Master's thesis

# Automated testing of a dynamic web application

Niclas Olofsson

June 18, 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 are used as well as test-driven development methodology, with the purpose of getting a broad perspective on the subject. This thesis studies common issues with testing web applications, and discuss drawbacks and advantages of different testing approaches. It also looks into quality factors that 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 Boozon Ekberg, Malin Nylander and Madeleine Nylander, 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 Ekberg for help, support, interesting discussions and ideas.

I would also like to give thanks to my supervisor Anders Fröberg as well as my examiner Erik Berglund for overall help and support. Additionally, I would like to thank Sophia Björsner for linguistic feedback and help with improving my language.

Last but not least, I would like to thank my opponent Filip Källström as well as my friend Fredrik Palm for proof reading and helpful suggestions and discussions during the whole working process.

Niclas Olofsson
Linköping, June 2014

# Contents

# 1 Introduction

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 software defects 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 grow, maintenance and bug fixing becomes an increasing part of the development. The size of the application might make it unlikely to test in a satisfying way by manual testing.

The commissioning body of this project, GOLI, is a startup company that is developing a web application for production planning called GOLI Kapacitetsplanering. Due to requirements from customers, the company wishes to extend its application to include new features for handling planning of resources.

The existing application uses automatic testing to some extent. The developers however feel that these tests are cumbersome to write and take long time to run. The purpose of this 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.

## 1.1 Intended audience

The intended audience of this thesis is primarily people with some or good knowledge of programming and software development. It is however not required to have any knowledge in the area of software testing or test methodologies. This thesis can also be of interest for people without programming knowledge that is interested in the area of software testing and development.

## 1.2 Conventions and definitions

### 1.2.1 Writing conventions

Terms written in *italics* are subjects that appear further on in this thesis. Understanding the meaning of these may be required for understanding concepts later on.

Source references that appear within a sentence refer to the particular sentence, while source references at the end of a paragraph refer to the whole paragraph.

### 1.2.2 Choices and definitions

The term *testing* refer in this thesis to automatic software testing with the purpose of finding software defects, unless specified otherwise. The GOLI Kapacitetsplanering software is referred to as *the GOLI application*. The term *test-driven development methodologies* is used as a collective term for both test-

and behavioral driven development (explained in section 2.4.1 and 2.4.2).

Code examples are written in Ruby[1] 2.1.1 since that is the primary language of this project. The focus of the code examples is yet to be understandable by people without knowledge of Ruby, rather than using Ruby-specific practices. For example, implicit `return` statements are avoided since these may be hard to understand for people used to languages without this feature, such as Python or Java. Code examples typically originate from implemented code, but names of classes and functions have been altered for copyright, consistency and understandability reasons.

The built-in Ruby module Test::Unit is used for general test code examples in order to preserve the independence of a specific testing framework as far as possible, although other testing frameworks are also mentioned and exemplified in this thesis. Another reason for this choice is to avoid introducing unnecessary complexity for these examples.

The area of software development contains several terminologies that are similar or exactly the same. In cases where multiple different terminologies exist 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. A footnote with alternative terminologies is present when such terminology is defined.

## 1.3 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. The focus of the case study is how automated testing can be done in the specific application. The results from the case study are used for discussing how testing can be applied to dynamic web applications in general.

The main research question is to determine how testing can be introduced in the scope of the GOLI application. Which tools and frameworks are available, and how well do they work in the given environment? How can test-driven development methodologies be used and how can we evaluate the quality of the written tests? The research is focused on techniques that are relevant the specific application, i.e. techniques relevant to web applications that uses Ruby on Rails[2] and KnockoutJS[3] with a MongoDB[4] database system for data storage.

## 1.4 Scope and limitations

There exist 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 cover automatic testing, as opposed to manual testing where a human does the execution and result evaluation of the test. Testing static views or any issues related to the deployment of a dynamic web application is not covered by this thesis either.

[1] `https://www.ruby-lang.org`
[2] `http://rubyonrails.org/`
[3] `http://knockoutjs.com/`
[4] `https://www.mongodb.org/`

# 2 Theory

## 2.1 Introduction to software testing

"Software is written by humans and therefore has bugs" [sic]. This quote was coined by John Jacobs [6], and explains the basic reason for software testing. Most programmers would agree that defects tend to show up during the software development process as well as in the finished software. The ultimate goal of the testing process is to establish that a certain level of software quality has been reached, which is achieved by revealing defects in the code [27].

Automated software testing is typically performed by writing pieces of code[1] called *tests* [18]. Each test uses the implemented code that we want to evaluate, and performs different assertions to make sure that the result is the same as we expect. Code listing 2.1 shows a very simple function and code listing 2.2 shows a piece of code for testing it.

For writing more complicated tests, and in order to manage collections of tests, writing a bunch of `if` statements for every test is repetitive and tedious. Several languages provide `assert` statements[2], which checks if the given condition is true, and raises an exception otherwise – the assertion *fails*. The same test using assertions is shown in code listing 2.3. However, we often want to have more support than just statements like this. We may for example want to run a piece of code before each test for a specific module, assert that a function throws an exception, or present different error- messages depending on the assertion. A *testing framework*[3] typically provides such functionality. [18]

The code for each test is executed by a *test runner*, which in many cases is included as a part of the testing framework. The test runner finds and executes our collection of tests (called *test suite*), and then collects and reports the results back to the user. If at least one assertion in a test fails during the execution, the test is considered to have failed. A test may also fail due to syntax errors and unexpected exceptions.

## 2.2 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 permits the developer to ignore certain details of the software, and instead focus on other details.

Consider the development of a simple game with basic graphics. On the lowest level possible, a such game 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, one can use third-party frameworks for drawing graphics to the screen and detecting collisions. The operating system and programming language takes care of handling bus-accesses and memory management. This allows the developer to focus on designing the game logic itself, rather than bothering with drawing individual pixels or figuring out where in the memory to store data. [29]

[1] There are also techniques for recoding clicks when testing graphical interfaces, but that is not considered in this thesis.
[2] The Ruby core module does not provide this as a reserved word, but it is included as part of the Test::Unit module.
[3] Also called testing tool.

Code listing 2.1: An example function.

```
1
2  def plus(x, y)
3    return x + y
4  end
```

Code listing 2.2: A piece of code that could be used for testing the function in code listing 2.1.

```
1
2  def test_plus
3    if !(plus(1, 2) == 3)
4      raise
5    end
6    if !(plus(3, −4) < 0)
7      raise
8    end
9  end
```

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 as seen in figure 2.1. 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 permit us to ignore a lot of details. Due to the high level of abstraction, a large part of the code must be run in order for the test to be completed. Testing at a lower level requires a much smaller part of the code to be run. Different levels of testing have different advantages, drawbacks and uses, which are covered in the following subsections.

### 2.2.1 Unit testing

*Unit testing*[4] refers to testing of small parts of a software program. 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 [38]. Since each unit test only covers a small part of the software, it is easy to find the cause for a failing test. On the other hand, a single unit test only verifies that a small part of the application works as expected.

#### Testability

Writing unit tests might be hard or easy depending on the implementation of the tested unit. Hevery [26] 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[5], global time references and hard-coded dependencies as some causes for making implementations hard to test.

Global states infer a requirement on the order the tests must be run in. This is bad since the order might change between test runs, which would make the tests fail for no reason. Global time references is bad since it 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 means that one unit should only have limited knowledge of other units, and only communicate with related modules [12]. If this principle is not followed, it is hard to create an isolated

[4] Also called low-level testing, module testing or component testing.
[5] Also called the principle of least knowledge.

Code listing 2.3: A basic test for the function in code listing 2.1.

```
1
2 def test_plus
3   assert plus(1, 2) == 3
4   assert plus(3, -4) < 0
5 end
```

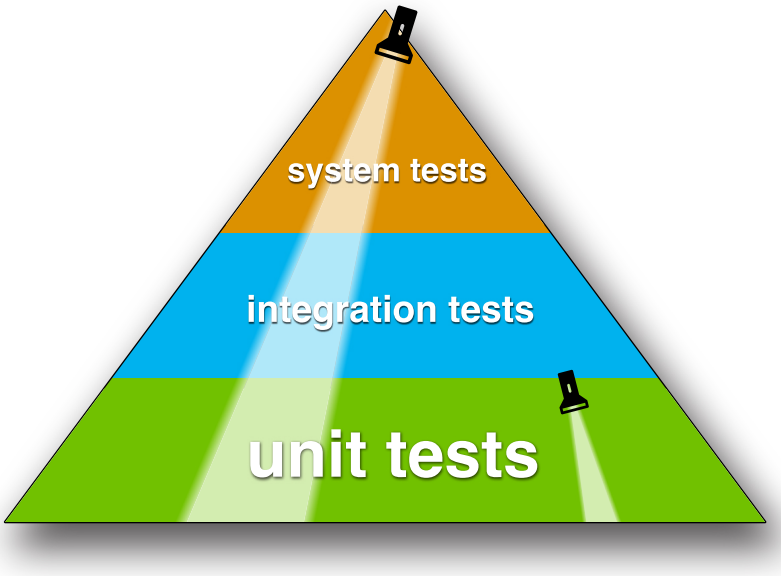


Figure 2.1: The software testing pyramid, with two flashlights at different levels illustrating how the level of testing affects the amount of tested code.

test for a unit that 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 factory objects

A central part of unit testing is the isolation of each unit, since we do not want tests to fail because of unrelated changes. One way of dealing with dependencies on other modules is to use some kind of object that replaces the other module during the execution of the test. The replacement object has a known value that is specified by the test, which means that changes to the real object will not affect the result of the test.

The naming of different kinds of replacement objects may differ, but two widely used concepts are *stubs* and *mocks*. Both these replacement objects are used by the tested unit instead of some other module, but mocks also sets expectations on how it can be used by the tested module beforehand. [23]

As mentioned, the main reason for using mocks or stubs is often to make the test more robust to code changes outside the tested unit. Such replacement objects might also be used instead of calls to external services such as web- based APIs 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 implemented yet.

Code listing 2.4: Example of how mocking might make tests unaware of changes which breaks functionality.

```
class Cookie
    def eat_cookie
        return "Cookie is being eaten"
    end
end

class CookieJar
    def take_cookie
        self.cookies.first.eat_cookie()
        return "One cookie in the jar was eaten"
    end
end

def test_cookie_jar
    Cookie.eat_cookie = Mock
    assert CookieJar.new.take_cookie() == "One cookie in the jar was eaten"
end
```

Another type of replacement object is called *factory objects*[6]. This kind of object 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 that fake objects may use some shortcut that 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. Factory objects also provide a single entry point for constructing data, which makes it easier to maintain changes to the object structure. [23]

Bernhardt [21] 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 2.4. 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 would fail in a production environment. This is since we have mocked an object that no longer exists in the `Cookie` class.

Some testing frameworks and plug-ins detect replacement of non-existing methods, and give a warning or make the test fail in these situations. Another possible solution is to do refactoring of the code to avoid the need for mocks and stubs. [21]

### 2.2.2 Integration testing

Writing unit tests alone does not give sufficient test coverage for the whole system, since unit tests only assures that each single tested module works as expected. Since a unit test only assures that a single unit works as expected, faults may still reside in how the units work together. 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. The purpose of *integration tests* is to test several individual units together, in order to see if a larger part of the software works as intended.

There are several ways of performing integration testing, as well as arguments and opinions about the different ways. Huizinga and Kolawa [27] state that integration tests should be built incrementally by extending unit tests. The unit tests are extended and merged 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

[6] Also called fakes.

been integrated since the latest test run.

Pfleeger and Atlee [38] refer to the type of integration testing described by Huizinga and Kolawa 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, and the difference between the approaches is in which order the units are integrated.

The way of testing functionality of multiple units in the same way as unit tests, i.e. by input data into a module and then examine its output, is sometimes called *integrated testing*. Rainsberger [40] criticizes this way of testing. When testing multiple units in this way, one loses the ability to see which part of all the tested units that are actually failing, he claims. 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 tested. Rainsberger also claims that this fact makes developers sloppier, which increases the risk of introducing mistakes that goes unnoticed through the test suite. If this problem is solved by writing even more integration tests, developers will have less time to do proper unit tests and instead introduce more sloppy designs.

Instead of integrated tests, another type of integration tests called contract- and collaboration tests is proposed by Rainsberger. 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. This is called a 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 works together as expected, Unit A and Unit C will also work together as expected.

One may however argue that large parts of the criticism pointed out by Rainsberger is based on the fact that integrated tests to a large part are used instead of unit tests, rather than as a complement as suggested by Pfleeger and Atlee as well as Huizinga and Kolawa. One option could also be to use contract- and collaboration tests when doing top-down or bottom-up testing, rather than using integrated tests.

### 2.2.3 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. [27]

The most relevant part of system testing for the scope of this thesis is functional testing. The purpose of functional testing 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. [27]

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 user to the system, log in and perform different types of searches in the catalog. Different techniques needs to be used in order to narrow down the number of test cases.

### 2.2.4 Acceptance testing

*Acceptance testing* is an even more high-level step of testing than system testing. Its purpose is to determine whether or not the whole system satisfies the criteria agreed upon by the customer. This process may involve evaluation of the results from existing system tests as well as doing manual testing and assure that features for certain use-cases exist. Unlike lower levels of testing, acceptance level testing not only

assures that the system works but also that it contains the correct features. [9, 27]

We consider acceptance testing to be outside the scope of this thesis, but have chosen to include a brief introduction to it for completeness and understanding of other concepts later on.

## 2.3 Testing web applications

### 2.3.1 Typical characteristics of a web application

Web applications typically share multiple properties with traditional software, i.e. software that runs as an application locally on a single computer. Many languages can be used for writing traditional software as well as web applications, and the objective of doing software testing is typically the same. There is however some key differences and features exhibited by web applications. Mendes and Mosley [33] mentions the following specific characteristics for web applications:

- It can be used by a large number of users from multiple geographical locations at the same time.
- The execution environment is very complex and may include different hardware, web servers, Internet connections, operating systems and web browsers.
- The software itself typically consists of heterogeneous components, which often uses several different technologies. For example it may have a server component that uses Ruby and Ruby on Rails, and a client component that uses HTML, CSS and Javascript.
- Some components, such as parts of the graphical interface, may be generated at run time depending on user input and the current state of the application.

Each of these characteristics contributes to different challenges when doing software testing. While some of these characteristics pose testing issues that are outside the scope of this thesis, other issues are highly relevant. It may for instance be impossible to use the same testing framework for the server-side components as for the client-side components since the components are written in different programming languages. Another example is the complexity of the execution environment, which may result in defects that occur in some environments but not in others.

### 2.3.2 Levels of testing

Defining levels of testing requires greater attention in web applications than in traditional software due to its heterogeneous nature. For example, it is hard to define the scope of a unit during unit testing since it depends on the type of component. Defining levels of integration testing infer the same types of problem, since one may for example choose to either integrate different smaller server-side units, or test the integration between server-side and client- side components. [33]
Mendes and Mosley propose that client-side unit testing should be done on each page, where scripting modules as well as HTML statements and links are tested. Server-side unit testing should test storage of data into databases, failures during execution of controllers and generation of HTML client pages. Integration testing should be done testing the integration of multiple pages, for example testing redirections and submission of forms. System testing is done by testing different use cases and by detecting defective pointing of links between pages.

The book by Mendes and Mosley is rather old, and one could argue that some parts of the proposed testing approach would not fit into a modern web application. As one example, several modern web applications consist of one single page, which makes it impossible to test the integration of several pages, form submissions or to test link between pages. One may also fail to see the purpose of testing hyperlinks at all three testing levels, and would also claim that generation of such links is handled to a large extent by modern web frameworks and thus is outside our application scope. One could also claim that the interaction between different pages is covered by the use cases evaluated during the system testing.

We do however believe that unit testing client-side as well as server- side code respectively is important, and also that use-cases are a relevant approach when doing system testing.

### 2.3.3 Browser testing

As previously mentioned, a web application may be run on a variety of different operating systems and browsers which may cause defects specific to each environment. In order to discover defects of that kind, one must be able to test the web application in each supported environment.

The *browser test* is a kind of test where a web browser is controlled in order to simulate the behavior of a real user, by clicking on buttons and entering text into text fields. This approach tests the application in the same way as when conducting manual testing, which is an advantage as well as a drawback. On one hand, this way of testing is by far the most realistic way of testing an application. On the other hand it often tests a lot of code and it may take a lot of effort to test things thoroughly and to find the cause of bugs when tests fail. Because of this, browser tests are generally only used when conducting system testing.

Cross-browser compatibility is an important issue for many web applications. This basically means that the application should work in the same way regardless of which browser the user are using, at least as long as we have chosen to support the web browser in question. A browser test can often be run in multiple browsers, and can thus help finding browser-specific software defects.

Browser tests are slow, mostly due to the level of testing, but also due to the overhead of executing scripts, rendering pages and loading images. Another downside is that it requires a full graphical desktop environment and thus may take some effort and resources to set up on a server. One way of achieving higher test execution speed is to use a *headless browser*. A headless browser is a web browser where most functionality of modern browsers has been stripped away in order to gain speed. It typically does not have any graphical interface and therefore does not require a graphical desktop environment. [44]

One may argue that by running the tests in a headless browser is pointless since no real users uses a such browser. Therefore, we cannot be sure that the tested functionality actually would work in a real browser just because it works in a headless browser. It is of course also impossible to find browser-specific defects when using a headless browser. It would be possible to find bugs specific to the headless browser itself, but that would be rather pointless. Sokolov [44] however claims that using a headless browser covers the majority of all cases and can be used in the beginning of projects where cross-browser compatibility is less of an issue.

Fowler [24] explains that browser tests often tend to be bound to the HTML and the CSS classes of the tested page, since we need to locate elements in order to click buttons or fill in forms. This makes the tests fragile since changes in the user interface may break them, and if the same elements are used in many tests, we may need to go over large parts of our test suite.

One way of writing browser tests without making them bound to elements of a page is called the *page object pattern*. The basic principle is that the elements of a page are encapsulated into one or several *page objects*, which in turn provides an API that is used by the test itself. If we have a page with a list of items as well as a form for creating new items, we can for instance create one page object for the page itself, another for the form and yet another for the list of items. The page object for the page itself provides methods for accessing the page objects for the form and the list of items. The page object for the form may provide methods for entering data, and the page object for the list may provide methods for accessing each item in the list. [24]

## 2.4 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 that can be followed during software development. Such activities can for instance be defining 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. [46]

Extreme programming (XP) is a software development methodology created by Kent Beck, who published a book on the subject in 1999 [19]. 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 [46]. 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.

### 2.4.1 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 [31]. It should be noticed that the phrase *test-driven development* is often used in several other contexts where general practices for testable code are discussed. In this section, we consider the basics of TDD in its original meaning.

Madeyski [31] 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 [31].

The Test First principle is a central theme in TDD. Beck [20] describes the basics of TDD in a "mantra" called *Red, green, refactor*. The color references refer to colors used by often test runners to indicate failing or passing tests, and the three words refer 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 introduced 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 that 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 a efficient way, which are listed below.

- Developers need 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.

### 2.4.2 Behavior-driven development

Behavior-driven development (BDD) is claimed to originate from an article written by North [37], and is based on TDD [10]. This section is based upon the original article written by North.

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. Each test is described by a sentence, typically starting with the word *should*. For a function calculating the number of days left until a given 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".

Many frameworks use strings for declaring the sentence describing each test. Using strings rather than traditional function names allows us to use a native language, and solves the problem of naming tests. The string describing the behavior is used instead of a traditional function name, which also makes it possible to give a human-readable error if the module fails to fulfill some behavior. This 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 met a software analyst and realized that writing behavior-oriented descriptions about a system had much in common with software analysis. Software analysts write specifications and acceptance criteria for systems before development, which are used when developing the system as well as for evaluating the system during acceptance testing. They came up with a way of writing scenarios in order to represent the purpose and preconditions for behaviors in a uniform way, as seen below.

> *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* (a language that exists everywhere). 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.

## 2.5 Evaluating test quality

There are several metrics for evaluating different quality factors of written tests. One key property of tests is that they must be able to detect defects in the code, since that is typically the very main reason for writing tests at all. We have chosen to call this property *test efficiency*. Another quality factor is the performance of the tests, i.e. the execution time of the test suite. This section explains the properties of these quality factors, as well as their purpose.

Readability and ease of writing of tests is another important quality factor which are not mentioned in this section, since it is hard to measure in an objective way. It also often depends on the testing framework rather than on the tests themselves. Instead, we evaluate these properties for each chosen testing framework in section 4.1.

### 2.5.1 Test coverage

Test coverage[7] is a measure to describe to how large extent the program source code is tested. If the test coverage is high, the program has been more thoroughly tested and probably has a lower chance of containing software bugs. [11]

[7] Also known as *code coverage*

Code listing 2.5: A small example program for explaining different test coverage concepts.

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

Multiple different categories of coverage criterion exist. The following subsections are based on the chapter on test case designs in Myers et al. [35] 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*[8]. We illustrate this by using an example from Myers et al. [35]. For the program given in 2.5, we could achieve full statement coverage by setting `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 were an error, it would also go unnoticed.

Myers et al. finds that these properties make 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 exist.

### Branch coverage

In order to achieve full branch coverage[9], 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 2.5, 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[10] 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 2.5 with a combination of the input values `a = 2`, `b = 0`, `x = 4` and `a = 1`, `b = 1`, `x = 1`.

[8] Some websites, mostly in the Ruby community, refers to this as C0 coverage
[9] Sometimes also called decision coverage and sometimes referred to as C1 coverage.
[10] Also called predicate coverage or logical coverage.

Code listing 2.6: Example of a piece of code before mutation.

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

One interesting thing showed by the example above is 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 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 2.5, 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 2.5. In other words is it often practically impossible to achieve full multiple-condition test coverage.

### 2.5.2 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. [15, 30]

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 2.6 shows an example of a function that should return true if both arguments are odd. Several mutated versions of this example are shown in code listing 2.7. The goal of each mutation is to introduce a modification similar to a bug introduced by a programmer. [30]

All tests that we want to evaluate are run for the original program as well as for each mutant. If the test results differ, the mutant is *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 2.8, 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 that are not equivalent mutants. [15, 30]

Code listing 2.7: Mutated versions of code listing 2.6.

```
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
```

Code listing 2.8: 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
```

Lacanienta et al. [30] presents the results of an experiment where mutation testing was used in a web application with an automatically generated test suite. Over 4500 mutants were 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 were killed, 216 were equivalent and 2 were not killed.

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. [32] 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.

### 2.5.3 Execution time

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

Bernhardt [22] talks about problems related to depending on large tests that are slow to run. For example, he 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 2.4.1, 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 that is focused on a small testing part of the system, rather than many system tests that integrates with large parts of the system.

Haines [25] 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 that only test an isolated part of the system, independent from any third-party classes.

# 3 Methodology

This chapter outlines the general research methodology of this thesis, and explains different choices. The methodology of this thesis is generally based on the guidelines proposed by Runeson and Höst [41] for conducting a case study. An objective is defined and a literature study is conducted in order to establish a theoretical base. A case study is then conducted in order to evaluate the theory by applying it in a real-life application context. Finally, the result is analyzed in order to draw conclusions about the theory.

## 3.1 Literature study

The literature study is based on the problem formulation, and therefore focus 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 have also chosen to also consider blogs and video recordings of talks from developer conferences.

While blogs are neither published nor peer-reviewed, they often express interesting thoughts and ideas, and 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 give readers larger possibilities of leaving feedback on outdated information and fact errors. It also makes it possible to discuss the subject to a larger extent and give additional views on the subject.

Recordings of people speaking at developer conferences have similar properties when it comes to their content, lack of reviewing process and greater possibilities for discussion. One benefit is however that speakers at such conferences tend to be experts on their subjects, which might not be the case for a majority of all people writing blogs.

Blogs and talks from developer conferences 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 might also fail to be available in online databases until after their embargo period has passed [14, 43]. This can make it hard to publish up-to-date scientific articles about some web development topics, since the most recent releases of commonly used web frameworks are less than a year old [5, 13, 16].

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 tilt in this thesis towards agile approaches and best practices used by the open-source community.

## 3.2 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. When combined, the different parts give a good overview on the chosen testing approach as a whole.

### 3.2.1 Refactoring of old tests

There have been previous attempts to introduce testing of the application. Developers did however stop writing tests since the chosen approaches were found to be very cumbersome. At the start of this thesis, 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.

The TDD methodology is used during the case study. This methodology is based on the principle of writing tests before implementation of new features, and then run the tests iteratively during development. The test suite should pass at first. Then a new test should be implemented and the test suite should fail. The test suite should then pass again after the new feature has been implemented. This of course presupposes that existing tests can be run and give predictable results.

Due to this presumption, the first step of the case study is to make all old tests run. Apart from being a condition 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 is the fact that they run too slow in order to be continuously in a test-driven 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.

### 3.2.2 Implementation of new functionality

As previously mentioned, the commissioning body of this project wishes to implement support for planning resources in the GOLI application. During this part of the case study, the functionality is implemented and tests are written for new parts of the system as well as for refactored code.

The purpose of this part of the case study 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.

### 3.2.3 Analyzing quality metrics

In order to evaluate the tests written in previous parts of the case study, test coverage is used as a measure. The last part of the case study focuses on analyzing quality metrics of the GOLI 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 experience of using test coverage as a measure for test quality, and to produce a measurable output of the case study.

## 3.3 Software development methodology

An iterative development methodology is used during the case study and the implementation of new functionality. We would however not say that any specific development methodology is used in particular, since we merely have chosen a few basic ideas and concepts which occur in agile development methodologies such as Extreme Programming or Scrum.

The development is performed in cycles, where the results and the future development are presented and discussed with the commissioning body approximately once a week. Test-driven development methodologies is used in the strictest way possible. However, neither the TDD nor the BDD development methodology is followed strictly, as concepts originating from both these methodologies are used.

The apprehension of TDD and BDD in some blogs and talks tend to deviate slightly from their original definitions by Beck and North. During the development process of this thesis, the original definitions of

these methodologies are used.

## 3.4 Choices of technologies

Selection of the technologies to use in a certain project is one of the most important steps, since it affects the rest of the development process. In this case, the commissioning body of this thesis mainly gave the choice of frameworks for development since the existing software used certain programming languages and frameworks. The main server-side was written in Ruby using the Ruby on Rails framework, and the client- side code was written in CoffeeScript[1] using the Knockout.js framework.

For the choice of testing-related frameworks, we chose to look for frequently used and active developed open source frameworks. Technologies that are used by many people intuitively often have more resources on how they are used, and also have the advantage of being more likely to be recognized by future developers.

Active development is another crucial property of used frameworks. Unless a framework is updated continuously, it is likely to soon 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[2], which uses information from the Github and RubyGems websites, was consulted in order to find frameworks with mentioned qualities.

[1] CoffeeScript is a scripting language that compiles into Javascript.
[2] `https://www.ruby-toolbox.com/`

# 4 Results

## 4.1 Evaluation of tools and frameworks

One important question of this project is to find a set of relevant frameworks in order to work with testing in a Rails application, and gather experience with working with these. This subsection presents our evaluation of the different frameworks and tools used for testing.

### 4.1.1 Ruby testing frameworks and tools

Before the case study, Cucumber[1] and RSpec[2] were used as testing frameworks for existing tests. We evaluated these frameworks, as well as considered new frameworks in the beginning of the case study.

#### Cucumber

We worked with Cucumber during the first part of the case study since the major part of all existing tests was written using this framework. Cucumber is a framework for acceptance-level testing using the BDD methodology. Tests, called *features*, are written using a ubiquitous language as seen in example 4.1. The action for each line, called *step*, of the feature is specified using a *step definition*, as seen in example 4.2. [4]

One benefit in using Cucumber is that all steps are reusable, which means that code duplication can be avoided. However, it can be difficult to write the steps in a way that they benefit from this, and sometimes it also requires a lot of parameters to be passed in to each step. Cucumber also provides code snippets for creating step definitions, which avoids some unnecessary work.

Apart from these benefits, we found Cucumber tiresome to work with. The separation between features and step definitions makes it hard to get an overview of the code executed during the test, and it is often hard to find specific step definitions. We also experienced problems with the mapping between steps and step definitions, since the generated step definition simply did not match the written step in some cases. In cases where we wanted to use the same step definition, but used slightly different language (such as the steps on line 12 and line 14 in code listing 4.1), adjusting the regular expression to match both steps was sometimes hard.

The chosen level of testing is another big issue. We felt that using the TDD-methodology was cumbersome to do with system tests, since these take long time to execute and affect a much larger part of the software than the part we are typically working with when implementing new functionality. Considering these drawbacks, we decided to not continue the use of Cucumber for the subsequent steps of the case study.

#### RSpec

Just like Cucumber, RSpec also claims to be made for use with BDD [7]. In contrast, it only uses descriptive strings instead of a full ubiquitous language and can be used for writing isolated unit tests as

[1] `http://cukes.info/`
[2] `http://rspec.info/`

Code listing 4.1: Example of a Cucumber test.

```
1  Feature: creating new cookies
2    As a bakery worker
3    So that I can sell cookies to my customers
4    I want to create a new cookie object
5
6  Background:
7    Given a cookie type called "Chocolate chip"
8    And I have created one cookie
9
10 Scenario: create a new cookie
11   When I visit the page for creating cookies
12   Then I should see 1 cookie
13   When I create a new Chocolate chip cookie
14   Then I should see 2 cookies
```

Code listing 4.2: Cucumber step definition for the step on row 13 in code listing 4.1.

```
1  When /^I create a new (.+) cookie$/ do |cookie_type|
2    # Code for creating a new cookie
3  end
```

well as integration- and browser tests.

As an alternative to RSpec, we also considered Minitest[3]. Similar to RSpec, Minitest is popular as well as actively developed. In contrast to RSpec, it is more modular and claims to be more readable. It also claims to be more minimalistic and lightweight. By looking at code examples and documentation, we however found that the syntax of the tests for recent versions of both these frameworks seemed to be very similar. We did not find any considerable advantages of using Minitest over RSpec and therefore concluded that RSpec was a better option, since some of the existing tests already was written using this framework and migrating these would require additional work.

We found it quite straightforward to write tests using RSpec, and to use descriptive strings rather than function names for describing tests as seen in code listing 4.3. The plug-in rspec-mocks[4] were used in some situations where mocking was required, since the RSpec core package does not include support for this. One major drawback of rspec-mocks is that it allows stubbing non-existent methods and properties, which as discussed in section 2.2.1 can be dangerous. We worked around this by writing a helper for checking existence of properties before stubbing them.

### Factory girl

One important framework used was factory_girl[5], which is used for generating factory objects. As discussed in section 2.2.1, factory objects behave just like instances of model objects, but have several advantages.

As alternatives to factory_girl, we also considered Machinist[6] and Fabrication[7]. Machinist was discarded since it is no longer actively developed. Fabrication is actively maintained and quite popular, although it has far less downloads and resources than factory_girl. When looking at the documentation and examples, the two frameworks seem to be very similar. When searching for a comparison between the two, some people favor one of them while other people favor the other. Since we did not find any significant differences between Fabrication and factory_girl, we chose the latter since it is more popular.

[3] https://github.com/seattlerb/minitest
[4] https://github.com/rspec/rspec-mocks
[5] https://github.com/thoughtbot/factory_girl
[6] https://github.com/notahat/machinist
[7] http://www.fabricationgem.org/

Code listing 4.3: Example of RSpec tests for a module.

```
describe Math do
  describe '#minus' do
    it 'returns the difference between two positive integers' do
      expect(Math::minus(3, 1)).to(eq(2))
    end

    it 'returns the sum if the second integer is negative' do
      expect(Math::minus(5, -2)).to(eq(7))
    end
  end

  describe '#plus' do
    it 'returns the sum of two positive integers' do
      expect(Math::plus(1, 2)).to(eq(3))
    end
  end
end
```

Code listing 4.4: Example usage of the factory defined in code listing 4.5.

```
FactoryGirl.create(:cookie, diameter: 4.5, thickness: 0.5)
```

There are many advantages of using a factory framework such as factory_girl rather than just instantiate model objects by hand (i.e. just write `MyModel.new` in Ruby to create a new model instance). First of all, factory_girl automatically passes in default values for required parameters, so that we only need to supply the attributes needed in a particular test. Secondly, related objects are also created automatically, which typically saves as huge amount of work compared to manual creation of objects. Code listing 4.5 and 4.4 shows a factory definition and its usage. The name of the cookie and a new `Bakery` object is created automatically since we do not give them as parameters when using the factory.

We did initially have some issues with the creation of related objects since the factory_girl documentation did not cover working with document-based databases such as MongoDB in our case, but we eventually found out how to do this.

One feature that we felt was missing in factory_girl was the ability to specify attributes on related objects, for example to specify the name of a Bakery when creating a new Cookie. It is of course possible to first create a Bakery object and then creating a Cookie and pass in the created bakery object, but a shortcut for doing this would have been convenient in some situations. To our knowledge, Fabrication also lacks this feature.

### Other tools

TimeCop[8] was used in order to mock date and time for a test that was dependent on the current date and time. We do not have much experience from using this tool, but it worked as we expected for our specific use.

[8] `https://github.com/travisjeffery/timecop`

Code listing 4.5: A factory definition for a Cookie model.

```
FactoryGirl.define do
  factory :cookie do
    name 'Vanilla dream'
    diameter 1
    thickness 2
    bakery { FactoryGirl.build(:bakery) }
  end
end
```

### 4.1.2 Frameworks for browser testing

#### Selenium

The absolutely most widespread framework for browser testing is Selenium[9]. In fact, we have not even been able to find any other frameworks for running tests in real browsers, possibly since it takes a lot of effort to integrate with a large number of browsers on a large number of platforms. Selenium supports a number of popular programming languages, for example Java, C#, Python and Ruby, and also has support the majority of all modern web browsers on Windows as well as Linux and Mac OS. The interface used by newer versions of Selenium for interacting with browsers, WebDriver, has been proposed as an W3C Internet standard [45], which may indicate that support for Selenium is very likely to continue to be present in future browser versions. [17]

Note that there are two major versions of Selenium; Selenium RC and the newer Selenium 2, of which we have used the latter.

While Selenium is very widespread and seems to be the only option for running tests in real browsers, its API is on a rather low level. In order to fill in a string of text to a text field, we have to locate the label of the field using an XPath[10] function, then figure out its associated text element and sending out text as keystrokes to this element. Rather than writing helper methods for such functionality ourselves, we decided to use a higher-level framework.

#### Capybara

Capybara[11] provides a much higher level API for browser testing compared to Selenium, and besides from using it with Selenium it can also be used with drivers for headless browsers or for testing frameworks using mock requests. Tests can also be written using multiple testing frameworks, such as RSpec in our case.

We did not find any particular difficulties when using Capybara. Its API provides very convenient high-level helpers for every common task that we needed, and we did not experience any problems with any of them. An example of a browser test using Capybara is found in code listing 4.6.

#### SitePrism

As mentioned in section 2.3, one way of writing more structured browser tests is to use the page object pattern. We chose to use this pattern from the start rather than cleaning up overly complicated tests afterwards. While this pattern is perfectly possible to use without any frameworks, we found a framework called SitePrism[12] that provided some additional convenient functionality.

SitePrism makes it possible to define pages of an application by specifying its URL, as well as elements and sections of a specific page by specifying a CSS or XPath selector. An example of a page definition can be seen in code listing 4.7. Page objects automatically get basic methods for accessing elements,

[9] `http://docs.seleniumhq.org/`
[10] XPath is a language for selecting elements and attributes in an XML document, such as an website.
[11] `https://github.com/jnicklas/capybara`
[12] `https://github.com/natritmeyer/site_prism`

Code listing 4.6: A browser test written in RSpec using Capybara.

```
describe 'creating new cookies' do
  before do
    FactoryGirl.create(:cookie_type, name: 'Chocolate chip')
    login_as(FactoryGirl.create(:user))
  end

  it 'should be possible to create new cookies' do
    visit(new_cookie_path)

    fill_in('Amount', with: 1)
    select('Chocolate chip', from: 'Cookie type')
    click_button('Save')

    expect(page).to(have_content('The cookie has been created.'))
  end
end
```

Code listing 4.7: Page definition for a page with a list of cookie information.

```

class CookieRowSection < SitePrism::Section
  element :name, '.name'
  element :amount, '.amount'
end

class CookiePage < SitePrism::Page
  set_url('/cookies')

  element :heading, 'h1'
  element :new_cookie_button, 'button#createCookie'
  sections :cookies, CookieRowSection, '.cookieInfo'

  def create_cookie
    self.new_cookie_button.click()
  end
end
```

and allow us to define additional methods for each page or element. This allows us to use higher-level functions in our tests rather than locating elements manually in our Capybara tests. An example test using SitePrism is seen in code listing 4.8.

One drawback of using SitePrism can be that elements currently must be specified using selectors rather than having the possibility to select them based on text, which is possible with Capybara directly. It would on the other hand be easy to create a method for doing this using the helper methods provided by Capybara. One could also claim that using text for locating elements only makes sense for some specific types of elements, such as buttons and links, not for elements in general.

Overall, we found the page object pattern as well as SitePrism very convenient to work with.

### 4.1.3 Javascript/CoffeeScript testing frameworks

There are a few different frameworks for testing Javascript or CoffeeScript code. We had previously good experiences from working with Jasmine[13]. We also found that this framework seemed to be very popular,

[13] `http://jasmine.github.io/`

Code listing 4.8: Browser test using SitePrism a page object defined in code listing 4.7.

```
1
2  describe 'cookie information page' do
3    before do
4      FactoryGirl.create(:cookie, name: 'Vanilla dream', amount: 2)
5      FactoryGirl.create(:cookie, name: 'Apple crush', amount: 4)
6      @cookie_page = CookiePage.new
7      @cookie_page.load()
8    end
9
10   it 'should contain a list of cookies in alphabetical order' do
11     expect(page.heading.text).to(eq('List of cookies'))
12     expect(page.cookies.length).to(be(2))
13
14     cookie = page.cookies.first
15     expect(cookie.name.text).to(eq('Apple crush'))
16     expect(cookie.amount.text).to(eq('2'))
17   end
18 end
```

actively developed and had good documentation[14]. Code listing 4.9 shows an example of a Jasmine test written in CoffeeScript. The syntax of Jasmine is inspired by RSpec, which is another advantage since RSpec is used for the server side tests.

### Test runner

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 and worked natively using the Jasmine Ruby gem. A screenshot of this test runner is shown in figure 4.1.

The Jasmine test runner did however have multiple issues. First of all, it runs completely in the browser. Switching to the browser and reload the page in order to run the tests are not excessively problematic, but might be tiresome in the long run when using a test-driven development methodology.

A larger problem with the default test runner was that the asset handling, i.e. the process of compiling CoffeeScript into Javascript. Rails handles this compilation upon each reload when using the Jasmine gem. Since the used version of Rails re-compiles all assets upon page load if any file has 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 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 system process.

Another issue with the Jasmine test runner is that syntax errors are printed in the terminal rather than in the browser window where the test result is reported, which is confusing. In practice, syntax errors were often undetected for a long time, which required more debugging than necessary.

Due to these issues, we switched to the Karma[15] test runner. Karma originates from a master's thesis by Jína [28], which covers several problems with the Jasmine test runner as well as with some other Javascript test runners. Karma was designed to solve several of these issues and to be used with test-driven software methodologies. Tests 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 4.2. Re- compilation of source files and tests is very fast and we did not experience any stability issues. One minor drawback is however that pending tests is not displayed very clearly. This can be solved by using a different Karma test reporter.

[14] 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 testing framework and the tests are very well-written.
[15] `http://karma-runner.github.io/`

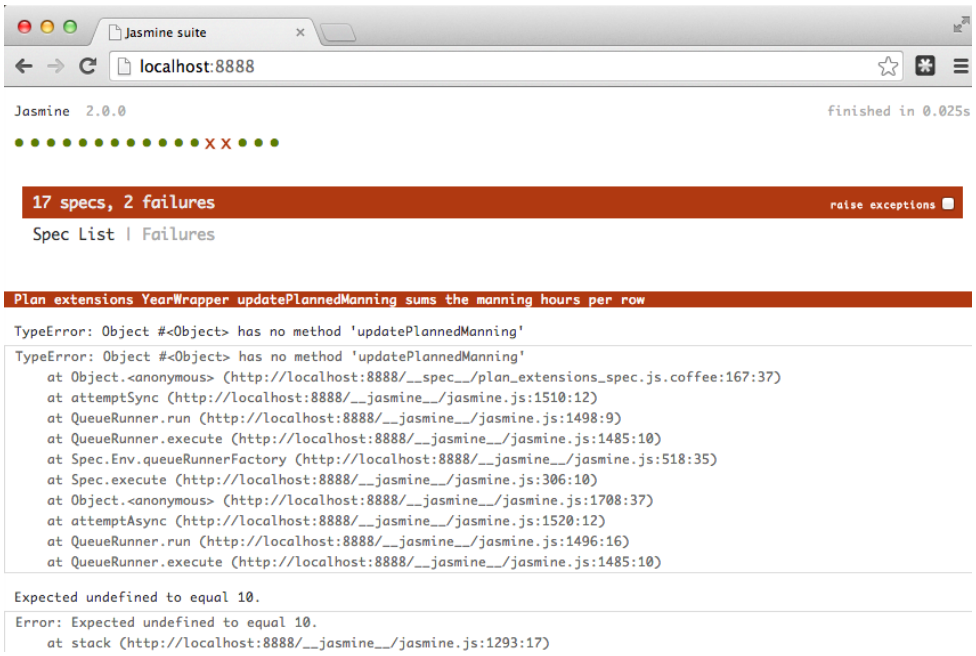


Figure 4.1: The test runner bundled with Jasmine.

Code listing 4.9: Example of Jasmine tests for a module (compare with code listing 4.3).

```
describe 'Math', ->
  describe 'minus', ->
    it 'returns the difference between two positive integers', ->
      expect(Math.minus(3, 1)).toEqual(2)

    it 'returns the sum if the second integer is negative', ->
      expect(Math.minus(5, -2)).toEqual(7)

  describe 'plus', ->
    it 'returns the sum of two positive integers', ->
      expect(Math.plus(1, 2)).toEqual(3)
```

It took some effort getting Karma to work with Rails, since Karma is written in Node.js and does not have any knowledge about which CoffeeScript files that exist in the Rails application. The task of finding the location of all such files became more complex since external Javascript libraries such as jQuery was loaded as Ruby gems, and therefore not even located inside the project folder. We ended up using a slightly modified version of a Rake script from a blog entry by Saunier [42] for bootstrapping Karma in a Rails environment. This script basically collects a list of filenames for all Javascript assets by using Rails internal modules for asset handling, and injects it into Karma's configuration file.

After the case study, we discovered another Javascript test runner called Teaspoon[16], which is built for Rails and can discover assets by default. We believe that this test runner looks very promising and may be more suitable for Rails projects than Karma. At the time of this writing, Teaspoon does not have support for the most recent version of Jasmine and therefore does not work with our test suite. We were thus unable to evaluate Teaspoon any further.

[16] `https://github.com/modeset/teaspoon`

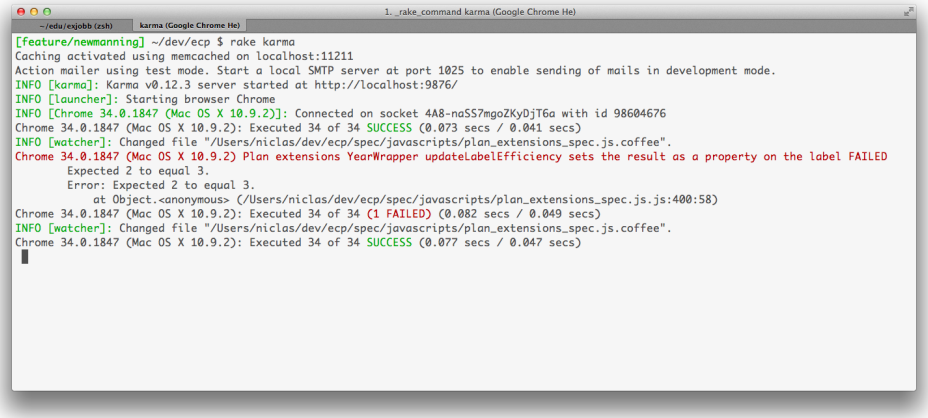


Figure 4.2: The Karma test runner.

### 4.1.4 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 2.5.1. 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 indicate that no such tool exists for Ruby at the time of this writing [1, 2, 3, 8].

We ended up using the SimpleCov[17] 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 4.3). [8]

#### CoffeeScript test coverage

For the client-side CoffeeScript, we used a plug-in for the Karma test runner called karma-coverage[18]. This tool basically integrates Karma with Ibrik[19], 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 that could be compiled correctly in the production code. The latest available release of Ibrik (version 1.1.1) also had major issues with certain constructs in CoffeeScript, which made the files impossible to analyze. These issues were 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 generates useful reports. As with SimpleCov, the coverage reports are produced as an interactive HTML report (see figure 4.4).

#### 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 were 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

[17] `https://github.com/colszowka/simplecov`
[18] `https://github.com/karma-runner/karma-coverage`
[19] `https://github.com/Constellation/ibrik`

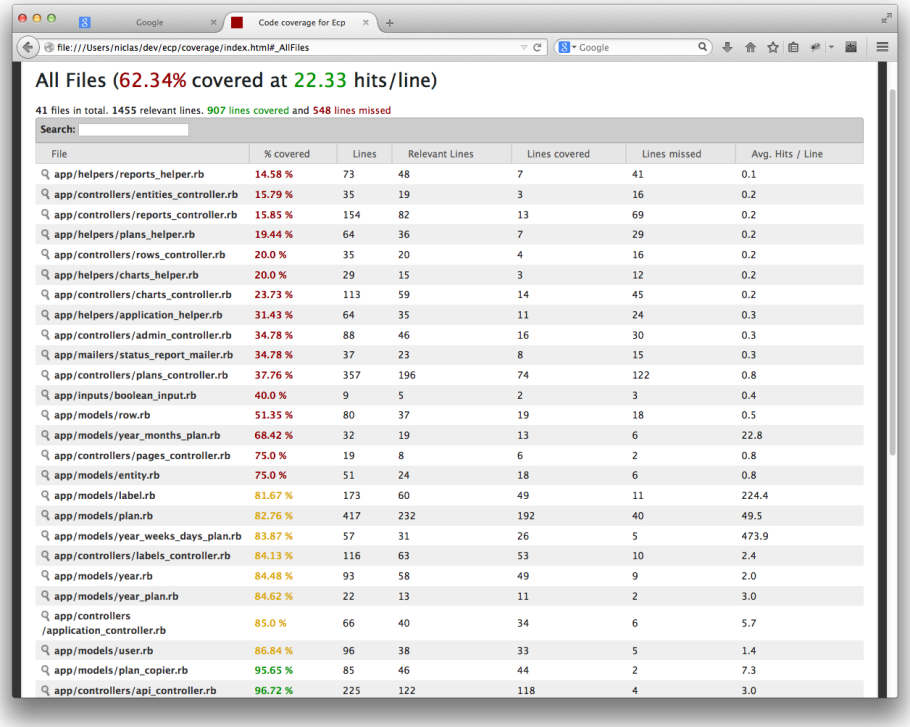


Figure 4.3: A test coverage report generated by SimpleCov.

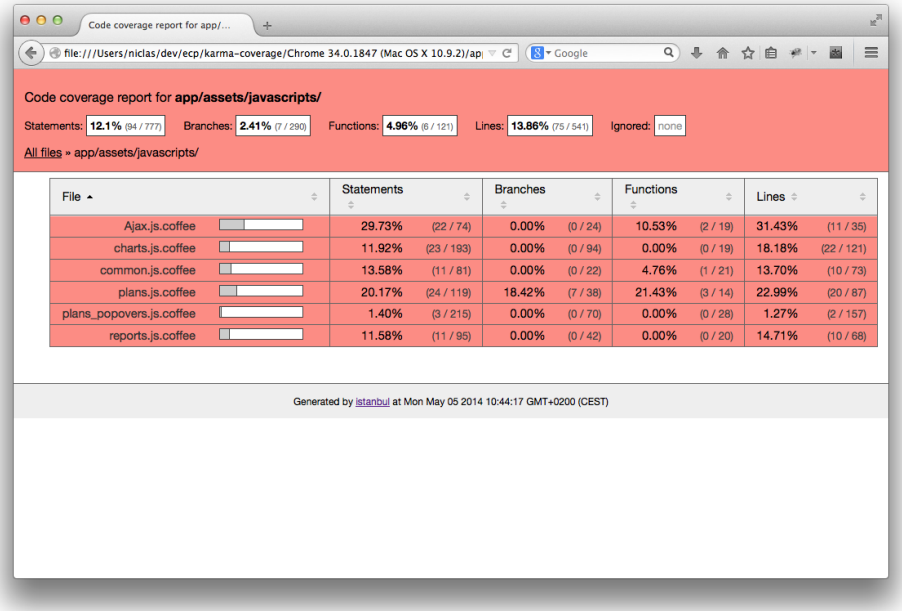


Figure 4.4: A test coverage report generated by istanbul-js using karma-coverage.

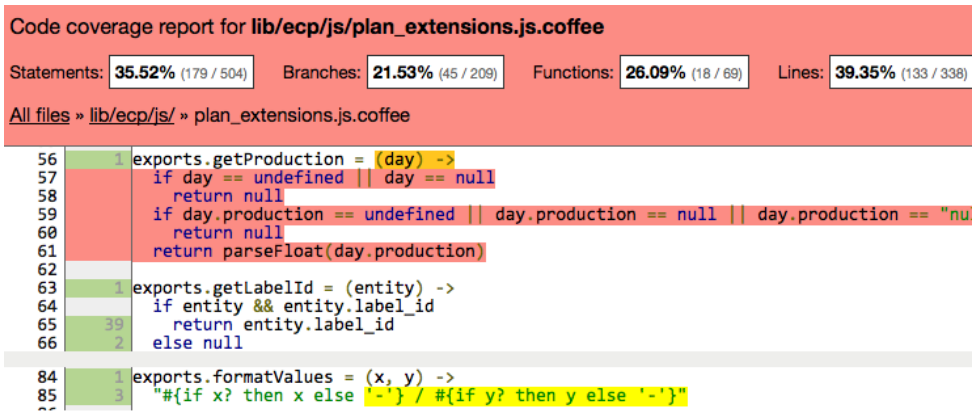


Figure 4.5: An excerpt from a coverage report generated using karma-coverage which demonstrates three different types of tested functions.

measure of how well tested the new and refactored 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 refactored code. Figure 4.5 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.

### 4.1.5 Mutation analysis

There exist a few different tools for mutation analysis of Javascript code. The ones we have found originate from academic research papers. Mirshokraie et al. [34] propose a solution that has been implemented as a tool called Mutandis [20]. Nishiura et al. [36] presents another approach that has been released as AjaxMutator[21]. Praphamontripong and Offutt [39] propose 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[22]. Using it for mutation testing tests written in CoffeeScript using Jasmine would probably be possible, but the effort of doing this is beyond the scope of this thesis.

For mutation analysis of Ruby code, we tried to use Mutant[23], which initially seemed very promising since it is actively developed and is used in several real-life software development projects. We were however unable to get Mutant to work properly, probably due to our recent versions of Ruby as well as Ruby on Rails. It was possible to start a mutation test, but the command never finished and no result was presented. Unfortunately we were unable to find any alternative tools for Ruby mutation testing.

[20] `https://github.com/saltlab/mutandis/`
[21] `https://github.com/knishiura-lab/AjaxMutator`
[22] `http://junit.org/`
[23] `https://github.com/mbj/mutant`

| Phase | No. of tests | Test coverage |
|---|---|---|
| Before the case study | 59 | 42.24 % |
| After the first part | 58 | 53.25 % |
| After the second part | 81 | 62.34 % |

Table 4.1: Statement test coverage for the RSpec unit- and integration tests at different phases.

| Phase | No. of tests | Statement coverage | Branch coverage |
|---|---|---|---|
| After the second part | 34 | 21.22 % | 10.14 % |

Table 4.2: Test coverage for the Karma client-side tests.

## 4.2 Chosen levels of testing

For the server side, we have chosen to mainly write integration tests, since the server side in this particular application is mostly concerned about loading and storing data. Most of these are integration tests of Rails controllers, which send a request with input data to the controller and asserts that correct data exist in database. We argue that it would be hard to write good unit tests for such logic without using an actual database and requests. Low-level unit tests without database access have instead been written for model instance methods, since these tend to have more well defined input and output data.

On the client-side, we have mostly focused on writing isolated unit tests, since most of all logic and calculations takes place on the client side in this particular application. A few large system-level browser tests was written in order to test the integration between the server- and client-side, as well as some of the integration between tested client-side units.

## 4.3 Test efficiency

As discussed in section 2.5, there are several measures for evaluating the quality of tests. This section presents the results of these different metrics for the GOLI application before and after different parts of the case study.

### 4.3.1 Test coverage

A quick overview of the results is presented in table 4.1, 4.2 and 4.3. More information is presented in the following sections. As discussed in section 4.1.4, 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 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 3.2.1) was focused on rewriting broken tests, since some of the existing tests were 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 tests were replaced by more fine-grained integration tests. The statement coverage of the GOLI application increased to 53 % after this part of the case study.

| Description | No. of tests | Test coverage |
|---|---|---|
| RSpec browser tests | 3 | 46.31 % |
| All RSpec tests | 84 | 66.81 % |
| Cucumber browser tests | 8 | 61.23 % |

Table 4.3: Statement test coverage including browser tests after the second 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 3.2.2) was focused on implementing new functionality while re-factoring old parts as needed and write tests for new as well as refactored 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.

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 refactored functions in general have 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 achieve 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.

#### Test coverage for 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 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.

### 4.3.2 Mutation testing

As discussed in section 4.1, we were unable to find any working frameworks for mutation testing our code. We did however do a manual test, where ten mutants were 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 mutants alive). Three of the living mutants mutated the same line in different ways.

For the client-side tests, eight out of ten mutants were 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`, which is not the case for many other programming languages[24].

[24]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 the case study | 59 | 20 s | 340 ms | 3.0 |
| After the first part | 58 | 8.3 s | 140 ms | 6.9 |
| After the second part | 81 | 7.6 s | 94 ms | 11 |

Table 4.4: Execution times of RSpec integration- and unit tests at different phases.

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

Table 4.5: Execution times of Jasmine unit tests.

## 4.4 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[25] 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.

A quick overview of the results is presented in table 4.4, 4.6, 4.5, and 4.7. The same results are also presented as text in the following sections.

### 4.4.1 Before the case study

The total running time of the 59 RSpec integration- and unit tests before the case study was 19.9 seconds. The average test execution time was 340 ms per test, which means an execution rate of three 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. It should however be noted 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 times are therefore misleading.

### 4.4.2 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 16 seconds.

### 4.4.3 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 refactored 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

[25] `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 the first part | 10 | 160 s | 16 s | 0.063 |
| After the second part | 8 | 155 s | 19 s | 0.052 |

Table 4.6: Execution times of Cucumber feature tests at different phases.

| 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 s | 0.26 |

Table 4.7: Execution times of RSpec browser 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, other developers added additional RSpec browser tests. At the latest revision of the code as of June 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. While these new tests are not strictly a part of the case study, we have chosen to include them in order to get a more robust value for the average time per test.

# 5 Discussion

This chapter discusses the results presented in chapter 4, and also mentions some of our experiences with the development process during the work with this thesis.

## 5.1 Experiences of test-driven development methodologies

I have striven for using test- and behavioral-driven development methodologies in the strictest possible way throughout this entire thesis. The purpose of this has been to gain experience of different advantages and drawbacks of these methodologies. In situations where I have found these methodologies to be unsuitable, I have however chosen to disregard some principles of these methodologies rather than try to apply them anyway.

Using a test-driven approach has two major advantages in our opinion. The most important is perhaps that it alleviates a sense of fear about whether or not the implementation really works as it is supposed to, and a fear that things may break when the code is refactored. The second important aspect is that the written code is self-testing; one can run a command in order to find out whether or not the code is in a working state, rather than testing it manually and see what happens. In particular, I found writing a failing test before fixing a software defect to be especially helpful. One can instantly see when the defect was fixed and therefore know when the work is done, and do not have to constantly test the software manually during development.

Another benefit from using the test-first principle is that one often think through the design before the implementation. The implementation has to be written in such way that it is testable, which makes the developer discover new ways of solving the problem and design the software better. In some cases this also reduces the amount of written code since we strive to minimize the number of cases that we have to test. Additionally, I personally find it tedious to write tests after the implementation is done. My opinion is that writing the tests first makes the development more fun than writing tests after implementation.

There is however no guarantee that a more testable design is better than a less testable design. For example, making it possible to test a module may require using of a large amount of fake objects such as mocks. It might also be required to split up the module into several sub modules that does not map very well to the real world and thus are difficult to understand. I believe that it is better to test on a higher level and perhaps discard the test-first principle in cases where making the code testable introduces more drawbacks than advantages to the overall design.

My main difficulty while using TDD was the *refactor*-part of the *Red, green, refactor* mantra. I found it hard to know whether or not code needed to be refactored, and to know how large part of the code to refactor. Some small changes required large parts of the code to be refactored in order for the code to be possible to unit-test. In some situations I found that the overhead of doing refactoring simply was so large that I decided not to do any refactoring at all. One reason for this may be that the old implementations have been written without testability or test-driven development in mind. The need for heavy refactoring would probably decrease over time if test-driven methodologies were used.

My experiences from using behavior-driven development principles are generally positive. I found that using descriptive strings for tests (rather than function names) forced me to split up tests into smaller parts, since each part must be possible to describe with a single sentence. It also felt natural to share

preconditions (such as *if the user is logged in*) for a set of tests using a string. One drawback is however that the full string for a test, including preconditions, tends to be very long. This can make it harder to perceive which of the tests that failed.

While using descriptive strings for describing tests and preconditions in general worked very well, while using a ubiquitous language when writing and reading tests felt unnatural. I experienced that the resulting test stories was hard to read and understand, and I could not see any gain from using it in this particular project. It might be worth considering for large software development projects with many roles and multiple stakeholders, but hardly for projects like the GOLI application where the developers are in charge of feature specifications as well as testing and implementation.

## 5.2 Test efficiency

### 5.2.1 Usefulness of test coverage

As seen in section 4.3.1, the test coverage for the existing RSpec integration tests was 42 % before this thesis, 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 was found when the coverage reports were inspected, in which certain files got full statement coverage even though there was not any tests for them.

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 that need testing, it is not possible to just exclude all such modules from the coverage report either. This is a symptom of the weakness of statement coverage, which is also previously discussed in section 2.5.1. 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.

Based on the weaknesses of statement test coverage, one may thus impeach its usefulness. I did however find that apart from the coverage score being a bad indicator on how well tested the code was, the statement coverage reports was highly usable for finding untested parts of the code. The nature of the 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, I 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 sounder metric for the overall testing level. Branch coverage of course also made it possible to discover a few more untested code paths.

### 5.2.2 Usefulness of mutation testing

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

My experiences with mutation testing however show 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 using higher-level integration tests rather than with unit tests. As mentioned in section 2.5.2, 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 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.

### 5.2.3 Efficiency of written tests

Since most of the implemented code consisted of extensions 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. I 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 during the different parts of the case study, which would not be the case if the test coverage of the newly implemented functionality were low, since the total SLOC[1] count has increased. Based on this fact, combined with the results of our subjective metric, I would argue that the efficiency of the tests written during this thesis is high in general. Mutation testing could be used in future in order to evaluate this conclusion further. It is however hard to tell whether or not the reason for this is because TDD methodology has been used.

## 5.3 Test execution time

### 5.3.1 Development of test execution time during the project

As seen in section 4.4, 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 were also refactored 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 refactoring 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, I could instead create only the objects needed for each specific test, instead of creating a huge amount of data that is not even used. RSpec also caches factory objects between tests that use the same objects, which gives some additional speed gain.

### 5.3.2 Execution time for different test types

In section 2.2, we discussed different levels of testing, and how this affects the amount of executed code and the test execution time. Table 4.5 shows that our isolated Jasmine unit tests runs at a rate of over 700 tests per second, while our system-level browser tests in table 4.7 runs at a rate of less than 0.3 tests per second. I would therefore say that the principle of writing many isolated unit tests and a few system 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 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 4.3, the Cucumber tests cover more code than the RSpec browser tests, and therefore take longer time to execute.

Another reason for the large difference in execution time 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. This is because a logged-in user is required in order to execute most tests, and the Cucumber tests does this by entering credentials into the login form for

[1] Source lines of code

each test. The RSpec test suite instead uses a test helper to log in the user before each test starts, except for when testing the login form itself. Yet another reason for the execution time difference might be the overhead of parsing the ubiquitous language used by Cucumber.

### 5.3.3 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. I 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. I will discuss some of my own experiences on this topic.

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 RSpec integration 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 before tests actually are run is considerably long, since Rails and related components are very large and takes some time to load. It would be possible to separate parts of the test suite from Rails and associated frameworks as mentioned in section 2.5.3, but I refrained from doing this since I considered it to affect the code structure in a bad way. Using tools like Zeus[2] can work around this problem, but I was unable to evaluate this tool due to incompatibilities with my Ruby version.

For browser tests, I believe that the most important thing is the ability to run these in some reasonable time. In this particular case, I 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 "reasonable 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.

## 5.4 Future work

During this thesis, I have discovered that the area of software testing is a very broad subject. The literature study disclosed several interesting topics among I 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 that has been considered in this thesis.

### 5.4.1 Ways of writing integration tests

As discussed in section 2.2.2, there are several opinions about integration tests and whether or not they should be written as integrated tests or not. The concept of contract tests proposed by Rainsberger [40] has been mentioned previously, but I 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?

### 5.4.2 Evaluation of tests with mutation testing

In section 2.5.2 and 4.1.5, the theory behind mutation testing is discussed, as well as some of my efforts on this subject. My results were however not very successful. I 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

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

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?

### 5.4.3 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 2.5.1. Another approach is to simply perform testing by using 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?

### 5.4.4 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 exist, 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?

# 6 Conclusions

This chapter presents some final thoughts and reflections on the results and summarizes the previously discussed conclusions. A summary of my proposed solution for testing a web application is also presented.

## 6.1 General conclusions

- Working with the chosen testing frameworks (RSpec and Jasmine) has worked very well for testing the GOLI application in particular. I believe that it would also work very well for testing most similar web applications.

- The experience of using test-driven development as well as some ideas originating from behavior-driven development has been pleasant. While this is a subjective opinion, I believe that this might be the case for many other software developers and for other projects as well. I have also found that the newly implemented functionality is well tested and that its test efficiency is high, which may be another benefit from using these methodologies. Writing tests using a ubiquitous language was however not beneficial for this particular application.

- The combination of many integration- and unit tests complemented by a few browser tests was successful for this project. It might however be hard to determine the best level of testing for certain functionality.

- The level of testing and the combination of different kinds of tests basically depends on the application. It is not possible to say that writing higher-level unit- or integration- test is generally worse than writing lower-level unit tests, nor the contrary. Using browser tests as the primary testing method is probably not the best solution for most projects, however.

- A test suite can speed up significantly by using factories to crate data before each test rather than manually create data for all tests when the test suite is initialized. While tests using the database are still not in the same magnitude of speed as low-level unit tests, they are still fast enough to be usable for using test-driven development methodologies.

- Using metrics for test efficiency such as test coverage is usable for finding parts of the code that lacks testing, in order to make tests better. I believe that using statement test coverage may work well practice, even if branch coverage is more helpful and returns more fair coverage percentages. The increased test efficiency could possibly lead to finding more software defects due to more efficient tests, but I have not found any defects by using test efficiency metrics in this project.

## 6.2 Suggested solution for testing a web application

For a web application with at least some amount of client-side and server-side code, I would recommend unit testing for the client as well as the server. In addition, system-level browser tests should be used. Most of the written tests should be on unit- and integration- tests, but the exact proportions depend on the application.

I would recommend applications that use Javascript or CoffeeScript on the client side to use the Jasmine testing framework together with the Karma test runner. For testing the client-side in Rails projects, Teaspoon could be an alternative to Karma, which is discussed in section 4.1.3.

A server-side written in Ruby on Rails could be tested using RSpec, with factory_girl for generating test data. Rails controllers can preferably be tested using higher-level tests, while model instance methods often can be tested using lower-level unit tests. This is discussed more thoroughly in section 4.1.1.

Selenium is highly useful for system-level tests. I would however recommend using a higher-level framework such as Capybara rather than using Selenium directly. The page object pattern should also be used. SitePrism is useful for this purpose. These frameworks are discussed further in section 4.1.2.

# 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] Cucumber official homepage, . URL http://cukes.info/. Accessed 2014-05-19.

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

[6] Software quotes, . URL http://www.software-quot.es/software-is-written-by-humans-and-therefore-has-bugs. Accessed 2014-05-20.

[7] RSpec official homepage, . URL http://rspec.info/. Accessed 2014-05-19.

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

[9] Acceptance testing, . URL http://en.wikipedia.org/wiki/Acceptance_testing. Accessed 2014-05-21.

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

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

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

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

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

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

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

[17] Selenium (software), . URL http://en.wikipedia.org/wiki/Selenium_%28software%29. Accessed 2014-05-19.

[18] Test automation, . URL http://en.wikipedia.org/wiki/Test_automation. Accessed 2014-05-21.

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

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

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

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

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

[24] Martin Fowler. Pageobject, . URL http://martinfowler.com/bliki/PageObject.html. Accessed 2014-05-22.

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

[26] 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.

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

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

[29] 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.

[30] 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.

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

[32] 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.

[33] Maria Emilia Xavier Mendes and Nile Spencer Mosley. *Web engineering*. Springer, 2006. ISBN 9783540282181.

[34] 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.

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

[36] 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.

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

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

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

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

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

[42] Sébastien Saunier. Angular + rails with no fuss. URL `http://sebastien.saunier.me/blog/2014/02/04/angular--rails-with-no-fuss.html`. Accessed 2014-05-13.

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

[44] Denis Sokolov. Headless functional testing with selenium and phantomjs. URL `http://code.tutsplus.com/tutorials/headless-functional-testing-with-selenium-and-phantomjs--net-30545`. Accessed 2014-05-23.

[45] Simon Stewart and David Burns. Webdriver. URL `https://dvcs.w3.org/hg/webdriver/raw-file/default/webdriver-spec.html`. Accessed 2014-06-18.

[46] 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.