

Two of the Cucumber tests had been re-factored into RSpec browser tests at this stage, and the running time of the remaining eight Cucumber browser tests (with 111 steps) was 155 seconds. The average time

[2] `http://www.everymac.com/systems/apple/macbook-air/specs/macbook-air-core-i5-1.8-13-mid-2012-specs.html`

Phase	No. of tests	Total time	Time per test	Tests per second
After second part	3	12.4 s	4.1 s	0.24
At latest revision	10	38.2 s	3.8	0.26

Table 5.5: Execution times of RSpec browser tests

Phase	No. of tests	Total time	Time per test	Tests per second
After second part	34	0.043 s	1.3 ms	791

Table 5.6: Execution times of Jasmine unit tests

per test was 19 seconds.

The running time of the three newly implemented RSpec browser tests was 12.4 seconds, with an average time of 4.1 seconds per test. After the case study, additional RSpec browser tests was added by other developers. At the latest revision of the code as of May 2014, there are ten RSpec browser tests, and the total running time of these are 38 seconds. The average time per test is 3.8 seconds.

5.5 Readability and ease of writing

wip

6 Discussion

This chapter discusses the results. (wip)

6.1 Experiences of test-driven development

6.2 Test efficiency

6.2.1 Usability of test coverage

As seen in section 5.3.1, the test coverage for the existing RSpec integration-tests was 42 % before this thesis project, which is quite much considering that the existing tests was partially duplicated and only was focused on a few specific parts of the system. One reason for this is found when the coverage reports are inspected, where we can see that certain kinds code gets full statement coverage, even though there are not any tests for them. Method definitions and field definitions on model objects are such types, since they are executed as soon the application loads.

A major part of the modules with high test coverage score in the beginning of the case study was modules without any methods, such as data models. Since these modules also can have methods which needs tests, it is not possible to just exclude all such modules from the coverage report either. This is an symptom of the weakness of statement coverage, which is also previously discussed in section 3.3.1. Statement coverage is a very weak metric and it is absolutely possible to create a non-empty class with 100 % statement coverage without writing any tests for it.

One may thus impeach the usefulness of statement test coverage. We did however find that apart from the coverage score being a bad indicator on how well-tested the code was, analyzing the statement coverage report was highly usable for finding untested parts of the code. The nature of our code was such that it in general contained few branches, which of course makes statement coverage considerably much more useful than for code with higher complexity.

On the client-side, we also had the ability to evaluate branch coverage. The most important difference between the coverage scores for branch coverage versus statement coverage is that the branch coverage is zero for all untested functions, which makes the branch coverage score a more sound metric for the overall testing level. Branch coverage of course also made it possible to discover a few more untested code paths.

6.2.2 Usability of mutation testing

We believe that mutation testing may be a good alternative to code coverage as method for evaluating test efficiency. Our small manual test indicated that mutation testing and test coverage is related, since several of the alive mutants modified the same line, which also showed to have zero statement test coverage. The mutation tests also found non- equivalent modifications to the code which was not discovered by the tests, even though they had full statement and branch test coverage.

Our experiences with mutation testing however shows that more work is required before it is possible to use it in our application. Mutation tests may also have limited usability on the server side, which in our case is mostly tested with higher-level integration-tests rather than with unit-tests. As mentioned

in section 3.3.2, efficient mutation testing requires the scope to be narrow and the test suite to be fast, due to the large amount of possible mutants. This is not the case for integration-level tests. Mutation testing might thus be more useful for the client-side, which contains more logic and therefore is tested with isolated unit-tests to a larger extent.

6.2.3 Efficiency of tests written during this thesis

Since most of the implemented code consisted of extensions or refactoring to existing files, it is hard to get a fair metric value for how well tested the modified parts of the software is. This is because these metrics are given in percent per file, and since the same file contains a lot of unmodified code, that will affect the result. It is also hard to detect which parts of the code that has been modified. We ended up using a self-constructed, subjective metric, which imposes questions on whether or not we can draw any solutions about how efficient the tests for the new functionality is.

We can however see that the overall test coverage has increased

6.3 Test execution time

6.3.1 Evolution of test execution time

As seen in section 5.4, the execution time of the RSpec integration- and unit tests has decreased for every part of the case study, even though the number of tests has increased. On average, each test is more than twice as fast after the second part compared to before the case study.

The major reason for the increase of speed comes from the refactoring of old tests, initiated in the first part of the case study. Some old tests was also re-factored during the second part, which explains the decrease in test execution time between the first and the second part of the case study. One reason for the drop in average execution time after the second part was also the fact that a few more unit-tests was written, compared to before.

The single most important factor for speeding up tests when re-factoring the old tests was to reduce the construction of database objects. Before the case-study, all test data for all tests was created before every test, since the database needs to be cleaned between each test in order for one test not to affect other tests. By using factory objects, we could instead only create the objects needed for each specific test, instead of creating a huge amount of data which is not even used. RSpec also caches factory objects between tests which uses the same objects, which gives some additional speed gain.

By looking at the execution times for the Cucumber feature test (table 5.4), it might seem like these tests became a great deal slower after the first part of the case-study. One issue is however that Cucumber stops executing a test as soon as some part of the test fails. Since almost all of the tests failed at an early stage, large part of the test suite was never executed and the execution time before the first part of the case study is therefore misleading.

6.3.2 Execution time for different test types

In section 3.1, we discussed different levels of testing, and how this affects the amount of executed code and the test execution time. Table 5.6 shows that our Jasmine isolated unit- tests runs at a rate of over 700 tests per second, while our system- level browser tests in table 5.5 runs at a rate of less than 0.3 tests per second. We can conclude that the principle of writing many isolated unit tests and just a few system-level tests seems very reasonable unless we choose to discard the execution time completely.

One other interesting thing is the large difference between the Cucumber browser tests and the RSpec browser tests. Even though both types of tests are in the category of system-level tests, RSpec browser tests are more than four times faster than the Cucumber tests. One reason for this might be the size of the test. As seen in table 5.2, the Cucumber tests cover more code than the RSpec browser tests, and

therefore takes longer time to execute. Another reason might be that some Cucumber tests are redundant. For example, the Cucumber test suite tests login functionality multiple times, while the RSpec browser test suite only tests this functionality once. Yet another reason might be the overhead of parsing the ubiquitous language used by Cucumber.

6.3.3 Importance of a fast test suite

One may argue that there is no reason to have a fast test suite, and one may also argue that a fast test suite is very important. We think the importance of tests being fast depends on the mindset on the people writing them, how often the tests are run and in which way. We will discuss some of our own experiences on this area.

Having a super-fast unit test suite such as our Jasmine tests is really nice when using TDD methodology, since it feels like the whole test suite is completed in the same moment a file is saved. However, using integration-level RSpec tests which runs in about 150 ms also works fine when using TDD-methodology, as long as we only run tests for the affected module. Running the whole integration test suite continuously takes too much time, though. For the RSpec tests, the startup time is also considerably long, which is annoying but can be mitigated by using tools like Zeus[1].

For browser tests, we believe that the important thing is the ability to run these in some reasonable time. In our case, we believe that the RSpec browser test suite fast enough to be run before each deploy without annoyance, while the Cucumber suite barely fulfills the criterion of being run within ẗreasonable time". A good approach would be to eventually re-factor the Cucumber test suite into unit-, integration- and RSpec browser tests. Running browser tests on a separate testing server instead of locally is another alternative.

6.4 Future work

During this thesis, we have discovered that the area of software testing is a very broad subject. The literature study disclosed several interesting topics, among we could only consider a few. Some topics was also planned to be covered in this thesis, but was later removed. This chapter discusses a few ideas on ways to further extend the work which has been considered in this thesis.

6.4.1 Ways of writing integration tests

As discussed in section 3.1.2, there are several opinions about integration tests and whether or not they should be written as integrated tests or not. We mentioned the concept of contract tests as proposed by Rainsberger [33], but we have not been able to test this concept in practice. It would be intriguing to see how such approaches could be used. Is this procedure useful in practice? How is the fault detection rate affected by the way of writing these tests? Does the choice of programming language have a large impact on how useful contract tests are?

6.4.2 Evaluation of tests with mutation testing

In section 3.3.2 and 2.3.4, we discuss the theory behind mutation testing and some of our efforts on this subject. Our results was not very successful though. We believe that mutation testing is a very interesting subject with plenty of research basis. It could be rewarding to dig further on this subject, and see if there would be possible to create more robust and production-ready framework for mutation testing of web applications. Is it possible to reduce to number of equivalent mutants and total number of mutants, so that this technique can be applied to larger parts of the code base? How does the level of testing affect the mutation testing results? What needs to be done in order to create a robust framework for mutation testing which is really useful in a real web application?

[1] `https://github.com/burke/zeus/`

6.4.3 Automatic test generation

Automated test generation is the theory behind how tests can be generated automatically.This area was initially a part of this thesis, but was removed in order to narrow down the scope. One approach is to examine the code and find test cases in order to achieve good test coverage, as discussed in section 3.3.1, an another approach is to simply test random input data. There are already quite many scientific articles on this topic, but it would be interesting to see how this would work in this specific environment. Is it possible to automatically generate useful test cases for a Ruby application? How can automated test generation be combined with the use of a test-driven development methodology? How is the quality metrics (i.e. test coverage, running time) affected by using generated tests?

6.4.4 Continuous testing and deployment

One purpose of this thesis is to examine how software testing can be used in order to prevent bugs in production code. Even if tests exists, there is however a chance that tests are accidentally removed, not run before deployment, or not tested in all browser versions. One way to mitigate the chance of this could be to run tests continuously, and perhaps automatically deploy changes as soon as all tests passes. It would be interesting to see how this can be achieved in practice in this specific environment. How can the risk of bugs slipping out in production environment be reduced? Does these changes to the way of running tests change the importance of quality metrics? Is it possible to automatically measure quality metrics when tests are run continuously?

7 Conclusions

Conclusions.
The subjective measurements mentioned in section 5.3.1, indicates that it is likely that the newly implemented functionality is well tested and that its test efficiency is high.

References

[1] Does C1 code coverage analysis exist for Ruby?, . URL http://stackoverflow.com/questions/289321/does-c1-code-coverage-analysis-exist-for-ruby. Accessed 2014-04-29.

[2] C1 or C2 coverage tool for Ruby, . URL http://stackoverflow.com/questions/11048340/c1-or-c2-coverage-tool-for-ruby. Accessed 2014-04-29.

[3] Code coverage and Ruby 1.9, . URL http://blog.bignerdranch.com/1511-code-coverage-and-ruby-1-9. Accessed 2014-04-29.

[4] KnockoutJS: Downloads - archive of all versions, . URL http://knockoutjs.com/downloads/index.html. Accessed 2014-02-10.

[5] The Ruby Toolbox: Code metrics, . URL https://www.ruby-toolbox.com/categories/code_metrics. Accessed 2014-04-29.

[6] Behavior-driven development, . URL http://en.wikipedia.org/wiki/Behavior-driven_development. Accessed 2014-02-11.

[7] Code coverage, . URL http://en.wikipedia.org/wiki/Code_coverage. Accessed 2014-02-19.

[8] Law of Demeter, . URL http://en.wikipedia.org/wiki/Law_of_Demeter. Accessed 2014-02-12.

[9] Django (web framework) - versions, . URL http://en.wikipedia.org/wiki/Django_%28web_framework%29#Versions. Accessed 2014-02-10.

[10] Embargo (academic publishing), . URL http://en.wikipedia.org/wiki/Embargo_%28academic_publishing%29. Accessed 2014-02-10.

[11] Mutation testing, . URL http://en.wikipedia.org/wiki/Mutation_testing. Accessed 2014-04-03.

[12] Ruby on Rails - history, . URL http://en.wikipedia.org/wiki/Ruby_on_Rails#History. Accessed 2014-02-10.

[13] Unit testing, . URL http://en.wikipedia.org/wiki/Unit_testing. Accessed 2014-02-12.

[14] Extreme programming, . URL http://en.wikipedia.org/wiki/Extreme_programming. Accessed 2014-05-06.

[15] Kent Beck. *Test-Driven Development by Example*. Addison-Wesley Longman, 2002. ISBN 9780321146533.

[16] Gary Bernhardt. Boundaries. Talk from SCNA 2012, . URL https://www.destroyallsoftware.com/talks/boundaries. Accessed 2014-02-10.

[17] Gary Bernhardt. Fast test, slow test. Talk from PyCon US 2012, . URL http://www.youtube.com/watch?v=RAxiiRPHS9k. Accessed 2014-02-10.

[18] Martin Fowler. Mocks aren't stubs. URL http://martinfowler.com/articles/mocksArentStubs.html. Accessed 2014-02-12.

[19] Corey Haines. Fast rails tests. Talk at Golden gate Ruby conference 2011. URL http://www.youtube.com/watch?v=bNn6M2vqxHE/. Accessed 2014-02-20.

[20] Miško Hevery. Psychology of testing. Talk at Wealthfront Engineering. URL http://misko.hevery.com/2011/02/14/video-recording-slides-psychology-of-testing-at-wealthfront-engineering/. Accessed 2014-02-12.

[21] Dorota Huizinga and Adam Kolawa. *Automated defect prevention: best practices in software management*. Wiley-Interscience, IEEE Computer Society, 2007. ISBN 9780470042120.

[22] Vojtěch Jína. Javascript test runner. *Master's thesis report from Czech Technical University in Prague*, 2013.

[23] Jeff Kramer and Orit Hazzan. The role of abstraction in software engineering. In *Proceedings of the 28th international conference on Software engineering*, pages 1017–1018. ACM, 2006.

[24] R. Lacanienta, S. Takada, H. Tanno, X. Zhang, and T. Hoshino. A mutation test based approach to evaluating test suites for web applications. *Frontiers in Artificial Intelligence and Applications*, 240, 2012.

[25] Lech Madeyski. *Test-driven development: an empirical evaluation of agile practice*. Springer-Verlag, 2010. ISBN 9783642042881.

[26] Lech Madeyski, Wojciech Orzeszyna, Richard Torkar, and Mariusz Jozala. Overcoming the equivalent mutant problem: A systematic literature review and a comparative experiment of second order

mutation. *IEEE Transactions on Software Engineering*, 40(1):23, 2014. ISSN 00985589.

[27] S. Mirshokraie, A. Mesbah, and K. Pattabiraman. Efficient javascript mutation testing. Proceedings - IEEE 6th International Conference on Software Testing, Verification and Validation, ICST 2013, pages 74–83, 2013.

[28] Glenford J. Myers, Tom Badgett, and Corey Sandler. *The art of software testing, third edition*. John Wiley & Sons, 2012. ISBN 9781118133132.

[29] Kazuki Nishiura, Yuta Maezawa, Hironori Washizaki, and Shinichi Honiden. Mutation analysis for javascript web application testing. The 25th International Conference on Software Engineering and Knowledge Engineering (SEKE'13), pages 159–165, 2013.

[30] Dan North. Introducing BDD. URL `http://dannorth.net/introducing-bdd/`. Accessed 2014-02-11.

[31] Shari Lawrence Pfleeger and Joanne M. Atlee. *Software engineering: theory and practice (international edition)*. Pearson, 2010. ISBN 9780138141813.

[32] Upsorn Praphamontripong and Jeff Offutt. Applying mutation testing to web applications. Sixth Workshop on Mutation Analysis (Mutation 2010), 2010.

[33] Joe B. Rainsberger. Integrated tests are a scam. Talk at DevConFu 2013. URL `http://vimeo.com/80533536`. Accessed 2014-02-12.

[34] Per Runeson and Martin Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, 2009.

[35] Elsevier Science. Understanding the publishing process in scientific journals. URL `http://biblioteca.uam.es/sc/documentos/understanding_the_publishling_process.pdf`.

[36] X. Zhang, T. Hu, H. Dai, and X. Li. Software development methodologies, trends and implications: A testing centric view. *Information Technology Journal*, 9(8):1747–1753, 2010. ISSN 18125638.