Mika Paulasaari

Tools for Code Quality in Front-end Software Development

Helsinki Metropolia University of Applied Sciences

Master of Engineering

Information Technology

Master's Thesis

13 April 2018

Helsinki Metropolia University of Applied Sciences

Author(s) Title Number of Pages Date	Mika Paulasaari Tools for Code Quality in Front-end Software Development 53 pages 13 April 2018
Degree	Master of Engineering
Degree Programme	Information Technology
Specialisation option	Networking and Services
Instructor(s)	Ville Jääskeläinen, Head of Master's program (Metropolia)

This thesis was created to identify ways to improve and enforce code quality using automated tools in front-end software development projects. The problems caused by a bad code quality vary from bugs to project delays and in general cause issues with the project timeline and budget. Finding ways to improve the code quality is important so that in the future the software development projects can be produced at the highest possible quality and that they offer the best value for the clients. Producing a high-quality code would also improve the image of the company and help to create long-lasting partnerships with clients.

The study consists of a theory section, which defines good code quality attributes in depth. The following chapter contains two case study analyses conducted on past large-scale software development projects. The projects included some tools to improve the quality of the code, but they both also failed in some quality areas. These analyses were utilized to first find out the common pain points in software development projects, which reduced the code quality and secondly to help to identify methods that could solve these issues.

The scope of the thesis includes only front-end software development related coding issues and tools. Some non-automated methods and processes are identified in the theory section, but the main focus was to identify automated tools that are relatively easy to configure and use throughout different projects. For the case study scope the two past projects were chosen.

The results of the thesis show that issues, which could be solved by enforcing the code quality with automated tools, caused many of the problems in the past projects. The set up of the projects determined a lot of the type of issues that arose, but most of them could have been avoided with a better planning and implementation of the quality enforcement tools. The results and suggestions chapter consists of a list of tools and guidelines for how to use them to solve the most common coding related challenges.

Keywords	coding, quality, front-end, software, development, testing

Table of Contents

List of Abbreviations

API	Application Programming Interface
AST	Abstract Syntax Tree
ATDD	Acceptance Test-Driven Development
CI	Continuous Integration
CLI	Command Line Interface
CSS	Cascading Style Sheets
E2E	End-to-end
ES	ECMAScript
HTML	Hyper Text Markup Language
JS	JavaScript
JSON	JavaScript Object Notation
OOP	Object-Oriented Programming
PHP	PHP: Hypertext Preprocessor
QA	Quality Assurance
RTF	Rich Text Format
SASS	Syntactically Awesome Style Sheets
SCSS	Sassy CSS
TDD	Test-Driven Development
UI	User Interface
URL	Uniform Resource Locator
YACC	Yet Another Compiler-Compiler
YAML	YAML Ain't Markup Language

1 Introduction

This master's thesis focused on the tools that enforce good code quality in front-end software development. The aim of this study was to suggest recommendations for how to design, implement and administer tools for code quality in large-scale front-end software development projects.

The company for which this thesis was produced is Idean Enterprises and is abbreviated Idean for future reference. Idean is a design agency specializing in customer experience, user experience, user interface, and service design. Idean has a global network of 11 studios in 5 countries and over 230 digital strategists, designers, and developers. Idean was founded in 1999 and the current main headquarters is located in Palo Alto, California. In 2017 multinational consulting corporation CapGemini acquired Idean and made it part of CapGemini Consulting.

Large-scale software development projects are difficult to manage if no processes exist that enforce good code quality and best working practices. If there are no processes set up to help enforce good coding practices the responsibility falls on the individual programmers who write the code. Even the most experienced programmers make mistakes sometimes and the earlier these mistakes are noticed the easier it is to fix them. At Idean there have been some large-scale projects that have suffered from the lack of processes to enforce good code quality. A common example of the problems that have arisen are software bugs, which mean that programmers had to invest time in debugging and fixing the bugs instead of writing new features. There was clearly a demand to identify the pain points of the development work and to find solutions that help the programmers' daily work and result in better quality products. This thesis aims to answer the following question:

What tools to choose for maximizing code quality in front-end software development projects and how to use them?

This thesis includes a study to identify what good code quality means. Also included is a list of tools that are available and explanation on how to use them effectively for best results. In addition, the study includes case study analyses of two of Idean's past projects to give an understanding of the technical pain points in the projects. Based on

those findings, recommendations for what tools to use to maximize code quality are suggested. The outcome of the study is a code quality tools guideline for Idean's internal use in projects. To stay focused on the main topic, the scope of this thesis was determined to include only front-end coding related quality issues. Furthermore, the number of case studies used was limited to two.

This thesis has been divided into 6 sections. The first chapter introduces the problem and explains why it is important to find a solution for it. The second chapter explains the methods and material used for the research and for developing this study. The third chapter analyzes existing knowledge and background theory in code quality. The fourth chapter explains the current situation and shows the analysis of the case studies and the revealed findings. The chapter five combines the information from the theory part and applies that to the framework of the case studies. The sixth and the final chapter conclude the thesis and shares final thoughts about the project.

2 Methods and Material

To develop the solution, first, existing knowledge about code quality and how to improve and enforce it will be explored. Secondly, case studies of Idean's past projects, that fit the characteristics of a large-scale front-end software development project, will be analyzed. The analysis will give a good understanding of what the pain points are and where the most common mistakes happen in such projects. Thirdly, based on these findings, recommendations for the tools to use to maximize code quality will be suggested.

The following research method was utilized for this study.

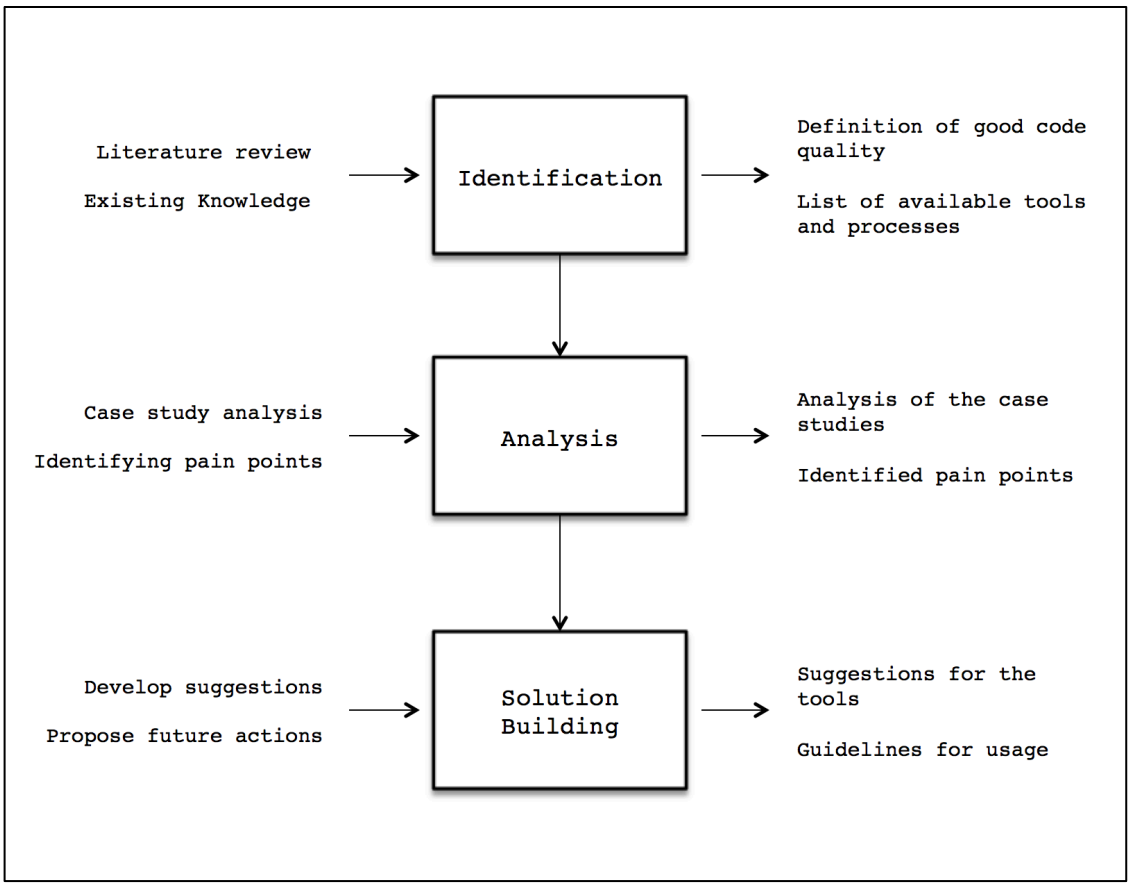


Figure 1. The research method for the study.

As shown in Figure 1, the study started with an identification phase, in which existing knowledge about code quality and the ways to enforce it were investigated mostly through literature reviews. The theory part is a high-level study about the existing technologies, without going too much into details of how they can be implemented. The

outcome of this phase will be a good definition for code quality and a list of tools and methods to enforce good code quality.

The study continues to the case study analysis phase in which the selected projects will be analyzed using the case study analysis method. In this phase the case studies are analyzed in depth and the pain points will be identified and alternative solutions will be proposed. The outcomes of this phase will be summaries of the case studies, collections of the identified issues within the cases, the alternative course of actions for the identified challenges and recommendations for the best solutions to solve the issues.

Finally, in the actions phase, the solutions from the theory part are coupled with the identified issues from the case studies to introduce a set of proposed solutions. In this phase, the methods and tools from the theory section are put into the framework of front-end software development and the technologies are explained in more detail together with concrete examples of how to use them. The outcome of this phase will be a comprehensive list of tools together with instructions for enforcing code quality in front-end software development projects.

Case Study Analysis

Case study analysis is the main research method used in this thesis and it will function as a guide for how to develop the solution for the research problem. A case study is usually a summary of a business case that contains a choice to be made to solve a challenge or a problem. It can be based on a situation that has actually happened as represented, or parts of it might have been altered for privacy concerns. In most case studies, a choice must be made to achieve a conclusion, even if the decision is to not to change anything. (Laudon, 2005)

A case study analysis consists of more than just a summary of the situation. According to Laudon, the analysis' main purpose is to find the key problems, evaluate alternative solutions for the issues and propose applicable outcomes (2005). A case study analysis can be executed in several ways, but for this thesis, the following steps have been selected:

1. Summarize the case and its key facts.

2. Identify the main challenges or problems.

3. Evaluate and determine alternative solutions to the issues.

4. Recommend the best solutions.

These steps are explained in more detail in the following sections.

2.1 Summarizing the Case

The first step is to summarize the case and its key facts. The case situation is explained at the level of detail that the reader will get a good understanding of the case. Important facts about the case are identified and described in this step. The facts and figures presented about the case can be expected to be true, although any identifying information might be obscured for privacy reasons. Statements and interviews from the individuals involved in the case together with other facts are presented to gather data about the case. Certain assumptions have to be made based on the data and the correctness of the conclusions is dependent on the view of the person conducting the case study.

2.2 Identifying the Key Issues

In the second step, the main challenges and issues are identified based on the facts provided in the first step. Often the cases include several issues and they must be evaluated based on their importance. The main focus will be set around the major issues as they have the biggest impact on the case. The issues are summarized in few sentences with an explanation on how the issue affected the outcome of the case. The explanations can vary from technical difficulties to human-centric problems and management issues. Information system issues are commonly caused by a combination of technology, management and organizational challenges and when analyzing the issues the key point is to identify the nature of the problems. Often the issues occur because of a combination of these factors.

Management Issues

Management issues can stem when the management is not functioning at a necessary level to acknowledge the lack of performance in the organization. This can be caused by the management not having sufficient information about the situation or from the lack of actions to address the problems.

Technological Issues

Technological challenges might arise from the infrastructure of the organization. The hardware, software, network, file systems and other technical factors might not be working at the level required to successfully complete the tasks. There might also be lack of documentation and guidelines for how to use the technology efficiently.

Organizational and Human Factor Issues

The work environment and its culture, organization structure, work processes and groups and communication between the internal and external groups can cause organizational and human factor issues. Depending on the nature of the work there might be a need to follow government regulations. There might also be other determining external factors, such as customer needs or supplier challenges.

2.3 Identifying and Evaluating Alternative Solutions

The third step will list the alternative courses of action that could have been taken to solve the issues. In the case of technological issues this could mean upgrading the infrastructure or acquiring different software and tools, as there might already exist better alternatives for the current technologies. Making the change might require also changes in the work processes, organizational structure or management conventions, depending on the impact of the proposed solution.

In some cases there might be constrains on what the organization can actually do and the technological solutions might not be altered. In some cases the evaluated solution might carry too high expenses or it might be too challenging to implement. Therefore every proposal should take in account constrains of the organization and evaluate how viable solutions they are for the case. Ideally the proposals should be evaluated from

their technical, financial and operational point of views to determine their feasibility. If there are no clear facts to support the proposal, but its feasibility is based on assumptions, it should be mentioned along the decision.

2.4 Recommending the Best Solutions

The final step of a case study analysis will recommend the best solutions together with explanations on why they were chosen. It might also include information about the other alternatives and why the proposed solution was picked over them. This step should give the reader a good understanding of the thinking process behind picking the solutions and what assumptions were used to come to the final conclusion. In many cases, there might not be only one perfect solution, but many of them carry different benefits and risks, which have been evaluated during the process.

3 Attributes of Good Code Quality

Code quality is a combination of various attributes and conditions, which depend on the used business case. The following attributes are some of the most common ones for defining code quality, but there are tens of more attributes that can be used in addition depending on the code that is being evaluated (Microsoft, 2009).

Maintainable

A maintainable code is written in a manner that other people are expected to read and understand it. Ways to achieve maintainable code is to keep the code simple, short and consistent, use code comments, follow coding best practices and have meaningful naming conventions for variables, functions and components. The code should not be overly complicated, but anyone who has to work on the codebase must be able to understand the context in order to make changes. Therefore the main focus should be on the programmers that have to read and maintain the code in the future, sometimes even without any prior knowledge about the codebase. Maintainability can also sometimes be thought as readability, or how easy it is to read the source code.

Documented

Similarly to maintainability, a well-documented code is easier to understand and maintain by other programmers. Documentation can be used to describe for example how the application should be used, how the code works or what kind of standards the code is following. Software documentation consists usually of external documents to describe the software at a high level, and of embedded documentation in the source code, which explains the inner workings of the software at more detailed level.

Well-structured

Good structure for code means evaluating how the different parts of the software codebase are interacting with each other. In large organizational software codebases a good structure is especially important because it has a direct correlation to how much time and money it will take to develop, test, maintain and extend the system.

Tested

A code should have thorough tests to prevent any unwanted bugs from being introduced. Having well-written tests against the code will also enforce maintainability and extensibility as the programmers can modify the code with ease of mind knowing that the tests will more likely show if something gets broken. Testing can be performed at many different levels in the software development process. Each level has been designed to detect different types of issues.

Reliable

A code can be considered reliable when it works as expected and does not fail even in edge case situations. Reliable applications have fewer bugs and downtime and are therefore better for the business.

Efficient

A cleanly written code is often faster to execute than poorly written code. Maintenance is easier for well-written code and because it is correctly done and refactored it is also faster to execute.

Follows Standards

The code should follow any rules and regulations set by the client organization. The other programmers will understand easier the code and its maintenance takes less effort when it is following common standards. Following coding standards creates more consistent and uniform code regardless of each programmer's personal styles.

Extensible

An extensible code can adapt well to changing conditions and adding new functionalities to it is easy. Highly extensible code correlates to less time that the programmers have to spend when adopting new functionality to it.

However, not all of the listed quality attributes can be enforced by using automated tools, but some of them can only be executed and evaluated manually by a program-

mer familiar with the code. This thesis will concentrate on the attributes that can be automatically executed and enforced by utilizing different tools. The following chapters will further explain these types of attributes and their importance to code quality.

3.1 Software Documentation

Documentation for the source code should be thorough enough, but having overly detailed documentation might become too burdensome to maintain. Examples of common technical software documents are Application Programming Interface (API) documentations and `.README` files. The programmers responsible for the API development usually create different types of how-to guides and general overview documents about the APIs. The people utilizing this kind of documentation include other programmers, testing engineers and end-users of the application. Having up-to-date documentation is very important no matter what type the application is, as the architectures and program structures change over time (Barker, 2003).

Software documentation within the source code can be achieved with automated tools that generate the code documents without requiring much extra effort from the programmer. Examples of such tools are Javadoc, Doxygen, Ndoc and JSDoc. Each of them usually supports one specific programming language or framework. These tools can parse comments in the source code and create reference manuals in various easily accessible formats, such as hypertext markup language (HTML) files.

Auto-generated documentation can be appealing to programmers for many reasons. Since they are parsed automatically from the source code the programmer does not have to use any other tools to create the documentation and the process if therefore very effortless. Also, as the documentation is being written at the same time as the code itself, it is easy to keep it up to date. A downside of this type of documentation is that only a person with programming skills can create and edit them. In order to update the documentation, the code would have to be compiled or built again. However, it is possible to automate the updating process as well, which makes it easier.

3.2 Code Testing

Software testing can be approached from many different angles. Code reviews, walkthroughs and syntax inspections are common methods used in static testing. The other main testing approach, called dynamic testing, means executing the program and

running it against predefined test cases. Static testing is usually utilized as proofreading to find written errors in the code and it can be used together with text editors to check the source code for structure, syntax and data flow for static analysis. Static testing is often applied while the code is being written as dynamic testing happens when the program, or parts of it, is being executed. When dynamically testing specific sections of the code, such as modules or functions, it is common to execute the program in a debugged environment and utilize stubs and drivers in the process. Another way to differentiate between static and dynamic testing methods is to think of static testing as a verification and dynamic testing as a validation. Usually, techniques of both approaches are needed when creating a comprehensive testing plan for a program.

Software testing methods are also commonly categorized into white-box testing and black-box testing methods. These two categories are utilized to illustrate the perspective that the programmers writing the tests take, and how much exposure they have to the source code when they are creating the test cases. In addition, a third category, which is a combination of the two, called grey-box testing is commonly used (Patton, 2005).

White-Box Testing

White-box testing is also called glass box testing or transparent box testing and refers to being able to see the source code. White-box tests inspect the inner workings of a program in contrast to testing only the functionality that is available to the end-user. When designing white-box test cases the testing engineer decides on the inputs to use for the tests and what the appropriate outputs should be. Some programming skills are therefore required to be able to understand the internals of the system and design meaningful tests for it.

White-box testing methods can be implemented at the unit in integration and system testing levels, but it is most commonly used in unit level testing. White-box tests can be used to test routes in a unit, between many units when integrating, or in subsystems at the time of system level testing. This type of testing method might reveal plenty of errors and issues, but it may not be able to identify missing specification parts or unfinished requirements (Patton, 2005).

Different white-box testing techniques include:

- API testing to test the application by utilizing its private and public application programming interfaces. The purpose of API testing is to check that the APIs respond correctly to a given request.
- Code coverage is a requirement level that the application must fulfill through testing. For example, a pre-defined percentage of the functions in the application must be tested to satisfy the level of code coverage.
- Fault injection includes causing errors on purpose in the application to determine the effectiveness of the testing strategies.
- Mutation testing is a method in which the source code is mutated to see how well the test cases can identify the problem.
- Static testing methods.

Black-Box Testing

Black-box testing considers the program as a non-transparent box in which the source code is not visible to the tester. In this approach, the functionality is being investigated without knowing anything about the inner functionalities or structures of the program. The only knowledge about the program that the testing engineer has is the expected outcome, but nothing about how the program achieves it.

An advantage of black-box testing is that the testing engineer must not have any prior knowledge of programming. Therefore the testers might have a different mindset as the programmers and write tests that underline different functionalities. The outcome is more robust test cases. On the other hand, because the testers have no visibility to the source code, they have an increased risk of writing unnecessarily many tests for a case that could have been tested more simply with a white-box testing method. Furthermore, the testers might miss writing tests for some parts of the program altogether. Black-box testing methods can be applied to all software testing levels but are most commonly used in high-level tests and in unit testing (Patton, 2005).

Examples of black-box testing methods are:

- Equivalence partitioning splits the input data into equivalent data partitions for the test cases.
- Boundary value analysis is used to design the tests so that they consist of ranges of boundary values.

- Error guessing method requires an experienced testing engineer, who uses their past knowledge of common software failure situations to create test cases that check how well the program handles situations that can cause errors.

- State transition testing technique observes how the state of the program changes with different kind of input types.

- Use case testing is used to find test cases that utilize the whole program from start to finish. Each use case outlines the user interactions with the program to complete a wanted outcome.

- Decision table testing combines the possible inputs and outputs in a table to demonstrate which test scenarios produce which results.

Grey-Box Testing

Grey-box testing is the combination of black-box and white-box testing methods. In grey-box testing, the tester might be aware of the internal structures and functionalities of the software, but the tests are written from the end user's perspective. This testing method can be used at any level of software testing, but most commonly it is utilized at the integration testing level (Patton, 2005).

Examples of grey-box testing methods include:

- Matrix testing is used to ensure the software meets the requirements set in the specification.

- Regression testing ensures that new changes to the software do not break any existing functionalities.

- Orthogonal array testing is a statistical way of testing the system with all possible inputs to find logical errors.

The following sections will explain in more detail different static and dynamic testing methods and the levels of testing as well as when applicable, whether they are white-box, black-box or grey-box testing methods.

3.2.1 Static Testing

Static testing is the analysis of a program that is achieved without the building or executing the actual program, as opposed to dynamic testing, which is performed after or during the execution of the program. Static analysis is usually implemented on the program's source code or parts of it. In the next two chapters two commonly used static testing methods are explained.

Code Linting

A typical method of static testing is code linting, which involves analyzing the source code and checking for syntax errors, bugs, code structure and other types of stylistic mistakes. Stephen C. Johnson created the first linter in 1978 at Bell Labs to debug yet another compiler-compiler (YACC) code he was creating for C language in order to run the Unix operating system on a 32-bit computer. The term "lint" is derived from the unwanted pieces of fiber, or lint, found in fabric (Johnson, 1978).

Optimizing compilers also implement similar code analysis as linters do to find ways to create faster code. Johnson mentioned the issue in his 1978 paper by saying *"the general notion of having two programs is a good one",* since they are focusing on different matters and therefore enabling the programmer to *"concentrate at one stage of the programming process solely on the algorithms, data structures, and correctness of the program, and then later retrofit, with the aid of lint, the desirable properties of universality and portability"* (1978).

Nowadays code compilers have a considerable amount of lint-like functionality, but similarly linting tools have also expanded to identify more wide-range issues in the code than before. For example, linters can raise flags about syntax errors, detect usage of deprecated functions, check for undeclared variables, and identify text formatting practices and many other types of complex features.

Linting can be very useful in particular with coding languages that are executed directly without compiling, such as JavaScript and Python. These so-called interpreted languages are missing the compiling state in which the errors would be identified before the actual execution. Linters can be used for debugging typical errors, such as code

syntax differences, or purposed for more demanding issues such as finding suspiciously constructed code.

Static Type Checking

Static type checking is another type of verification process, which analyses the source code of a program and checks for the safety of the types. A program that succeeds in static type checking is then assured to fulfill a set of type safety attributes for the program's inputs.

Java programming language is an example of a statically typed language and it requires that the variable types be defined in the code. Once the code is compiled, the compiler checks that only values of correct types are assigned. A more modern example of type checking from web development is TypeScript language, which is a superset of JavaScript and comes with built-in type checking. TypeScript compiles into normal JavaScript code so it can be run in web environments, such as web browsers or Node servers, but the types are checked at runtime without the need to compile the code first.

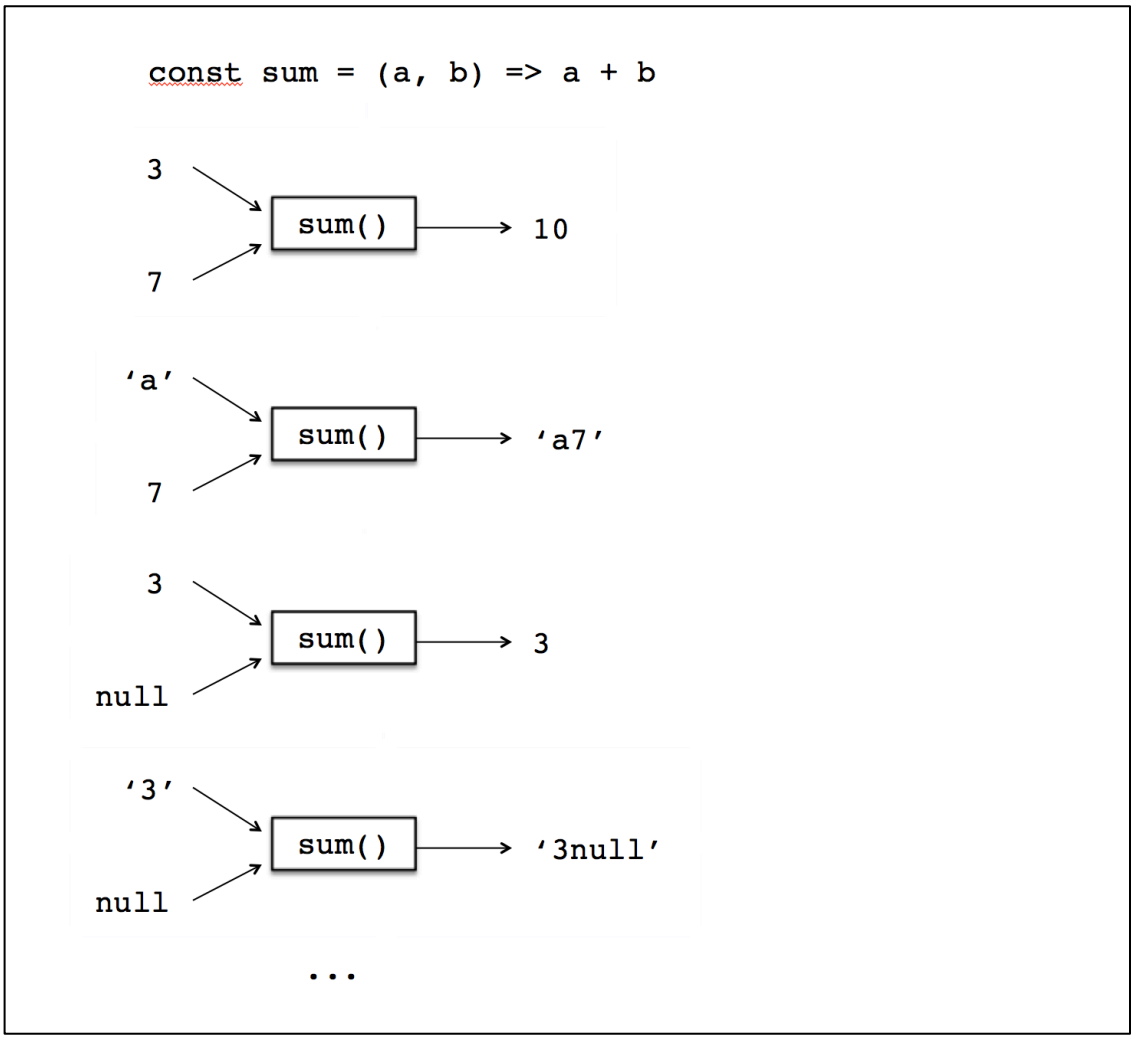


Figure 2. Without type checking a function yields unexpected results (Perez, 2009).

Figure 2 shows a fictional JavaScript function `sum()`, which accepts two inputs and outputs a sum of them. Without type checking, the function will accept any input formats and can yield unexpected results. The function is supposed to accept two numbers and return a sum of them, but JavaScript is a fairly forgiving programming language and runs without errors while adding up different types of inputs as well, for example a number and a string. With type checking the programmer could make sure that the function accepts only numbers for inputs as the function would prompt an error if the received inputs were not of the correct type.

3.2.2 Dynamic Testing

Dynamic testing is usually thought to have four different levels: unit testing, integration testing, system testing and acceptance testing. Software tests are generally organized

either by the test's specificity level or based on the stage of the development process they are added on. Regression testing is a common way to test against breaking changes in the software, but it is not considered to be a separate level of testing, as it is a technique that can be utilized at any level of testing.

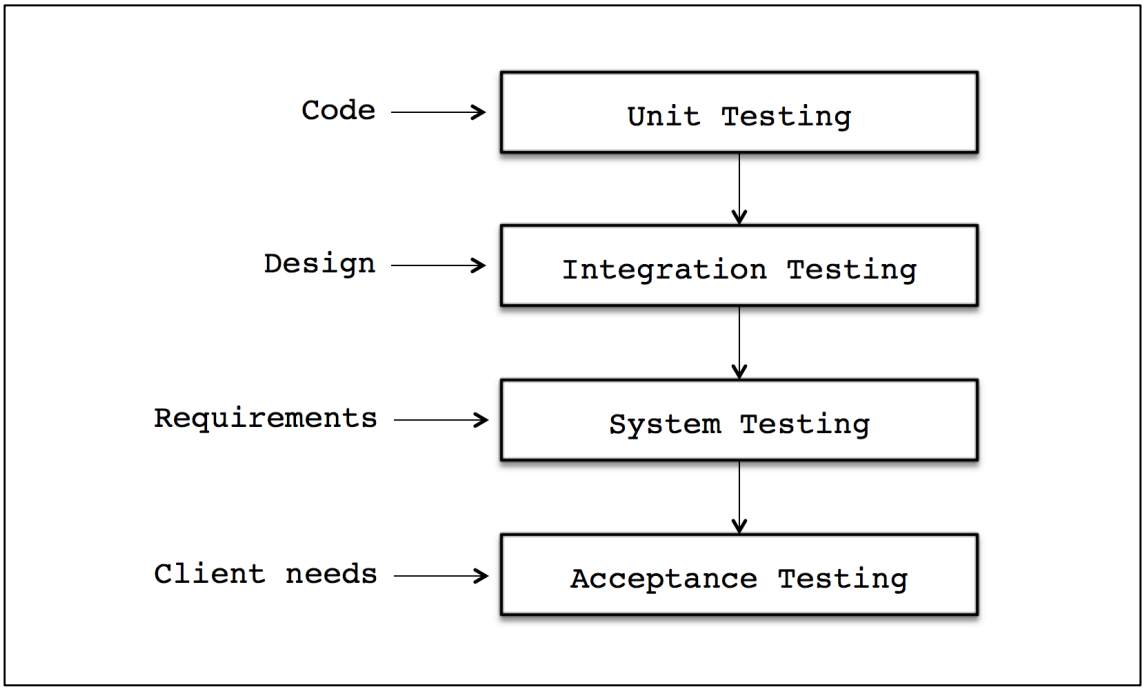


Figure 3. The four common dynamic software testing levels (Hackingig, 2017).

Figure 3 shows the four commonly recognized software testing levels and what parts of the software are being tested at each level. It also illustrates what are the determining factors that define the tests. The following chapters will explain in more detail these testing levels and their purposes.

Unit Testing

Unit testing is the first level of dynamic testing. During unit testing, the smallest parts of the software, called units, are tested. The purpose of this type of testing is to verify that each individual part of the software works as intended. A unit has no clear definition, but a function, method, class or a procedure can be considered a unit depending on the programming language. In object-oriented programming (OOP) a method is the smallest unit that can be tested. The method might belong to a super, abstract or a child class. When deciding what is a unit, the important part is to make sure that the

target cannot be broken down into smaller testable pieces. Otherwise, it will conflict with the philosophy of unit testing. Commonly used helpers in unit testing are frameworks, stubs, drivers and mock objects (Patton, 2005).

Unit tests are commonly written by programmers who are working on the codebase. Unit testing is a white-box testing method because the purpose is to verify that a specific unit is working as expected and therefore insights into the constructs of that unit are required. There can be multiple fests for a single unit to ensure that all the edge cases and branches are tested. Unit testing is an important part of software testing, but on its own, it cannot ensure the bigger functionality of the software. It is best utilized to make sure that the pieces of which the software is constructed are working independent of each other. The purpose is to make sure that any construction errors will not be passed forward to the later testing stages.

Integration Testing

Integration testing is the level of software testing that occurs after unit testing and before system testing. In integration testing, the singular parts of the software are merged together and tested as a larger group. Integration testing methods take as input the pieces of the software that have gone through unit testing, groups them together and perform the tests that have been defined in the integration phase. The output of this testing phase is an integrated system that is prepared for system testing (Patton, 2005).

The purpose of integration testing is to detect any errors in the interfaces and interactions between the individual modules and components. The testing continues to progressively group larger pieces of software together until the whole system is built and tested as a whole. Integration testing is usually automated using continuous integration practice, which integrates the software at regular intervals to find errors more quickly.

System Testing

System testing is conducted on a completely integrated system and its purpose is to verify that the system is compliant with the requirements specified for it. System testing does not require knowledge about the inner logic of the code and is therefore considered a black-box testing method. It tests how well the integrated components function

with each other and the system as a whole. It can also test the user experience and verify that different inputs produce the expected outputs. A test engineer performs system testing on the completed product before it is released to the public. The tester takes user's perspective for the testing and focuses on the external parts of the software when designing the test cases (Guru99, n.d.).

Acceptance Testing

Acceptance testing is the final level of software testing and it is used to test the system for acceptability. System testing phase tests the software against technical requirements, but acceptance testing tests that it is compliant with the business requirements and is ready to be delivered to the end-users. Acceptance testing is normally performed as black-box testing, but it does not necessarily follow any strict procedure (Patton, 2005).

3.2.3 Snapshot Testing

Snapshot testing can be a useful tool when the objective is to confirm that the user interface (UI) did not have unexpected side effects from code changes. For example, snapshot testing a mobile app could mean to render the UI component being tested, take a screenshot of the current state of the component, and then compare it to a reference screenshot that was captured earlier and stored in the code base. Image recognition software can detect any variations pixel-perfect from the two snapshots and point out the differences.

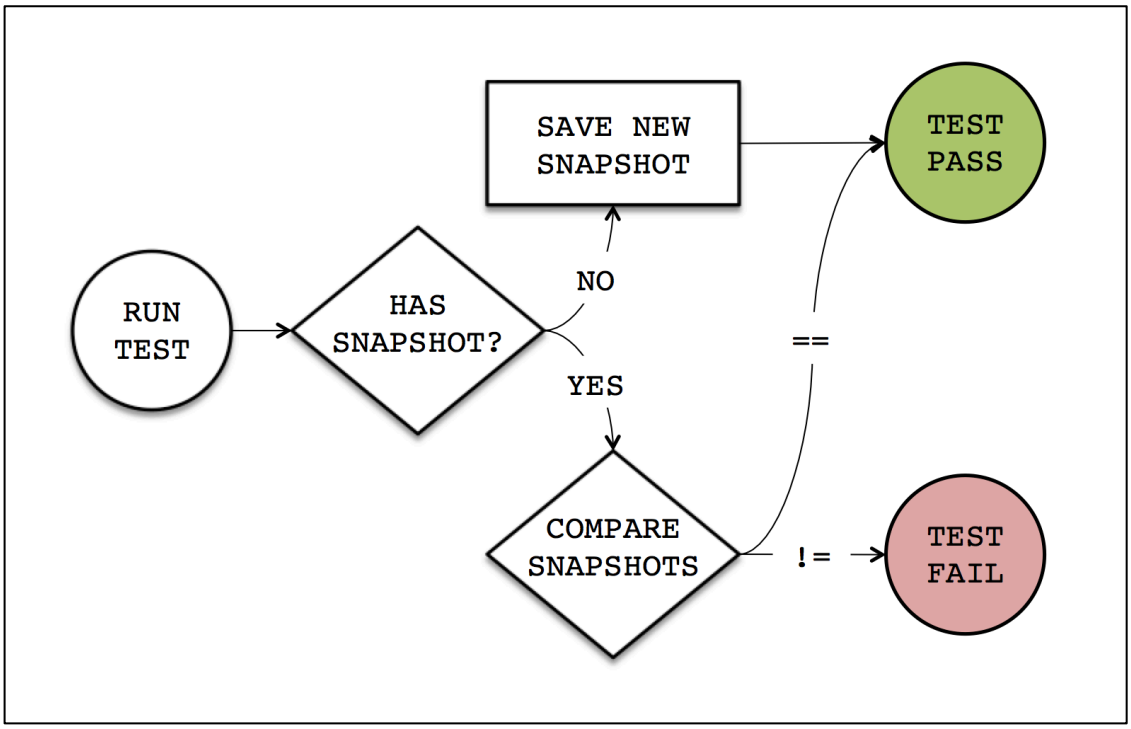


Figure 4. Flow chart of snapshot testing (Vieira, 2017).

The flowchart in Figure 4 shows the possible outcomes of a snapshot test. If the snapshots match or there is no initial snapshot, the test will pass. On the other hand, if there is an initial snapshot and it does not match with the new version the test will fail. The mismatch might be a result of a programmer mistake and requires further changes to make the snapshots match again. On the other hand, the component could have been deliberately modified and the mismatch was expected to happen. In the latter case, the correct action is to create and store a new snapshot and delete the previous, outdated version. A similar approach can be used with programming languages that support test rendering. In such cases instead of rendering the actual UI, which would need the whole app to be built, a serialized text version of the UI can be rendered and parsed in text format.

Snapshot tests should be written as deterministic, so when the same test is run several times for a component that has not been modified, the tests should produce the same outcome each time. The tests should not include any non-deterministic data that might be different depending on the environment where the tests are run. For example, tests for a component that displays the current time should not depend on the actual time of the testing environment, but a mock value, for example, a string `123`, should be pro-

vided to represent the time. This way will ensure that the generated snapshots will always match, no matter what the actual time during the testing is. To test that a component actually renders the correct current time, a unit test should be used instead. The snapshot files should be committed to codebase together with the components and their tests as they provide the comparison point for the snapshot tests (Facebook, n.d.).

3.2.4 Code Coverage

In software development code coverage refers to a measurement represented as percentage and it illustrates what amount of the source code is executed while tests suites are run. When the code coverage is high for a particular program, it means that a large amount of its source code was executed while the tests were run. High code coverage gives the impression that the program contains less undetected bugs than a program with lower code coverage. Test coverage can be calculated by various metrics, but one of the most basic methods is to measure the ratio of a program's subroutines and statements that get called while the test suite is executed, compared to the ones that are not called (Patton, 2005).

At least one code coverage criteria has to be used to determine the percentage of code that has been exercised while running the test suite. A code coverage criterion is normally specified as a condition that the tests have to meet in order to pass successfully. The main code coverage criteria include:

- Function coverage measures how many of the program's functions are called.
- Statement coverage is to measure the number of statements that are executed.
- Branch coverage measures the amount of branches being executed, such as if-else statement possibilities. Another way to think about branch coverage is to check that all possible variations in the code are being executed.
- Condition coverage, measures if every Boolean sub-expression has been assessed with the two possible values, `true` and `false`.

Code coverage is a useful tool to find parts in the program's source code that have not been tested. It does not, however, function as a direct measurement of how well the

code has been tested and how well written the tests are. Sometimes the desired level of code coverage has been determined beforehand and the programmers are expected to meet that target level. This can in times introduce negative effects if the targeted coverage level can be reached with poorly written tests. Tests are considered written poorly when they are testing parts of the program that are rarely failing, with the intention of boosting the coverage percentage instead of testing the parts that are really essential for the program to work correctly.

Planning beforehand is essential when writing meaningful and well-functioning tests. Test-driven development (TDD) is a useful method that can be utilized to achieve good tests. In TDD development cycle the tests are written before the actual code. Well-tested software might have code coverage in the range of 80-90%, but reaching 100% coverage should not be the target. A very low coverage percentage does imply that the software is not tested as thoroughly as it should be, but very high numbers are not always the implication of the opposite either. A satisfactory amount of testing consists of more complicated matters than the code coverage percentage alone can offer. In general, the code has sufficient amount of testing when the production code rarely gets bugs and when the programmers can confidently change parts of the code without the fear of causing bugs. However, if the tests are slowing down the development work significantly, it might be the cause of having too many tests. Making a simple change in the code should not result in unnecessarily burdensome changes to the tests. Therefore, the tests should be concentrated around the main functionalities and parts of the program and be written in a way that they allow some non-essential changes to happen in the code without the need of re-writing the tests (Fowler, 2012).

3.2.5 Regression Testing

Regression testing is used to make sure that the code developed and tested earlier will still work the same way after it is modified or extended. Common changes that occur in the code include enhancements, bug fixes, configuration changes and dependency version updates. Regression testing can uncover these new bugs, or regressions, from the changes. In addition, software change-impact analysis can be implemented to figure out functional or non-functional areas in the program that could be influenced by the changes.

Usually, regression-testing means re-running completed tests from earlier runs and inspecting whether the behavior of the program has mutated. Another common method is to check if previously fixed bugs or errors have appeared again. Regression testing can be run to cover the whole application every time new changes are introduced, or an appropriate selection of tests can be chosen to cover a particular change.

In software development, it is usually considered best practices to create a test case for a bug that has been discovered and fixed. The purpose is to have the test create a situation in which the bug emerged earlier and therefore make sure that any following changes do not expose that bug again. This kind of testing approach can be done manually, but the more effective way is to use automated testing tools that allow the testing environment to run the regression tests automatically. If the application is affected by external forces, such as third-party hosted services, it might be necessary to run the tests at regular intervals to catch any failures in the dependencies as quickly as possible. Regression test suites can be run after each time new changes introduced to the codebase and the application is re-built, or every night when there is the least activity within the application. Although most commonly they are run using a combination of both of the strategies (Patton, 2005).

4 Case Study Analysis

In this section, the two case studies are analyzed using the four-step case study analysis research method described in the chapter 2. Both of the chosen projects were large-scale software development projects involving dozens of participants and the work continued over the span of several years.

4.1 Case Study A

The project selected for the case study A was conducted for a large international hi-tech company in California and for future reference will be called Company A. The product under development was a fairly complex management system with different types of assets and entities that were to be configured using the application. The product was supposed to be run in a web browser, but it was not available to the public, as it requires local installation. The project was a continuous development project that started with a small contribution at first but evolved larger during the process. The company A hired Idean to provide design and development for the front-end part. The front-end development began with a small team of two developers, but as the work started to become more resource heavy, several other developers were brought in. Eventually, the front-end team consisted of three separate teams, one in the United States and two in Europe, and totaling of about 10-15 programmers.

The main technology used in the front-end was React JavaScript framework for the user interface and Redux for state handling. The Company A provided the back-end for the application and created APIs through which the front-end would interact with the back-end. As the product was a web application, HTML and Sassy CSS (SCSS) were also utilized in most parts.

The project had a fairly typical software development workflow. The front-end was separated into its own repository and utilized a branching model created by Vincent Driessen called GitFlow. The model makes parallel development easy as it separates the development work from the finished product. The main branch in which the development work is being done is called `develop`. When the programmer starts to develop a new feature or a bug fix they would branch off from the latest develop branch. Once the feature has been finished the programmer would create a pull request to have the changes reviewed and then merged back into the develop branch. Only the lead developer has the ability to merge feature branches into develop, but anyone involved is

allowed to review code and give their approval or comments. Once the develop branch is in a state that it can be released, a `release` branch is created from the latest state of develop. After that only bug fixes are generally merged into release branch, as it should be feature-ready by the time of creation. When there are no more bugs in the release branch it is tagged with a version number and merged into the `master` branch. Master branch contains publish-ready code and is never worked on directly (Herbert, 2013).

In addition to the development workflow, the server setup was also separated into different environments. There were four server environments: `develop`, `staging`, `testing` and `production`. The separation of these stages made sure that any changes would not go through to production without enough scrutiny and testing on the way.

In the project unit tests were created for most of the components and utilities. The Company A had set a code coverage target of 80%, but it was not strictly enforced or followed. The unit tests were not run as part of the continuous integration (CI) flow. Linters were used for both JavaScript and SCSS code, but they also were not integrated with the CI. Code reviews were conducted by all programmers and only approved pull requests were merged. Programmers were expected to test their changes as thoroughly as possible, using different browser environments and inputs. A team of one to two quality assurance (QA) testers went through the changes and reported bugs if any were found. They also tested the whole application from time to time to find any new bugs.

Main Challenges and Problems

The biggest challenge for the project came from the scale of the front-end team. As the team was spread out to different parts of the world there was rarely mutual time with all team members to discuss project related issues. Therefore, it would have been essential to have very strict enforcement of certain coding related issues, such as structure and design of the code. The linters were set up to enforce generic rules, but they were no so precise that they could have prevented bad code from being written. Also, because the CI server did not run the linters, enforcing them became the responsibility of each individual programmer. At times when new programmers were introduced to the project, they were not familiar with linters and did not configure them correctly with their

text editors. Therefore they never saw any linter errors in their editors and thought that nothing is wrong with their code.

Another issue was the lack of unit tests and changes that kept breaking them. Again, if the unit tests had been a part of the CI flow their enforcement would have become a requirement to get any changes merged in. The CI server did not however run them and therefore new changes could break the tests and be merged in without anyone noticing. Lack of guidance about writing the unit tests was also a problem, as some programmers skipped writing tests for the features they had created. Keeping up with the 80% code coverage target became an impossible task.

Programmers also got too relaxed with their testing, as they knew that the QA team would do the final testing on the changes. Therefore many changes were not tested with different browsers or had some other bugs that would have been easily noticed. If the QA team found bugs on the changes, they returned the ticket back to the person working on them. In addition, since the QA team was conducting regression testing manually, they could not keep up with all the bugs that appeared. Therefore some errors went unnoticed for long periods of time.

Alternative Solutions

The code linters could have had more strict rules so they would have caught more issues with the code. They could have also been integrated locally as part of the committing stage so that the commits would have failed if the linters did not pass. In addition, they could have been executed at the CI server as well, so that the pull request would be rejected or flagged if it did not pass all linters. Making sure that all the team members had their text editors configured to run linters in real-time could have been encouraged and documented better.

The creation of unit tests could have been stricter. Once a new feature was created it could have been checked to contain at least a minimum set of unit tests written for it. This could have been enforced by a commit hook or at the CI server. Running the test suites could also have been part of the CI process so that no changes with breaking tests would have been merged. Documented examples of well-written tests could have helped programmers who were not very familiar with testing to get started more easily.

Having automated regression tests could have enforced testing on many browsers. The regression test runs could have been executed on the CI server or manually. Passing the regression tests could have been a requirement to have the changes approved for merging. Alternatively, the QA team could have been larger so that the testers would have had a reasonable workload and could have identified more bugs.

Recommended Solutions

Having stricter linter rules would have clearly increased the quality of the code. In addition hooking the linter execution as part of the CI flow would have prevented any low-quality code from being accepted. Setting them up as a commit hook might not have been the best solution because sometimes it is easier to do many small commits while working on a big feature and leave the cleaning up in the end. Making sure that each programmer has their text editors configured to show linter errors in real-time would be optimal, but with the CI server running the linters that would not be strictly necessary.

Similarly, hooking up the unit test runs with the CI flow would have significantly increased the quality of the testing. No one could commit test-breaking changes if the CI server would run the tests for each pull request. Checking that every new feature has a minimum amount of tests written for them could have a negative impact as it could slow the development work. Still, mandating that features should have tests written for them at some point and having common test cases documented would encourage team members to keep the test coverage high.

Performing manual regression testing is not an optimal solution. It is very prone to human errors and takes an unreasonable amount of time to perform. A better solution would be to have the QA team write automated tests for the application and hook the regression tests as part of the CI flow. Then any new changes would be checked in case they break any previously written functionality. The tests could be configured to run in multiple browser environments to ensure maximum compatibility. The responsibility to check that a new feature without existing tests works as expected, or that a bug fix really fixes the bug, would primarily fall to the creator of the code, and secondly to the reviewer.

4.2 Case Study B

The project described in the Case Study B was produced for a large international networking company in Helsinki and is referred to as the Company B in the future. The product involved was a web-based system management tool involving configuring and managing different entities. The back-end environment had a very complicated system that was presented and managed through the user interface. Similarly to the product described in the Case Study A, it also was created using web technologies, required a local installation and was not available over the Internet to the public. The work had started as a prototyping project to test the feasibility of the product and slowly evolved into a production level development project. At the time when Idean joined the project to take over the front-end development and design, the product was already being prepared for its first release. The original front-end team continued to work on the project for a few months while the new team was becoming familiar with the product and codebase. The size of the core front-end team varied from five to eight people, mostly located in the same place, but at times split between two European countries.

The front-end user interface was built with React JavaScript framework. The number of external packages was kept fairly low because of a scrutinizing approval process that the Company B had in place for third-party dependencies. The Company B provided the back-end and the APIs for the front-end interaction. As with web applications commonly, the used technologies included HTML and cascading style sheets (CSS), but the latter one was eventually replaced with SCSS.

The project utilized a more uncommon Gerrit workflow for the development work. The whole application resided in the same repository but was separated into individual containers. In the Gerrit workflow branches are not used in general at all. All the development work was done on the master branch. Locally, the programmers pulled the latest code from master and branched off from it, or just worked on the `HEAD` of the master branch. All the changes to a feature or a bug fix were committed on one go referencing the master and that commit in Gerrit acted similarly as pull requests do in the GitFlow. In the Gerrit web interface, the reviewers inspected the commits and either left comments asking for changes or approved the commit and made it ready to be merged. If changes were needed, the original commit was amended and Gerrit would match it with the correct review by using unique `Change-ID`s. All participants conducted code reviews and anyone could merge the changes into the current master. However, the

master branch was not published automatically to production, but a separate workflow took place for the releases. During it, the release branch was branched off from the current master and only bug fixes were merged in it.

Continuous integration was also utilized in the process. Whenever a programmer pushed changes to the server, the CI would run tests against the changes. On the front-end there were snapshot tests for the main views and linters were used for JavaScript and SCSS code. Both the snapshot tests and linters were integrated with the CI flow, so they were run for all committed changes. In addition, automated regression tests were utilized for the user interface. The main use cases of the application had test cases designed for them. These tests were part of the CI flow, but they were not automatically run for every commit, because it took between one to two hours to complete the whole regression test cycle.

Main Challenges and Problems

When Idean joined the project, it soon became clear that the prototyping code from the beginning of the project was still used for the production. The structure of the repository and the code itself were lacking understandability and best practices. The linter rules used for the JavaScript code were not very well defined and did not enforce many good practices. The CSS code was not linted at all and the styling code was separated between CSS files and the inline code in the React components, which made it even harder to understand. In addition to the bad structure, the front-end code also did not have any comments or documentation, so some of the more complex components were very difficult to work with.

Despite having some snapshot tests, they did not really serve the purpose. The snapshot tests had been implemented without thinking how they would be useful and they were not kept updated when new views were created. Furthermore, the front-end components did not have any unit tests written for them, so as a result, some components were not tested at all.

The Gerrit workflow of having only one commit per feature meant that many commits would end up being very large. In GitFlow a pull request can consist of an infinite amount of small commits, and the creator of the pull request can show their workflow and thinking through the commit history. Despite Gerrit being developed by Google, it

never received such popularity as the two most common version control software GitHub and GitLab. Therefore Gerrit is also missing a lot of new functionalities that its rivals have, but also not having such a simple functionality as easy branching strategy is a clear weakness.

The regression tests were a great help verifying that new changes did not break any old functionality. However, they were also slowing down the development work significantly because of the way they had been created. The tests were very unforgiving to any view or component structure changes and could fail by the smallest changes in the code. The programmers had to spend a significant amount of their work time debugging the tests to find out why they failed and then fixing them. The tests were also not very consistent as there were many different implementations to achieve the same basic functionality.

Alternative Solutions

Once it became clear that the front-end code was at a prototype level instead of production, the development could have been started from scratch creating well-thought and constructed code instead of hot-fixing the existing one. The linter rules could have been improved and set to be stricter to prevent unwanted code from being created. Using CSS and inline styles in the component could have been avoided and instead use SCSS with corresponding linter. Code comments could have been added to allow for a better understanding of the code. Also, an automated documentation creation could have been possible by using JSDoc or similar tool, to further improve the maintainability.

The snapshot test strategy could have been defined in more detailed, or they could have been left out altogether. Unit tests could have been written for the front-end components to improve the test coverage.

Instead of Gerrit, it could have been possible to use GitHub or GitLab. Between the two they offer very similar functionality and structure, but the main difference is that GitHub software resides online on GitHub’s servers and GitLab is available either online or as a local installation. Both solutions offer private repositories, which are not accessible or visible to the public.

The regression tests could have been written in a way that non-functional changes would not have been allowed to break them. There could have also been a smaller amount of regression tests since it took a very long time to run them. Identifying and concentrating just on the core functionalities of the application could have been enough. In addition, utilizing custom keywords to create uniform ways to handle events in the UI could have made the test creation and modification easier.

Recommended Solutions

The poor quality of the code ended up causing so many delays and so much re-factoring that it would have been better to re-write majority of the components and functionality from scratch. SCSS should have been implemented sooner so that it would have enabled the creation of modular style sheets and utilization of linting. Code comments should have been implemented as soon as possible. Automated document creation probably would not have brought extra benefits since the back-end and user interface were already documented fairly well.

Unit tests should have been written in addition to the snapshot tests. The snapshot tests are great for testing the rendering of the components, but unit tests would complement them by testing the functionality. Running the linters and snapshot and unit tests with a commit hook would have saved time because the CI was already running so many tests that the feedback of failed tests would have taken too much time.

Removing Gerrit in favor of GitHub or GitLab probably would not have worked out. The development had already been continuing for a good time and the effort to set up the workflow in a completely new environment would have been too burdensome. In addition, Gerrit was the main Git system used by the Company B in all their projects, and convincing them to change away from it would have been very difficult.

The regression tests should have been modified heavily before they became too widespread. By utilizing custom keywords, fixing the regression tests would have been much faster and easier. Also better utilization of the APIs instead of the UI when creating data for the test cases would have reduced the running time of the test suites.

5 Results and Suggestions

This chapter reflects the results of the case study analysis from the chapter 4 and applies the theory from the chapter 3 into the framework of front-end software development by listing concrete tools and methods for improving code quality.

5.1 Documentation with JSDoc

JSDoc is a tool to create annotations in the JavaScript source code by utilizing code comments. The use of code comments allows the programmers to describe the API for the code they are writing within the source files. Having comments in the code enables other programmers to get a good overview of the functionality at glance. In addition, JSDoc can process the source files and create documentation in HTML or rich text formats (RTF) for all parts of the application that have been annotated using the JSDoc markup language (JSDoc, 2011).

```
/**
 * Sort array of objects by key
 * @method sortByKey
 * @param  {string}  sortDir
 * @param  {string}  sortKey
 * @return {array}
 */

export const sortByKey = (sortDir, sortKey) => (
  sortDir === 'desc'
    ? (a, b) => b[sortKey] > a[sortKey]
    : sortDir === 'asc'
    ? (a, b) => a[sortKey] > b[sortKey]
    : false
)
```

Listing 1. JavaScript function with JSDoc code comments.

Listing 1 shows a block of JavaScript code with a function `sortByKey`. JSDoc documentation is included in the comments above the function and includes a short description of what the function does, what input it accepts, and what is the expected output of the function.

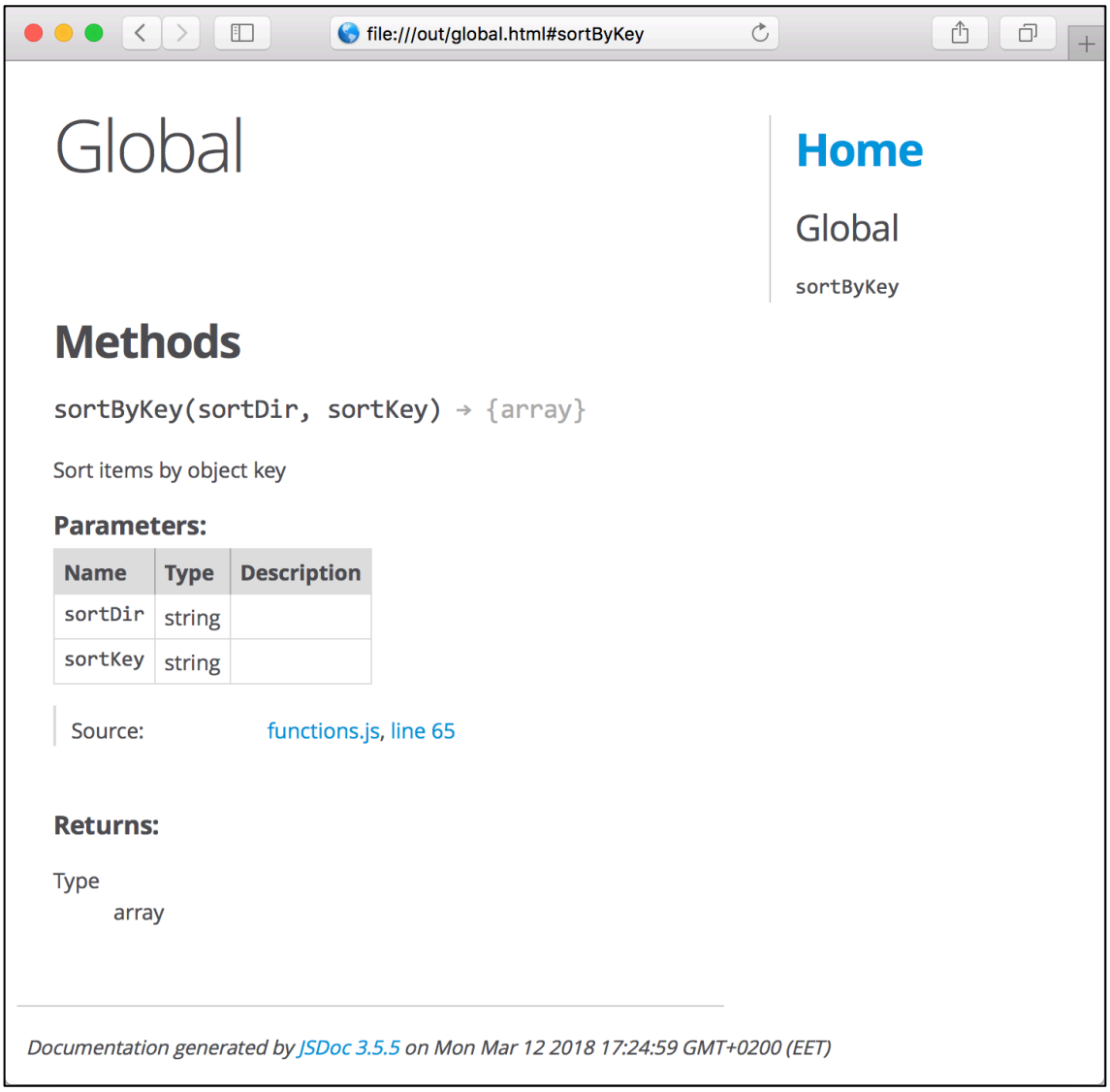


Figure 5. HTML output for JavaScript code block using JSDoc comments.

JSDoc comes with a command line tool that can process the entire application or individual files and create an output folder with HTML files for the documentation. This allows easier browsing of the documentation instead of reading through the source code. Figure 5 shows the HTML output for the function from Listing 1.

DocBlockr

DocBlockr is a plugin available for Atom and Sublime text editors and makes writing code comment documentation quicker and simpler. It can automatically parse the code and add the code comment block and function definitions in the format expected by

JSDoc. DocBlockr can be used with other popular programming languages also, such as PHP Hypertext Preprocessor (PHP), Java and C++.

Once DocBlockr is installed, it requires no configuration or key shortcuts but works in real-time while typing. Pressing the `Enter` key after typing the beginning tag for a comment block, `/**`, DocBlockr automatically creates a comment block. If the line after the comment holds a function definition, DocBlockr will try to parse the name and parameters of the function and add them to the documentation. In a similar manner, when written before a variable declaration, DocBlockr attempts to determine the data type of the variable. Using common naming conventions for variables helps, as DocBlockr assumes that variable names starting with `is` or `has` are booleans and `callback`, `cb`, `fn`, `next` and `done` are functions (Kalige, 2014).

5.2 Jest Testing Framework

Jest is a testing library developed by Facebook for testing JavaScript code, such as React applications. It was originally released in 2014 and quickly gained a lot of interest, but it took some time until the development community fully embraced it. Its core principle is to offer an easy testing platform that does not need configuration. Facebook reveals the reasoning for the approach as: *"We observed that when engineers are provided with ready-to-use tools, they end up writing more tests, which in turn results in more stable and healthy code bases"* (n.d.). Jest achieves efficiency by running tests in parallel and has a watch mode to check which files have been changed and can run tests only for them. It also includes many advanced features such as built-in JSDom to write browser tests that can be run in Node, asynchronous tests and support for stubs, spies and mocking (Facebook, n.d.).

Testing React components can be approached from many different angles. Examples of common testing strategies include (Ortega, 2017):

- Verifying that a certain function that has been passed as a prop is called when a particular event is triggered. This is a common type of unit test with React components.

- Checking the results of the render method with a given component state and verifying that it matches a predefined layout. This method in the Jest ecosystem is called snapshot testing.

- Counting the number of component's child elements and comparing that against the expected outcome.

Tests for components and modules are commonly included in the same folder as the test subject. By default, Jest expects that the test folder is named with two underscores before and after the word "tests", `__tests__`. In addition, the test file's naming convention should follow that of the components'. For example, a component named `Button.js` should have a test file named as `Button.test.js`. A common file structure for a web application has a folder called `components` and inside it are separate folders for each individual component. By using the same example, a Button component would have a path `components/Button/` and therefore its tests would be located in a folder `components/Button/__tests__/`.

Unit Testing with Jest

Utilizing Jest makes the most sense when coupled together with React, as they are both developed by Facebook. In general, too much testing of React components is not encouraged. By utilizing functional programming paradigm most of the business logic that should be tested thoroughly should not be included in the components themselves but written as separate functions. Still, there are times when it becomes useful to test React interactions, such as testing that the correct function with expected arguments is called when an element is clicked. To test the internal functionality of React, Facebook has released a library ReactTestUtils, which provides the basic functionality to test React applications. Facebook also recommends using another testing utility developed by AirBnB called Enzyme, which offers easy assertation, manipulation, and traversing of React component's output (Franklin, 2017).

When the React components are mounted and traversed utilizing Enzyme their properties, state and children props are easily accessible. There are two ways to mount the components with Enzyme, mounting and shallow mounting. The difference between the two is that mounting the component will load its entire DOM tree and shallow mounting only loads the root component in memory (Ortega, 2017).

```
import React from 'react'
import { configure, shallow } from 'enzyme'
import Adapter from 'enzyme-adapter-react-16'
import Button from '../Button'
configure({ adapter: new Adapter() })

describe('Button', () => {
  test('can be passed a className as a prop', () => {
    const button = shallow(
      <Button className="newClass" />
    )

    expect(button.find('button').hasClass('newClass')).toBe(true)
  })

  test('should call onClick function when clicked', () => {
    const mockFunc = jest.genMockFunction()
    const button = shallow(
      <Button onClick={mockFunc} />
    )
    button.find('button').simulate('click')

    expect(mockFunc.mock.calls.length).toBe(1)
  })
})
```

Listing 2. Jest unit test for a Button component.

An example of a React component's unit test suite is shown in Listing 2. On top of the file are imports for the packages that are required to run the test. The test suite itself consists of two test cases, so a `describe()` function is used to group the tests in one test suite. Using `describe()` is not required, but it helps to keep related tests better organized. The actual tests are triggered by a `test()` function. It has been given two arguments in the example. The first argument is the name of the test, usually written as an expectation of what the component should do to pass the test. The second argument is a function that defines the expectations for the test. A third and optional argument, `timeout`, also exists, and it is a number for a timeout in milliseconds that specifies how long the test should wait until aborting.

The first test shown in Listing 2 is testing that the component can be passed a prop `className` with a value `newClass`. The expectation set in the test is that once the prop is passed to the component it will render a HTML button element with the correct class name. The second test is testing the functionality of the component and checks that when the component is passed a function as prop `onClick`, it will be called exactly one time when the button element is clicked. The function `simulate()` is one of Enzyme's helper functions that can be used to simulate different types of events with HTML elements.

```
 PASS  src/components/Button/__tests__/Button.test.js
  Button
    ✓ can be passed a className as a prop (8ms)
    ✓ should call onClick function when clicked (2ms)

Test Suites: 1 passed, 1 total
Tests:       2 passed, 2 total
Snapshots:   0 total
Time:        1.055s, estimated 2s
```

Figure 6. Console output for a Button component's Jest unit tests.

The unit tests from Listing 2 produce an output shown in Figure 6 when no errors are detected. The figure clearly shows the name of the test suite that was run and the tests included in the suite. For performance optimization reasons the times that it took to run the tests are also shown.

Snapshot Testing with Jest

Testing of React components can take a lot of work, even for some simple sounding tasks such as verifying that certain text has been rendered. Instead of testing the React components outputs individually, Jest offers snapshot tests. Snapshot tests with Jest are not especially helpful when testing interactions, but they are a very effective way to verify that the components or views are rendering the correct output. When running a snapshot test, Jest renders the React component or view and saves the output in a separate file in JavaScript object-notation (JSON) format. Then every time the tests are run, Jest creates a new JSON output and compares that to the snapshot that was stored earlier. If a component's behavior has been modified and the snapshots do not match anymore, the programmer needs to take appropriate actions to fix the issue.

Snapshot testing a React application gives the programmer the ability to test the behavior and rendering of components without writing a lot of assertions. They also act as a fail safe to make sure that component's behavior has not changed accidentally. This does not, however, mean that every component should have its own snapshot test. The best way to utilize snapshots tests is to choose components that have highly essential functionality and are most critical to keep working as expected. Taking snapshots of too many components might slow down the tests and create a lot of extra work with constant updates of the snapshots. Another important point is that React, being developed by a large technology company such as Facebook, is very thoroughly tested by its developers and therefore the users of Jest should not end up testing the framework instead of their own code.

```
import React from 'react'
import renderer from 'react-test-renderer'
import Button from '../Button'

describe('Button', () => {
  test('renders correctly', () => {
    const tree = renderer.create(
      <Button />,
    ).toJSON()

    expect(tree).toMatchSnapshot()
  })
})
```

Listing 3. Jest snapshot test for a Button component.

Writing snapshot tests is usually very trivial. An example snapshot test can be seen in Listing 3. A test function is called instructing it to render a Button component with a text `Save`. Then the expectation for the test to pass is set to have the new snapshot to match the one stored from a previous run.

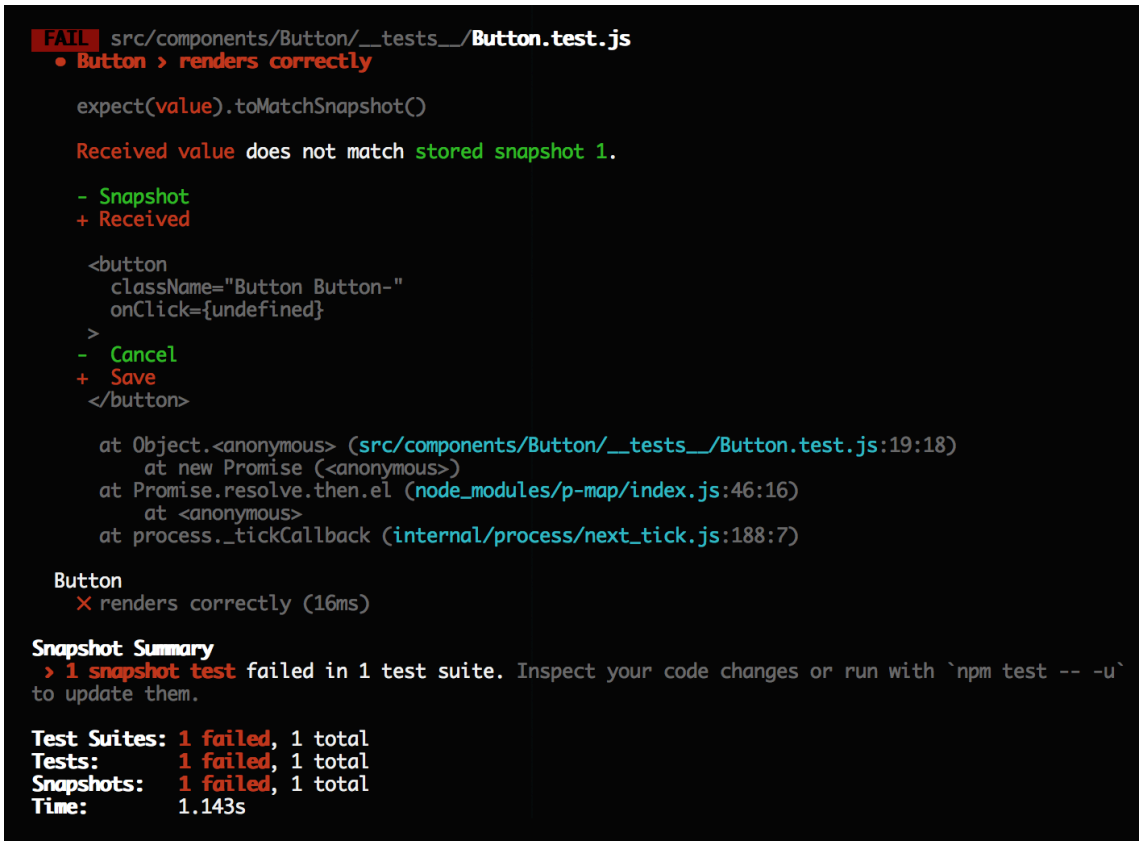


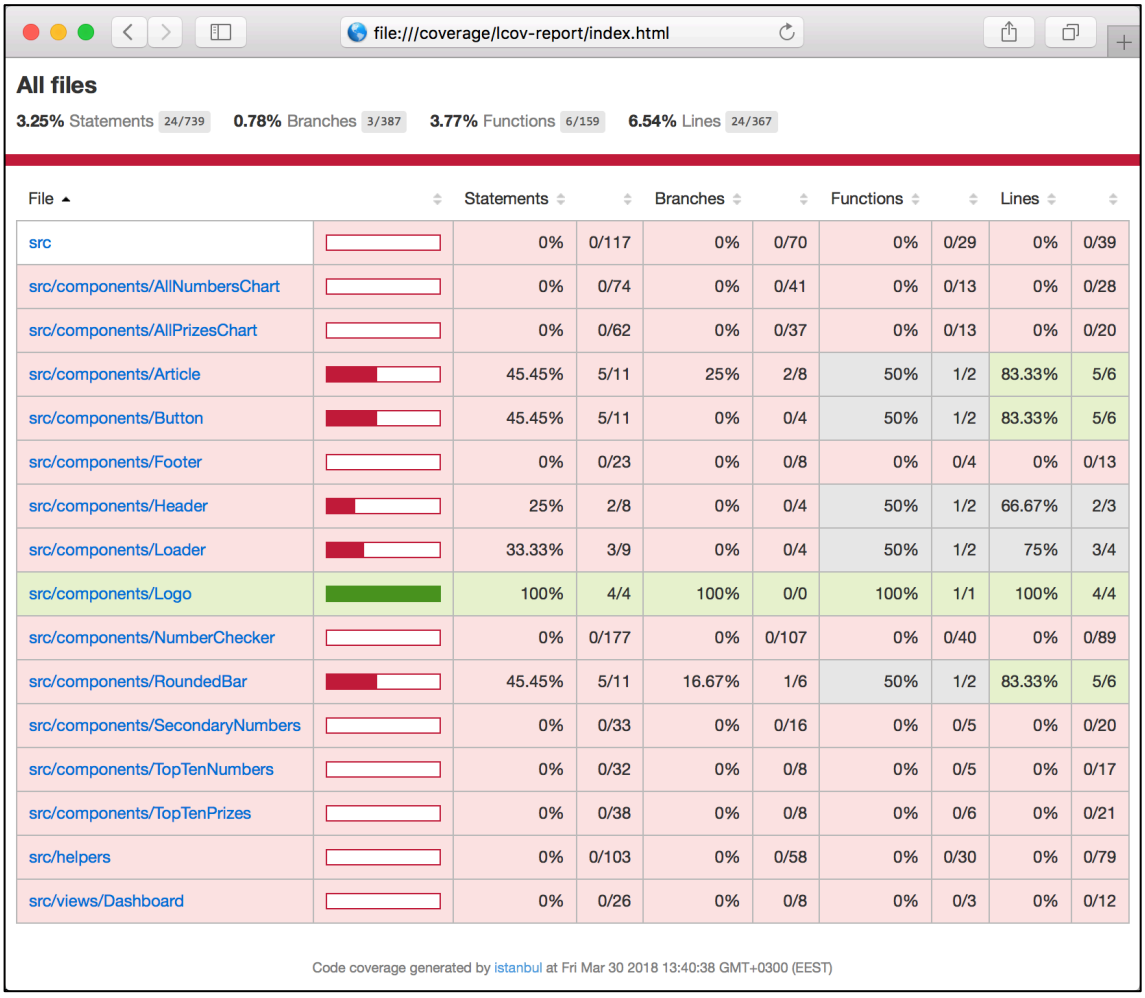
Figure 7. Console output of a failed Jest snapshot test.

Figure 7 shows a console output of a failed Jest snapshot test. The explanation of the failure is mentioned on the top as `Received value does not match stored snapshot 1.` and tells the programmer that something has changed within the component that made it differ from the previous snapshot. The highlighted lines in the middle of the figure indicate that the stored snapshot had a button with text `Save` and the new snapshot had a button with text `Cancel`. On the bottom of the figure, the snapshot summary instructs the programmer to inspect if the code changes were expected to break the test and to update the snapshot if that is the case.

Code Coverage with Jest

Jest has a code coverage tool called Istanbul built into it. Istanbul was developed by Krishnan Anantheswaran from Yahoo and is a popular client-side JavaScript code coverage tool. The integration with Jest has been made easy and requires no configuration from the user. To collect coverage a keyword `--coverage` should be used when executing the tests. By default, Jest will output the coverage report in HTML format as well

as in the console, but the coverage can also be integrated to show a real-time coverage with text editor plugins.



File	Statements		Branches		Functions		Lines	
src	0%	0/117	0%	0/70	0%	0/29	0%	0/39
src/components/AllNumbersChart	0%	0/74	0%	0/41	0%	0/13	0%	0/28
src/components/AllPrizesChart	0%	0/62	0%	0/37	0%	0/13	0%	0/20
src/components/Article	45.45%	5/11	25%	2/8	50%	1/2	83.33%	5/6
src/components/Button	45.45%	5/11	0%	0/4	50%	1/2	83.33%	5/6
src/components/Footer	0%	0/23	0%	0/8	0%	0/4	0%	0/13
src/components/Header	25%	2/8	0%	0/4	50%	1/2	66.67%	2/3
src/components/Loader	33.33%	3/9	0%	0/4	50%	1/2	75%	3/4
src/components/Logo	100%	4/4	100%	0/0	100%	1/1	100%	4/4
src/components/NumberChecker	0%	0/177	0%	0/107	0%	0/40	0%	0/89
src/components/RoundedBar	45.45%	5/11	16.67%	1/6	50%	1/2	83.33%	5/6
src/components/SecondaryNumbers	0%	0/33	0%	0/16	0%	0/5	0%	0/20
src/components/TopTenNumbers	0%	0/32	0%	0/8	0%	0/5	0%	0/17
src/components/TopTenPrizes	0%	0/38	0%	0/8	0%	0/6	0%	0/21
src/helpers	0%	0/103	0%	0/58	0%	0/30	0%	0/79
src/views/Dashboard	0%	0/26	0%	0/8	0%	0/3	0%	0/12

Figure 8. HTML output of Jest code coverage.

Figure 8 shows the HTML output of code coverage for an application. The coloring of the table rows indicates how well certain parts of the program have been tested. The green color is indicating a good coverage and red indicates lacking coverage. The coverage has been divided further into statements, branches, functions and lines, each indicating slightly different parts of the program. In the provided example the code coverage shows that majority of the components are lacking tests altogether and the ones that are tested are only partially written as well. The only component with an adequate amount of tests to reach 100% coverage is the Logo.

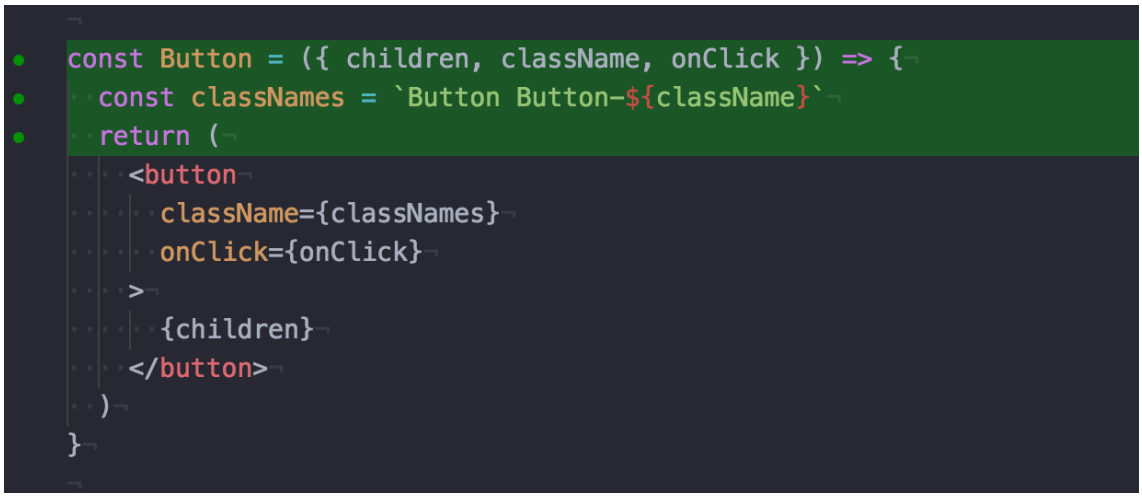


Figure 9. Atom text editor plugin highlighting covered lines.

Using a plugin to show coverage in the text editor is an easy way to spot tested, and untested parts of the code. Figure 9 shows a screenshot of Atom text editor using a plugin lcov-info. The highlighted lines indicate that the business logic of the Button component is covered by at least one type of test.

5.3 E2E Testing with Robot Framework

Robot Framework is a testing framework for automating acceptance tests in acceptance test-driven development (ATDD). It was developed by Nokia Networks in 2005 and released as an open source software in 2008. The code syntax applies keyword-drive approach in the tests and supports many different formats for writing the test cases. The supported formats for the tests are HTML, separated with tabs, or in plain text format utilizing space separation or pipe separation for the fields. The framework can be further extended with testing libraries that are implemented with Python or Java and provides a broad ecosystem that consists of several ready-to-use testing libraries and tools that are the results of different projects. Because the framework contains a modular architecture, users can also create their own keywords utilizing the existing ones and therefore create their own utility libraries for reusability.

Robot Framework does not depend on any specific operating system. It was implemented with Python but can be also used with Jython, which is an implementation of Python aimed to run on Java, or IronPython, which is an implementation of Python for .NET and Mono platforms.

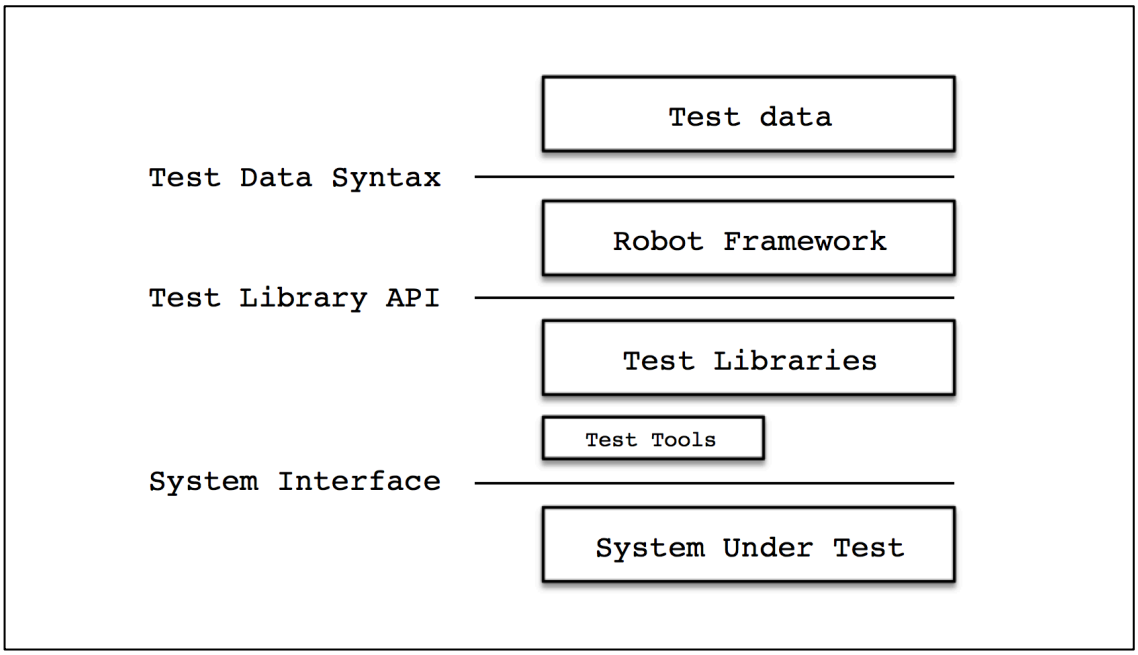


Figure 10. The modular structure of Robot Framework (Robot Framework, n.d.).

Figure 10 shows the modular structure of Robot Framework and demonstrates how it has been structured into separate parts. When Robot Framework is initiated, it receives the test data and begins executing the test cases. It then outputs the results of the tests as reports and logs. The framework itself does not have any knowledge about the system being tested, as the communication goes through the test libraries. The libraries can plainly utilize application interfaces or further extend the usage with test tools that are used as drivers (Robot Framework, n.d.).

SeleniumLibrary

SeleniumLibrary, previously known as Selenium2Library, is a Robot Framework testing library designed for web applications. It utilizes the Selenium WebDriver modules for automatically controlling the web browser. In addition to WebDriver, Selenium consists of many other browser automation tools that can be used as a browser plugin to record and repeat interactions in the browser for exploratory testing, or as part of a wider library that utilizes specific parts of it, as in the case of WebDriver in SeleniumLibrary. With SeleniumLibrary it is easy to create end-to-end (E2E) tests that are run in a real browser environment for web applications.

```
*** Variables ***
${BROWSER}  chrome

*** Settings ***
Library     SeleniumLibrary

Suite Setup     Open browser  http://localhost:3000  ${BROWSER}
Suite Teardown  Close all browsers

*** Test Cases ***
Sort Items
  Wait Until Page Contains Element   css=.Button-sort
  ${xAxis}=                          Get Text  css=.Chart-x-ticks
  Click Element                      css=.Button-sort
  ${xAxisSorted}=                    Get Text  css=.Chart-x-ticks
  Should Not Be Equal                ${xAxis}  ${xAxisSorted}
```

Listing 4. Example Robot test for a web application.

Listing 4 shows an example of a Robot test suite written for a web application. On top of the code, a variable `${BROWSER}` is created to allow testing with different browsers. The default browser environment is set to Chrome. Secondly, SeleniumLibrary is defined to be used as the testing library. Special keywords `Suite Setup` and `Suite Teardown` are used to instruct the framework with what should happen before the test cases are run and after they have been executed. In this case, as the test target is a web application, the setup step is to open a browser and navigate to uniform resource locator (URL) `http://localhost:3000`, which is the default URL for a local web environment. The teardown step, on the other hand, will close the browser, as it is not needed anymore after the tests have finished.

The actual test case shown in Listing 4 is testing sorting of items in a chart. The test starts by instructing Robot Framework to wait until an HTML element with a class name `.Button-sort` is located on the page. The wait is necessary because the browser takes some time to load the application and render it. Secondly, the test saves the current items of the chart's x-axis in a variable called `${xAxis}`. The next line tells the browser to click on the sorting button and right after that, it saves the changed state of the x-axis in another variable called `${xAxisSorted}`. The final step in the test is the

verification part in which the previously saved variables are compared and the expectation is that they should not be equal. This test assumes that initially the chart items are not sorted and by clicking the sorting button it can establish that the sorting did indeed change the order of the items.

Figure 11. Console output of a finished Robot Framework test suite.

Figure 11 shows the console output of the test suite after it has been run. From the figure can be seen that `Sort items` test was run and it passed. On the bottom of the figure, Robot Framework shows the output files that it automatically creates when tests are executed.

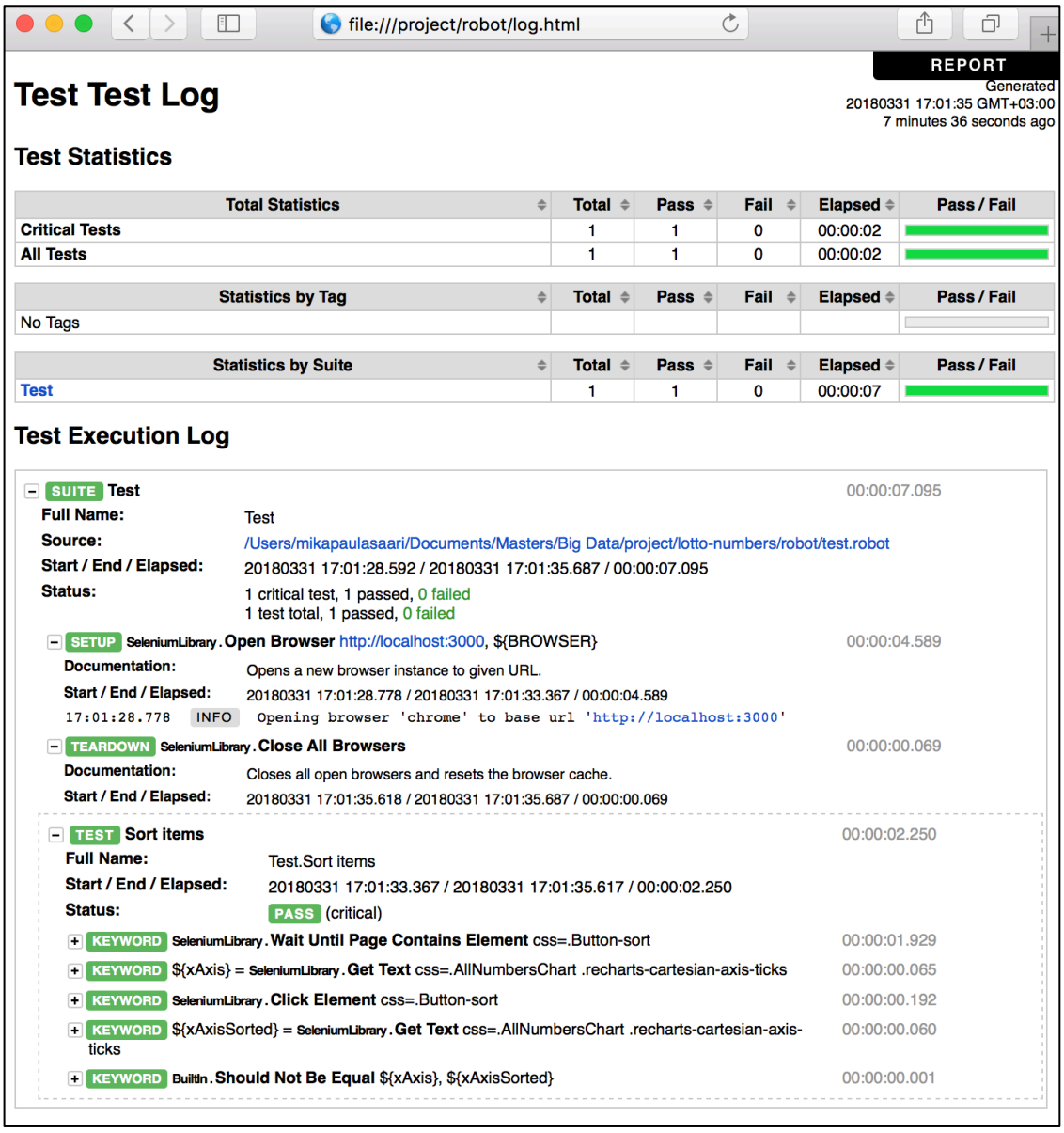


Figure 12. HTML output of a Robot Framework test suite.

Figure 12 shows the HTML output log file created automatically by Robot Framework for the executed test suite. The log file contains more information and is in a more easily understandable format than the console output and is, therefore, a better choice for debugging the failed test cases. Using separate test libraries makes it possible to configure Robot to take screenshots and video captures of failed tests, or provide even more robust log information about the state of the application at the moment of the failure.

Writing thorough E2E tests with Robot Framework and automating the test runs with the CI workflow enables robust regression testing for a web application. The CI server can be configured so that the tests become a part of the deployment pipeline and are then run every time a new code is pushed in the repository. This will ensure that no breaking changes make it to merging as long as the tests are passing.

5.4 ESLint for JavaScript

Nicholas C. Zakas created ESLint in 2013 and published is as an open source project. ESLint can be used through a Command-line interface (CLI) or as a plugin for popular text editors, such as Atom and Sublime Text. ESLint has been built around two open source projects, Espree and Estraverse. Espree produces an Abstract Syntax Tree (AST) from the source code. Estraverse includes traversal functions that can parse the output AST. While the traversing is happening, ESLint keeps track of each node that is being visited.

The idea behind ESLint was to enable programmers to build their own rule sets for linting. ESLint is therefore very flexible and modeled to have all the rules customized. There are some built-in default rules, but they can be overwritten or extended as wished. The rules can also be enabled or disabled in the source code, so that a rule might not apply to a particular section or another rule might only be enabled for a specific line of code. This allows some exception rules to be included if needed.

```
{
  "parserOptions": {
    "ecmaVersion": 6,
  "env": {
    "es6": true,
    "browser": true,
  },
  “rules": {
    "no-alert": 0,
    "no-multi-spaces": 0,
    "no-nested-ternary": 0,
    "max-len": 80
  }
}
```

Listing 5. Example ESLint configuration.

Listing 5 shows an example of ESLint rule configuration. At the top are the configurations for the parser and the environment in which the ESLint is executed. In the example, the EcmaScript (ES) version is defined to be 6 for the parser and therefore all the code will be checked against rules set in the ECMA-262 standard for that version. The environment again confirms that `es6` syntax should be enabled and that the application is supposed to be run in a browser environment. Lastly, the rules section lists the actual rules to use for linting and states that there should be no alert functions in the code, no multiple spaces, ternary functions should not be nested and that the maximum line length is limited to 80 characters.

Figure 13. ESLint inline error in Atom text editor.

A typical ESLint error can be seen in Figure 13. When using an ESLint plugin with Atom text editor the plugin shows any linting errors in real-time. In the case shown in the figure, the line with the error contains two spaces, which is in violation of the rule `no-multi-spaces` shown in Listing 5. When used as a text editor plugin, ESLint also offers an option to automatically correct the problems. Some of the rules are fairly explicit and simple and such automatic fixes can also be very straightforward, for example removing the second space from the line. On the other hand, the rule `no-alert` states that there should be no `alert()` functions used in the code and fixing the problem by simply removing all alerts would probably not yield wanted results. Therefore the automatic fix works best for small issues like typos and any more complex problems are best solved manually.

5.5 SASS Lint for Style Sheets

SASS Lint is a linter run in Node environment and it works for both Syntactically Awesome Style Sheets (SASS) and SCSS code. The difference between the two standards is that SCSS is an extension of CSS syntax and therefore all valid CSS code is also valid SCSS, as opposed to SASS, that has its own syntax and has to be compiled into CSS. SASS is the older one of them, but SCSS has since become much more popular. A tool called SCSS Lint also existed before the creation of SASS Lint, which might ex-

plain why SASS Lint is using the name it does despite supporting both formats. Because of the similarity of the standards' names, they have become nearly synonymous amongst programmers who usually refer to SCSS despite occasionally using the term SASS.

SASS Lint offers two ways for configuration, a `sass-lint.yml` file or a `.sasslintrc` file. A .sasslintrc file can be written in JSON or Yaml ain't markup language (YAML) formats and converting from one format to another can be achieved easily with a tool such as json2yaml. SASS Lint can be directed to use a global configuration file by defining the path using the `sasslintConfig` option in the project's `package.json` configuration file.

```
...
files:
  include:
    - 'src/**/*.s+(a|c)ss'
  ignore:
    - 'node_modules/**'

rules:
  indentation:
    - 2
    - size: 2
  variable-name-format:
    - 2
    - convention: camelcase
  quotes:
    - 2
    - style: single
```

Listing 6. Part of an example sass-lint.yml configuration.

Listing 6 shows a part of an example SASS Lint configuration. On top of the configuration the formats and paths are defined for files that should be included in the linting process and others that should be ignored. Secondly, in the rules section, different rules are configured for the linter dictating what the convention should be for code indentation, variable naming and quote styling.

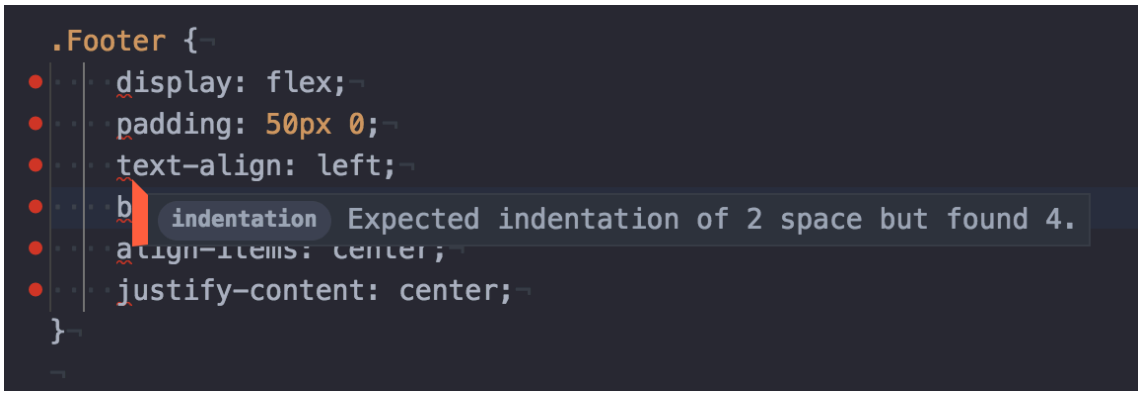


Figure 14. SASS Lint error in Atom text editor.

Figure 14 demonstrates SASS Lint errors in the Atom text editor. The rules shown in Listing 6 define that the code should be indented by two spaces, but the code in the figure has an indentation of 4 spaces. Atom highlights each erroneous row and shows an error message on when hovering the erroneous line.

5.6 Static Type Checking with Flow

Flow is an open-source static type checker for JavaScript. It is developed and maintained by Facebook and originally published in 2014. The method of type checking in Flow is opt-in and means that the programmer doesn't have to parse all of their code at once. Flow checks the types automatically whenever it is possible. Flow supports all the standard primitive JavaScript types such as `number`, `string`, `array` and `object`. An exception is that it does not consider `null` and `undefined` as own types. The main difference with Flow compared to TypeScript is that it can be included in existing projects, as with TypeScript, the codebase has to be re-written. In addition, as Flow is an opt-in method, it does not force the use of type annotations.

As with the linters, Flow can also be used as CLI tool, or as a plugin for the popular text editors. Installation works the same way as with any other editor plugin and once installed, Flow can start parsing the project immediately without further configuration.

Figure 15. Flow errors in Atom text editor.

Figure 15 shows an error triggered by Flow in which the user has not defined the types of the attributes of the function `sortByKey`. Flow prompts the user to define the annotations, or types, for the attributes. The error looks very similar to the one emitted by ESLint in

Figure 13, except that Flow does not offer automatic fixes to the problems that ESLint does.

6 Summary and Conclusions

This master thesis project included a research and a solution building to identify tools to improve code quality in front-end development projects. The main objective of the study was to suggest recommendations for how to design, implement and administer tools for code quality in large-scale front-end software development projects. The research phase of the study defined the attributes of good code quality and what tools and processes are currently available to enforce it. The analysis phase was conducted using the case study analysis research method to investigate two case studies of previous projects. Combining the knowledge from the research phase with the defined needs from the analysis phase created the final results.

The creation of the study was a fairly straightforward process. Writing the theory part and extensively studying the attributes of good code quality first helped greatly when the case study analyses were conducted. Knowing the ways that code quality could be maintained made finding alternative solutions for the case study issues easy. After the analysis, moving on to the Results and Suggestions chapter was a natural step. The creation of the results chapter was also the most enjoyable part of the thesis process as it included more concrete actions, for example conducting a significant amount of testing with the different tools.

When starting a front-end development project, planning ahead for the future needs is essential. No matter what size the project is, having a good toolset to help enforce best practices and reduce bugs is always beneficial. There are many ways to enforce the quality throughout the process and choosing which tools and methods to use depend on the business case and the size of the project.

Having a good documentation is professional and benefits the whole team now and in the future. With automated tools, the documentation creation becomes much easier and faster. Creating snapshot and unit tests will require some time and effort, but they are great for making sure that the user interface works as expected and that no unwanted side effects are produced. Code coverage is a concrete way to measure how well the application is tested and requires no extra effort once the tests have been set up. Regression tests are one of the most time consuming to set up and execute, but if they are utilized correctly they can be one of the most helpful ways to prevent unwanted bugs. Using linters and static type checkers is very trivial but can save tremendous

amounts of time by notifying about typos, inconsistencies and errors. The most time-consuming part is setting them up and deciding on the set of rules, but once they are constructed, they can be re-used in future projects.

More testing and fine-tuning may be carried out in the future to find the best ways to utilize the tools for specific projects. As mentioned earlier, once the set up has been done well, it can be used later on other projects as well. In addition to the tools described in the thesis, other types of recommendations might be conducted as well as things that cannot be automated. Examples of such recommendations would be naming conventions, data structures and text editor environments. Creation of a project template with the entire set up ready was considered in the beginning, but as technologies evolve fast, the maintenance of such a template could become too burdensome. Therefore keeping this thesis at more theoretical level was seen as a better option.

References

Barker, T. (2003). *Writing Software Documentation: A Task Oriented Approach*. 2nd ed. New Jersey: Pearson Education, pp. 21-22.

Facebook. (n.d.). *Jest. Delightful JavaScript Testing*. [online] Available at: https://facebook.github.io/jest [Accessed 2 Apr. 2018].

Fowler, M. (2012). *TestCoverage*. [online] ThoughtWorks. Available at: https://martinfowler.com/bliki/TestCoverage.html [Accessed 31 Mar. 2018].

Franklin, J. (2017). *How to Test React Components Using Jest*. [online] SitePoint. Available at: https://www.sitepoint.com/test-react-components-jest [Accessed 1 Apr 2018].

Guru99. (n.d.). *What is System Testing? Types & Definition with Example*. [online] Available at: https://www.guru99.com/system-testing.html [Accessed 18 Mar. 2018].

Hackingig. (2017). *What are software testing levels?*. [online] Available at: http://hackingig.com/what-are-software-testing-levels [Accessed 26 Mar. 2018].

Herbert, S. (2013). *Introducing GitFlow*. [online] DataSift. Available at: https://datasift.github.io/gitflow/IntroducingGitFlow.html [Accessed 1 Apr. 2018].

Johnson, S. (1978). *Lint, a C Program Checker*. New Jersey: Bell Laboratories, p. 9.

JSDoc. (2011). *@use JSDoc*. [online] Available at: http://usejsdoc.org [Accessed 18 Mar. 2018].

Kalige, N. (2014). *Dockblockr Package*. [online] Atom. Available at: https://atom.io/packages/docblockr [Accessed 18 Mar. 2018].

Laudon, K. and Laudon, J. (2005). *Essentials of Management Information Systems*. 6th ed. New Jersey: Prentice Hall, pp. 11-12.

Microsoft Patterns & Practices Team, (2009). *Microsoft Application Architecture Guide*. 2nd ed. Microsoft Press, pp. 191-203.

Ortega, A. (2017). *Test-driven React.js Development: React.js Unit Testing with Enzyme and Jest*. [online] Toptal. Available at: https://www.toptal.com/react/tdd-react-unit-testing-enzyme-jest [Accessed 2 Apr. 2018].

Patton, R. (2005). *Software Testing*. Indianapolis: Sams Publishing, pp. 54-62, 108-124, 202-232, 259.

Perez, P-L. (2009). *Type Checking Javascript*. [online] SlideShare. Available at: https://www.slideshare.net/pascallouis/type-checking-javascript [Accessed 11 Mar. 2018].

Robot Framework. (n.d.). *Robot Framework*. [online] Available at: http://robotframework.org [Accessed 2 Apr. 2018].

Vieira, L. (2017). *Snapshot testing React components with Jest*. [online] Medium. Available at: https://medium.com/@luisvieira_gmr/snapshot-testing-react-components-with-jest-3455d73932a4 [Accessed 18 Mar. 2018].