Final thesis

Automated testing of a dynamic web application

Niclas Olofsson

April 28, 2014

Technical supervisor: Mattias Ekberg, GOLI AB

Supervisor: Anders Fröberg, IDA, Linköping University

Examiner: Erik Berglund, IDA, Linköping University

Linköping University
Department of Computer and Information Science

Abstract

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

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

Acknowledgments

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

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

Contents

1 Introduction

1.1 Background

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

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

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

1.2 Problem formulation

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

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

1.3 Scope and limitations

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

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

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

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

1.4 Conventions and intended audience

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

Code examples are written in Ruby[4] 2.1.1 since that is the primary language of this thesis, but with emphasis on being understandable by people without knowledge of this language rather than using Ruby-specific tools and practices. The built-in *Test::Unit* module is used for general test code examples in order to preserve independence of a specific testing framework to as wide extent as possible, although other testing frameworks are also mentioned and exemplified in the report.

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

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

2 Methodology

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

2.1 Introduction

The methodology of this thesis is generally based on the guidelines proposed by Runeson and Höst [24] for conducting a case study. An objective is defined and a literature study is conducted in order to establish a theoretical base. The theory is then evaluated by applying it in a real-life application context, and the result is analyzed in order to draw conclusions about the theory.

2.2 Literature study

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

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

Blogs and conference talks does have another benefit over articles and books since they can be published instantly. The review- and publication process for articles is long and may take several months, and also might not be available in online databases until after their embargo period has passed [6, 25]. This can make it hard to publish up-to-date scientific articles about some web development topics, since the most recent releases of well-used frameworks are less than a year old [1, 5, 8].

Utilized alternative sources are mainly relied upon recognized people in the open-source software community. Due to this, one might notice a skew in this report towards agile approaches and best-practices used by the open-source community.

2.3 Choices of technologies and frameworks

The choice of frameworks for development was mainly given by the constituent, since the existing software was written in Ruby on Rails with KnockoutJS and a MongoDB database.

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

features and bug fixes are released.

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

[1] https://www.ruby-toolbox.com/

3 Theory

3.1 Levels of testing

3.1.1 Unit-testing

Unit testing[1] refers to testing of small parts of a software. In practice, this often means testing a specific class or method of a software. The purpose of unit tests is to verify the implementation, i.e. to make sure that the tested unit works correct [22].

Since each unit test only covers a small part of the software, it is easy to find the cause for a failing test. Writing unit tests alone does not give a sufficient test coverage for the whole system, since unit tests only assures that each single tested module works as expected. A well- tested function for validating a 12-digit personal identification number is worth nothing if the module that uses it passes a 10-digit number as input [9].

Testability

Writing unit tests might be really hard or really easy depending on the implementation of the tested unit. Hevery [15] claims that it is impossible to apply tests as some kind of magic after the implementation is done. He demonstrates this by showing an example of code written without tests in mind, and points out which parts of the implementation that makes it hard to unit-test. Hevery mentions global state variables, violations of the Law of Demeter, global time references and hard-coded dependencies as some causes for making implementations hard to test.

Global states infers a requirement on the order the tests must be run in, which is bad since it is often non-deterministic and therefore might change between test runs. Global time references is bad since it depends on the time when the tests are run, which means that a test might pass if it is run today, but fail if it is run tomorrow.

The Law of Demeter[2] means that one unit should only have limited knowledge of other units, and only communicate with related modules [4]. If this principle is not followed, it is hard to create an isolated test for a unit which does not depend on unrelated changes in some other module. The same thing also applies to the usage of hard-coded dependencies. This makes the unit dependent on other modules, and makes it impossible to replace the other unit in order to make it easier to test.

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

Stubs, mocks and fakes

Another way of dealing with dependencies on other modules is to use some kind of object that replaces the other module. The replacement object has a known value that is specified in the test, which means that changes to the real object will not affect the test. The reasons for using replacement objects is often to make it more robust to code changes outside the tested unit. It might also be used instead of calls to

[1] Also called low-level testing, module testing or component testing
[2] Also called the principle of least knowledge

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

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

class CookieJar
    def take_cookie
        self.cookies.first.eat_cookie()
        "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
```

external services such as web-based API:s in order to make the tests run when the service is unavailable, or to be able to test implementations that depends on classes that has not been yet.

The naming of different kinds of replacement objects may differ, but two often used concepts are *stubs* and *mocks*. Both these replacement objects are used by the tested unit instead of some other module, but mocks sets expectations on how it can be used by the tested module on beforehand. [13]

Another type of replacement object are *fakes* or *factory objects*. This kind of object is typically provides a real implementation of the object that it replaces, as opposed to a stub or mock which only has just as many methods or properties that is needed for the test to run. The difference between a fake object and the real object is typically that fake objects uses some shortcut which does not work in production. One example is objects that are stored in memory instead of in a real database, in order to gain performance. [13]

Bernhardt [11] mentions some of the drawbacks with using mocks and stubs. If the interface of the replaced unit changes, this might not be noticed in our test. Consider the scenario given in 3.1. In this example we have written a test for the `take_cookie()` method of the `CookieJar` class, which replaces the `eat_cookie()` method with a stub in order to make the `CookieJar` class independent of the `Cookie` class.

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

Some testing frameworks and plug-ins detects replacement of non-existing methods and warns or makes the test fail if such mocks are created [11]. Another possible solution is to re-factor the code to avoid the need for mocks or stubs.

3.1.2 Integration-testing

Integration testing refers to the testing phase where several individual units are tested together. Since units tests only assures that a single unit works as expected, faults may still reside in how the units works together.

Huizinga and Kolawa [16] states that integration tests should be built incrementally by extending unit tests so that they span over multiple units. The scope of the tests is increased gradually, and both valid and invalid data is given into the integration tested system unit in order to test the interfaces between smaller units. Since this process is done gradually, it is possible to see which parts of the integrated unit

that is faulty by examining which parts have been integrated since the latest test run.

Pfleeger and Atlee [22] refers to this type of integration testing as *bottom-up testing*, since several low-level modules (modules at the bottom level of the software) are integrated into higher-level modules. Multiple other integration approaches such as *top-down testing* and *sandwich testing* are also mentioned. The difference between the approaches is in which order the units are integrated.

Rainsberger [23] criticizes one kind of integration tests, which he refers to as *integrated tests*. This refers to integration tests that are testing the functionality of multiple units in the same way as unit tests, i.e. by input data and examine the output. When testing multiple units in this way, one loses the ability to see which part of all the tested units that are actually failing. As the number of tested units rises, the number of possible paths will grow exponentially. This makes it hard to see the reason for a failed test, but also makes it very hard to decide which of all this paths that needs to be testes. Integrated tests therefore puts much less pressure on the tested functions compared to unit tests.

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

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

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

3.1.3 System-testing

System testing is conducted on the whole, integrated software system. Its purpose is to test if the end product fulfills specified requirements, which includes determining whether all software units (and hardware units, if any) are properly integrated with each other. In some situations, parameters such as reliability, security and usability is tested.[16]

The most relevant part of system testing for the scope of this thesis is functional testing. The purpose of this is to verify the functional requirements of the application at the highest level of abstraction. In other words, one wants to make sure that the functionality used by the end-users works as expected. This might be the first time where all system components are tested together, and also the first time the system is tested on multiple platforms. Because of this, some types of software defects may never show up until system testing is performed.[16]

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

3.2 Test methodologies

3.2.1 Test-driven development

Test-Driven Development (TDD) originates from the Test First principle in the eXtreme Programming (XP) methodology, and is said to be one of the most controversial and influential agile practices [18].

Madeyski [18] 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 [18].

The Test First principle is a central theme in TDD. Beck [10] describes the basics of TDD in a "mantra" called *Red/green/refactor*. The color references refers to colors used by often test runners to indicate failing or passing tests, and the three words refers to the basic steps of TDD.

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

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

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

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

3.2.2 Behavior-driven development

Behavior-Driven Development (BDD) is claimed to originate from an article written by North [21], and is based on Test-Driven Development (TDD) [2], which was discussed in section 3.2.1. This section is based upon the original article by North [21].

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

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

Code listing 3.2: A small example program

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

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

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

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

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

3.3 Evaluating test efficiency

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

3.3.1 Test coverage

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

Multiple different categories of coverage criterion exists. The following subsections are based on the chapter on test-case designs in Myers et al. [20] 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*. We illustrate this by using an example from Myers et al. [20]. For the program given in 3.2, we could achieve full statement coverage by setting `a = 2`, `b = 0` and `x = 3`.

This requirement alone does not verify the correctness of the code, however. 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

[3] Also known as *code coverage*

are less than zero. If this was an error, it would also go unnoticed.

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

Branch coverage

In order to achieve full branch coverage[4], tests must make sure that each path in a branch statement is executed at least once. This means for example that an if-statement that depends on a boolean variable must be tested with both `true` and `false` as input. Loops and switch-statements are other examples of code that typically contains branch statements. In order to achieve this for the code in 3.2, we could create one test where `a = 3`, `b = 0`, `x = 3` and another test where `a = 2`, `b = 1`, `x = 1`. Each of these tests fulfills one of the two if-conditions each, so that all branches are evaluated.

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

Condition coverage

Condition coverage[5] means that each condition for a decision in a program takes all possible values at least once. If we have an if- statement that depends on two boolean variables, we must make sure that each of these variables are tested with both `true` and `false` as value. This can be achieved in 3.2 with a combination of the input values `a = 2`, `b = 0`, `x = 4` and `a = 1`, `b = 1`, `x = 1`.

One interesting this about the example above it that logical 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 imply branch coverage, even if that is sometimes the case. A combination of the two conditions, *decision coverage*, can be used in order to solve make sure that the implication holds.

Multiple-condition coverage

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

For the code given in 3.2, this requires the code to be tested with $2^2 = 4$ combinations for each of the two decisions to be fulfilled, eight combinations in total. This can be achieved with four tests for this particular case (`x a b` values `2 0 4` and `2 1 1` and `1 0 2` and `1 1 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. In other words can it is typically impossible to achieve full multiple-condition test coverage.

3.3.2 Mutation testing

An alternative to draw conclusions from which paths of the code that is run by a test, as done when using test coverage, is to draw conclusions from what happens when we modify the code. The idea is that if the code is incorrect, the test should fail. Thus, we can modify the code so it becomes incorrect

[4] Sometimes called *decision coverage*
[5] Also called *predicate coverage* or *logical coverage*

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

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

Code listing 3.4: Mutated versions of 3.3

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

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.[7, 17]

There are numerous ways of creating mutations to be used in mutants. One could for example delete a method call or a variable, exchange an operator for another, negate an if-statement, replace a variable with zero or null-values, or something else. Code listing 3.3 shows an example of a function which should return true if both arguments are odd. Several mutated versions of this example is shown in code listing 3.4. The goal of each mutation is to introduce a modification similar to a bug introduced by a programmer.[17]

All tests which we want to evaluate are run for the original program as well as for each mutant. If the test results differ, the mutant is considered to be *killed*, which means that the test suite has discovered the bug. Some mutants may however have a change that does not change the functionality of the program. An example of this can be seen in 3.5, where two variables are equal and therefore not affected by replacing one of them with the other. This is called an *equivalent mutant*. The goal is to kill all mutants which is not equivalent mutants.[7, 17]

Lacanienta et al. [17] presents the results of an experiment where mutation testing was used in a web application with an automatically generated test suite. Over 4500 mutants was generated, with a test suite of 38 test cases. Running each test case for each mutant would require over 170000 test runs. A large part of the program was therefore discarded and the evaluation was focused on a specific part of the software. which left 441 mutants. 223 of these was killed, 216 was equivalent and 2 was not killed.

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. [19] 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.

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

```
1  def odd?(x, y)
2      x = y
3      (x % 2) && (y % 2)
4  end
5
6  def equivalent_mutant(x, y)
7      x = y
8      (x % 2) && (x % 2)
9  end
```

3.4 Other test quality factors

3.4.1 Execution time

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

Bernhardt [12] talks about the problem with depending too much on large tests which are slow to run. He also mentions how the execution time of a test can increase radically as the code base grows bigger, even if the test itself is not changed. If the system is small when the test is written, the test will run pretty fast even if it uses a large part of the total system. As the system gets bigger, so does the number of functions invoked by the test, thus increasing the execution time.

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

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

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

Haines [14] also emphasizes the importance of fast tests, and proposes a way of achieving this in a Ruby on Rails application. The basic idea is the same as proposed by Bernhardt, namely separating business logic so it is independent from Rails and other frameworks. This makes it possible to write small unit tests which only tests an isolated part of the system, independent from any third-party classes.

3.4.2 Readability and ease of writing

wip

4 Approach

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

4.1 Hypothesis

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

4.2 Case study

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

4.2.1 Refactoring of old tests

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

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

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

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

4.2.2 Implementation of new functionality

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

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

4.2.3 Increasing test coverage

In order to evaluate the tests written in previous parts of the case study, code coverage is used as a measure. The last part of the case study focuses on analyzing parts of the application that is untested or very weakly tested, and write tests for them. We also evaluate the tests written in previous parts of the case study and complement them if needed.

The purpose of this part is to get a more solid experience of using test coverage as a measure for code quality, and to produce a measurable output of the case study. We also look at how TDD methodology works in an situation where the functionality is already implemented without tests, as opposed to using it when implementing new functionality.

5 | Results

This chapter presents the results of the case study.

6 | Discussion

This chapter discusses the results.

7 | Future work

This chapter discusses what could be done on the subject in a future study.

References

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[15] 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.

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

[17] 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.

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

[19] 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.

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

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

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

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

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

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