Analysis of the behavioral impact of code modifications

Jorge Hermano Carrilho da Silva Veiga

Thesis to obtain the Master of Science Degree in
Information Systems and Computer Engineering

Supervisor: Prof. Rui Filipe Lima Maranhão de Abreu

Examination Committee

Chairperson: Prof. Miguel Nuno Dias Alves Pupo Correia
Supervisor: Prof. Rui Filipe Lima Maranhão de Abreu
Member of the Committee: Prof. António Manuel Ferreira Rito da Silva

October 2020
Acknowledgments

I would like to thank my supervisors for all the support and availability shown to me during the process of developing this thesis, as well as the belief in me to develop it.

Moreover I would like to thank all the professors I crossed paths with during my time at IST, who all have contributed to this journey. I extend this thank you to all the colleagues I worked with over the years, we learned a lot together and this journey would not have been possible without them.

I would also like to thank all the participants in our evaluation process, which was done in a less than ideal way through the Summer and remotely. I’m thankful not only for the willingness to participate, but also for the excellent feedback they provided.

Finally I want to thank my family and friends for their support, not only during the time of writing this thesis, but through all the years I spent in college.
Abstract

Code reviewing is the process to review code modifications by a peer in order to reduce the risk of regression (also known as a "pull request" in GitHub).

Historically, the review process is based on the analysis of textual change of the code and the status of the execution of the tests (continuous integration). With this project, we explored a new way to review the code modifications by looking at the execution impact that the code modifications have on the system.

To accomplish this, we implemented a tool - Code Review Helper - that takes a pull request from GitHub, runs it and compares it to the stable version of the project, analysing its impact on the system's execution, and presents the most relevant information in order to assist the reviewer in the review process.

This information is related to the coverage of the lines altered by the pull request we are analysing, the changes introduced to the coverage of other areas of the project, flame graphs to visualize execution traces of the software, and test times in order to identify possible performance slowdowns (or possible improvements).

To assess the effectiveness of our tool, we asked a group of users to perform a code review using our tool and asked them how our views compare to the ones currently available on GitHub, with mostly positive results, especially regarding the use of code coverage and test times information in their code review.

Keywords

Pull Request; Code Review; Code Modification; GitHub
Resumo

Uma revisão de código é o processo de rever modificações de código com o objectivo de reduzir o risco de introduzir uma regressão (também conhecido como “pull request” no GitHub).

Historicamente, o processo de revisão é baseado na análise das mudanças textuais do código e do estado da execução dos testes (integração contínua). Com este projecto, explorámos uma nova forma de rever modificações de código olhando para o impacto que estas têm na execução do sistema.

Para este efeito, implementámos um novo sistema - Code Review Helper que, corre um pull request do GitHub e compara-a à versão estável do projecto, analisando o seu impacto na execução do sistema, e apresentado a informação mais relevante para ajudar no processo de revisão de código.

Esta informação está relacionada com a cobertura das linhas alteradas pelo pull request que está a ser analisado, as mudanças introduzidas à cobertura noutras áreas do projecto, flame graphs para visualizar traces de execução do software, e tempos de teste que permitem identificar possíveis abrandamentos de performance (ou possíveis melhorias).

Para avaliar a eficácia da nossa ferramenta, pedimos a um grupo de utilizadores para a utilizarem numa revisão de código e perguntámos-lhes como é que as nossas views se comparam às que existem no GitHub, com resultados maioritariamente positivos, principalmente em relação à informação relacionada com cobertura e tempo de execução dos testes.

Palavras Chave

Pull Request; Revisão de Código; Modificação de Código; GitHub
Contents

1 Introduction 1
  1.1 Motivation ................................................. 3
  1.2 Goals ....................................................... 4
  1.3 Contributions ............................................ 5
  1.4 Document Structure ...................................... 6

2 Related Work 7
  2.1 Code Reviews ............................................. 9
    2.1.1 Code Review Processes and Quality ................. 9
    2.1.2 Code Reviews on GitHub ............................ 10
  2.2 Programs’ Executions .................................... 10
    2.2.1 Visualization ....................................... 11
    2.2.2 Finding Differences in two (or more) Executions .. 12
  2.3 Coverage .................................................. 13

3 Code Review Helper 15
  3.1 Overview .................................................. 17
  3.2 Service Package .......................................... 18
    3.2.1 Technical Details .................................. 18
    3.2.2 Overview ............................................. 18
    3.2.3 Authentication ....................................... 19
    3.2.4 Download Git Projects .............................. 19
    3.2.5 Filter Lines Changed by a Pull Request .......... 20
    3.2.6 Retrieve File Names in a Repository ............... 21
    3.2.7 Find Pull Request Files ............................ 21
  3.3 Core Package ............................................. 21
    3.3.1 Test Cases .......................................... 22
    3.3.2 Running Tests ....................................... 22
    3.3.3 Test Times ........................................... 23
## List of Figures

2.1 Example of a flame graph ........................................... 11
3.1 Project packages and their interactions .......................... 18
3.2 Example of a standard git diff from where we’ll filter the added lines only ........................................... 20
3.3 Excerpt of a json file detailing the time differences for each test between versions ........................................... 24
3.4 Excerpt of a json file detailing the coverage differences between versions. The full file details all coverage counters for all classes of the project ........................................... 26
3.5 Example of a relatedAreas.json file ............................... 28
3.6 Excerpt of a deadCode.json file ..................................... 28
3.7 Code Review Helper’s download projects step .................. 37
3.8 Core folder after analysis step is run ............................. 38
3.9 Process of generating flame graphs .................................. 39
3.10 Code Review Helper’s homepage .................................... 40
4.1 Users opinions regarding ease of use of our tool .............. 45
4.2 Users opinions regarding whether the tool is an improvement when compared to GitHub ........................................... 46
4.3 Users opinions regarding how easy the code coverage views were to understand ........................................... 46
4.4 Users opinions regarding how easy the test times views were to understand ........................................... 47
4.5 Users opinions regarding how easy the flame graphs were to understand ........................................... 48
A.1 Main view (Section 3.4.3) ............................................. 60
A.2 Related Coverage (Section 3.4.4) .................................... 61
A.3 General Coverage (Section 3.4.5) .................................... 62
A.4 Test Times (Section 3.4.6) ............................................. 63
A.5 Flame Graphs (Section 3.4.7) ......................................... 64
Acronyms

CSS  Cascading Style Sheets
D3   Data-Driven Documents
HTML Hypertext Markup Language
IDE  Integrated Development Environment
POM  Project Object Model
SVG  Scalable Vector Graphics
XML  Extensible Markup Language
Introduction

Contents

1.1 Motivation ................................................. 3
1.2 Goals ....................................................... 4
1.3 Contributions ............................................ 5
1.4 Document Structure ................................. 6
This thesis presents a new way to review changes introduced by a pull request that’s based on the execution impact that said changes have on the system, as opposed to the traditional ways that are solely based on the analysis of the textual changes and whether the tests are passing or failing. The goal is to make the code review process more accurate and efficient, leading to an improvement in the code quality of repositories.

In this chapter, we expand on the motivation and goals for this thesis, describe the contributions that were achieved, and explain the structure of the document.

1.1 Motivation

In software development, code reviewing is the process in which a reviewer assesses the quality of another developer’s contribution before it gets added to the project's code base. Reviewers can reject contributions that would hurt the project, request changes to contributions before they’re added, or accept contributions that would be a positive addition. Essentially, code reviewers are responsible for evaluating and shaping individual contributions, and, in doing this, they contribute to the overall code quality of the project.

This concept is applied in the real world in many different instances. One such instance is a GitHub pull request. In a GitHub pull request, a contributor pushes his changes and contributions to a repository. This can be used in open source projects, for example. In that case, pull requests serve as a way to notify project maintainers about contributions that have been made, and to organize and initiate the code review and discussion process before the changes are potentially merged onto the main branch. Traditionally, this process is based on the textual changes of the code and the current status of the execution of the tests - our work is focused on exploring other potential ways to improve this process based on the behavioral impact introduced by a pull request.

On GitHub, when working on his own branch, a developer can introduce his own changes to a project using commits. If he then wants to merge those changes to the main branch (effectively adding his changes to the project), he can create a pull request. In order to do this, a developer must specify the branch that contains his commits and the main branch, along with a title and description for his pull request [1].

At a later time, a reviewer will review these proposed changes. In the current GitHub interface, a reviewer can review and discuss: commits, changed files, and the "diff" between the base and compare branches [2]. More specifically, a reviewer can leave comments on particular lines of code that have been changed (including recommendations for specific changes), along with a comment summarizing the review. A review can be either a comment (to provide feedback without explicit approval), an approval (this will merge the proposed changes to the main branch) or a request for changes (specific feedback...
Pull requests are not always successful, however. For example, if the changes proposed by a pull request are not needed, or if a different pull request presents a better solution, it is possible for a reviewer to close a pull request without merging it to the main branch. The branches associated with closed pull requests can also be deleted in order to keep the repository clean. This can also be done to merged pull requests. Deleted branches don’t fully disappear and can be restored later, if they’re necessary to the project again.

With this said, it becomes clear that the code reviewing process is a fundamental one when considering whether to add requested changes to a project. Good code reviews increase the likelihood of positive changes being merged in via pull requests, while ensuring that negative ones don’t make it to the project. This means that it is in our best interests to improve code reviews. Further, making the code review process as efficient as possible is desirable, as developers responsible for reviewing code often cite time management as one of the main challenges in code reviews [3]. If the time required to perform a good code review could be reduced, this issue would be alleviated.

Thus, our motivation is established. With pull requests and code reviews being such an integral part of software development, possible advances in how they’re conducted could be of huge benefit to code quality. In this case, we aim to explore a way to visualize changes that goes beyond the traditional diff and test status into something that gives a deeper look at how a pull request is affecting a project’s behavior during execution.

1.2 Goals

The main goal for this project is to analyse the impact of code changes on a project, and then showing the insights gathered to developers who are in the process of a code review, in order to help make this process as efficient and accurate as possible, which would in turn be a positive contribution to the quality of a project’s code base.

Many different types of changes can be made to a project, such as refactoring, addition of new features, or bug fixes, just to name a few. Sometimes, these changes can introduce bugs or regressions to the code, and it is important to find them as soon as possible. As such, the tool we developed, which complements the standard syntactic “diff” comparison with an analysis on the execution impact of a change can be used to prevent bugs or regressions introduced by that change from making it to the project’s code base. This is also one of the goals of a code review, so utilizing both in a complementary way, as we do in this project, makes sense.

The need for this complementary usage is especially clear in pull requests that don’t introduce any behavioral changes (for instance, pull requests that only change the documentation), which are better
reviewed using the traditional syntactic diff, so this is something that our tool also displays.

As such, our goal becomes clear: our tool aims to present details regarding the execution impact of a pull request to the reviewer in a way that helps in the code reviewing process. In particular, this view will add to the traditional information displayed by GitHub (commits, changed files, and the “diff” between the base and compare branches) with additional in-depth information regarding how a pull request affects the coverage, the performance (based on test times) and the execution profile of the project, with the goal of aiding the review process. After developing this tool, we ask a group of users to perform a code review using our tool, with the goal of assessing whether our tool is effective in helping the code review process or not.

1.3 Contributions

The tool that we developed with the purpose of meeting the previously defined goals has three main packages. Service - where all the work related to GitHub and its API is performed, Core, which executes the projects and parses the results, and Presentation, which is responsible for taking the information from the Core package and presenting it to the viewer.

As a result of our work, the following contributions can be identified:

- A system that automates the process of downloading the base version of a project as well as a version with the changes introduced by the pull request.

- A system that runs both downloaded versions (this is specifically about maven projects).

- A system that parses the outputs of the execution as they relate to:

  - coverage, also crossing this information with information from git regarding which lines were altered, in order to differentiate between coverage changes in areas of the code that were changed, and coverage changes elsewhere (possibly identifying unintended consequences of a pull request)

  - test times, comparing both versions in an attempt to find discrepancies that could uncover previously unnoticed performance differences introduced by the pull request

  - the execution profile, in particular by parsing the stack trace of the executions into a format that’s compatible with a flame graph tool.

- A series of views that complement the traditional ones displayed by GitHub when reviewing a pull request, to improve the efficiency and outcomes of this process.
• An evaluation of the developed system that shows us how useful a system of this kind can be when applied to real reviews of real pull requests, with positive results, especially regarding the use of information related to code coverage and test times when performing a code review.

1.4 Document Structure

In the current chapter (Introduction) we have introduced the work we developed and defined the goals we intend to accomplish with it.

In Chapter 2 (Related Work) we present works that have been previously developed and how they relate and serve as a basis to our work. This is split into a few different sections, to clearly separate work that concerns itself with code reviews from work related to visualizing and understanding a program’s execution.

In Chapter 3 (Code Review Helper) we describe the solution we developed, from the ideas behind it, to the implementation and technical details. This chapter is split into four sections, one for each different package (Service, Core and Presentation) which then contain the specific details of each package in greater detail, and one detailing the online availability of the code we developed.

In Chapter 4 (Evaluation) we show the methodology that we used to evaluate the effectiveness of our work in meeting the established objectives as well as its results.

In Chapter 5 (Conclusion) we assess the results of our work, including its strongest points and limitations, how we did in accomplishing the proposed goals, and finally we detail which specific areas can still be developed in future related work.
Related Work

## Contents

- 2.1 Code Reviews .................................................. 9
- 2.2 Programs’ Executions ........................................ 10
- 2.3 Coverage ......................................................... 13
In this chapter we present some of the research that has been developed in this field. We have three sections for the related work concerning code reviews and how to improve them, some ways to visualize program’s executions, and some work related to code coverage.

2.1 Code Reviews

First, we present the research related to code reviews. We split this into two subsections, one with the research related to code review processes and quality, and one with research specifically related to pull requests on GitHub.

2.1.1 Code Review Processes and Quality

A substantial amount of research has been developed in an attempt to assess code review quality and how to improve code review processes. Kononenko et al. [3] surveyed 88 Mozilla core developers with the goal of understanding the factors that affect time and decision of a review, developers’ perceptions of code review quality, and challenges that reviewers face when reviewing code. It was found that the perception of code review quality is mainly related to the thoroughness of the feedback, the quality of the code, and the reviewer’s familiarity with it. In regards to the main challenges a code reviewer faces, it was found that the biggest ones were managing personal priorities, maintaining their technical skill set, and mitigating context switching.

Wang et al. [6] implemented peer code review processes in an academic environment over two full academic years. Some challenges were encountered, such as lack of qualification from the students to do code reviews, conspiracy activities among the students, and a difficulty to control the entire process. To solve these issues, it is suggested that a blind review mechanism could be implemented, along with a game theory model.

In an attempt to find what constitutes a good code review, Bosu et al. [7] analyzed 1.5 million review comments from five Microsoft projects, finding, among other things, that reviewer selection is very important (reviewers familiar with the code base produce better reviews), and that the code changes themselves also affect review quality, as changing too many files at once makes the review comments less useful. To solve the issue of reviewer selection, it is recommended that inexperienced reviewers are, at first, included in reviews with experienced reviewers so that they can develop the experience and knowledge required to write good reviews. On the developer side, it is recommended that developers submit smaller and more incremental changes as often as possible, rather than implementing a large feature all at once.

Further expanding on the issue of large changes affecting code reviews, di Biase et al. [8] performed a study to determine how the decomposition of changes impacts code review and found that smaller
changes focused on one specific fix or change usually led to better outcomes, not only in regards to finding bugs and suggesting changes, but also in regards to the behavior of the reviewer and how they attempt to find context for the changes made. As a result they conclude that commits belonging to different concepts should be separated, and such practices should be adopted as best practice in software engineering.

2.1.2 Code Reviews on GitHub

In regards to pull requests on GitHub, Rahman and Roy [9], looked into 78 GitHub base projects and their respective successful and unsuccessful pull requests. The findings included correlations between several factors and the success of a pull request. Among these factors are programming language, application domain, technical topics discussed, project age and maturity and project developers and experience. Some of the findings are that three programming languages (Java, Ruby and Javascript) have a higher rate of unsuccessful pull requests, that programmers between twenty and fifty months of experience are the most productive in terms of getting pull requests approved, and that older projects with more forks have a higher rate of unsuccessful pull requests.

Yu et al. [10] performed a study on which factors influence the latency of the evaluation of a pull request in GitHub. It was found that latency is a very complex issue that requires many different variables in order to be properly explained. It is found that the number of comments that are part of the discussion of a pull request are the best predictor of pull request review latency. Another very important factor is the size of a pull request, as shorter changes are consistently reviewed faster.

Gousios et al. [11] explored how pull based software development works on a sample of 291 projects. The main findings were that most pull requests affected just a few dozen lines of code and that 60 percent were reviewed and merged or rejected in less than a day. Pull requests that altered recently modified code were found to be less likely to be accepted. The time to review is influenced, among other things, by the developer's track record and the project's test coverage. And finally, only 13 percent of pull requests are rejected due to technical reasons.

2.2 Programs’ Executions

In this section we present the research related to programs’ executions, first regarding their visualization, and then regarding how to find differences between multiple executions.
2.2.1 Visualization

Some work in this area has explored different possible ways of visualizing the execution of a program. Gregg describes flame graphs [5] [12], which propose an inverted icicle layout to visualize a software profile. In these graphs, the x-axis shows the stack population sorted alphabetically (this is because flame graphs don’t represent the passage of time) and the wider a rectangle is, the more often it was present in the stack. The y-axis shows the depth of the stack. An open source implementation which takes information of a software profile and creates the corresponding flame graph is also provided by the author. Figure 2.1 is an example of a flame graph.

The visualization of large traces can often be problematic. In order to address that problem Trümper et al. [13] propose icicle plots and edge bundles to visualize execution traces along with an attempt to solve the issue of visual scalability in a visual tool called Tracediff, which allows the comparison of large execution traces. This is done with a multiscale visualization method where users can go from the main differences in a program, to an intermediate level, and then all the way to a fine grained level at each function.

Cito et al. [14] present Context-Based Analytics, an approach that links runtime information to code
fragments and makes their relations explicit. This makes it easier to diagnose issues in large scale software, as they often come from development bugs, and identifying the code fragments responsible for those bugs can be a huge challenge. In a test with issues from IBM’s issue tracker, it was found that this approach could reduce the number of analysis steps taken before diagnosing an issue by around 48 percent.

For situations where concurrence is involved, Karran et al. [15] propose SYNCTRACE, which allows a new way to visualize multiple threads and their inter-thread correspondences with a technique based on activity diagrams and the aforementioned edge bundles, which make it easier to understand concurrent behavior.

### 2.2.2 Finding Differences in two (or more) Executions

Bezemer et al. [16] present differential flame graphs, which allow the depiction of the differences between two flame graphs [12] and provide an open source implementation of the approach, Flamegraphdiff. This has the goal of improving the process of comparing two (or more) different flame graphs, which can sometimes be difficult and tedious if done by hand, especially for larger graphs where the differences are subtle.

Alcocer et al. [17] propose performance evolution blueprint, a visual support that provides a way to compare multiple executions of software in an attempt to help understand the root of a performance slowdown. This visual support includes a visual metaphor where methods in red consume CPU the most, while methods in green are the opposite. It also presents run-time metrics that can help to better understand the reasons behind a performance slowdown.

Lahiri et al. [18] present SYMDIFF, a tool for equivalence checking and the display of behavioral differences over imperative programs. The tool is presented for the C programming language, but is language agnostic and can be extended to other languages, because it uses an intermediate verification language, Boogie, which can be translated to several different source languages. Lahiri et al. [19] describe the approach of Differential Assertion Checking to compare different versions of a program and how they behave in regards to a group of assertions, in an attempt to find whether there is an environment where the original version of the program passes the assertions, but the modified version doesn’t.

Fedyukovich et al. [20] present eVolCheck, a tool that incrementally checks software as it evolves. Instead of verifying systems from scratch on every update it focuses on the actual changes that were made. eVolCheck does not check for equivalence, unlike the aforementioned SYMDIFF. Instead it only checks whether bugs have sneaked into the code.

Danglot et al. [21] suggest DCI (Detecting code changes in Continuous Integration) which analyzes the test cases of the pre-commit version and selects the ones that exercise the parts of the code modified
by the commit, and then generates variants (test amplification). This is used in order to detect the behavioral changes introduced in the system by this particular commit.

Finally, and in regards to potential merge conflicts in software (also an important aspect to take into account when reviewing pull requests), Guimarães and Silva [22] present a tool called WeCode, which detects conflicts on behalf of the developers, introducing the concept of continuous merging inside an Integrated Development Environment (IDE), essentially allowing developers to resolve conflicts while the changes are still fresh in their minds.

### 2.3 Coverage

Evaluating the coverage of code is useful as it allows the identification of which areas of the code did not execute in a run of the program. The work in this area includes:

Wong et al. [23] studied the effects of test set size and code coverage on the fault detection effectiveness of a test suite and their results indicate that the correlation between code coverage and fault detection is higher than that of test set size and fault detection.

Kochnar et al. [24] measure the relationship between code coverage and its ability to kill real bugs from software systems. Using randomly generated test suites with differing levels of coverage and real bugs from two large systems they found a statistically significant correlation between code coverage and bug kill effectiveness. A correlation between test suite size and bug kill effectiveness was also found, but it is not as strong.

Gopinath et al. [25] ran an experiment on Java programs hosted on GitHub, with human generated test cases that were directly collected from the GitHub repositories, as well as test cases that were automatically generated by the Randoop testing tool. They found that in both cases, statement coverage does a good job in predicting suite quality. On the other hand, suite size and code complexity don’t turn out to be important.

On the other hand, Inozemtseva and Holmes [26] generated 31000 test suites for five systems consisting of 720000 lines of code and found that when the number of test cases in the suite is controlled for, there is only a low to moderate correlation between coverage and test suite effectiveness.

Chen et al. [27] present a method to measure software reliability that is based on time and code coverage measures. (Software reliability is the probability of a software not failing in a given environment in a given time interval). As coverage correlates with software reliability, and failures that occur later are less likely to cause the program to fail, their method shows potential to correct the overestimation of reliability that often occurs.

This concludes the related work section of this document. In the following chapter, we present our solution - Code Review Helper.
Code Review Helper

Contents

3.1 Overview ......................................................... 17
3.2 Service Package ............................................... 18
3.3 Core Package .................................................... 21
3.4 Presentation Package .......................................... 29
3.5 Algorithms ....................................................... 33
3.6 Motivation Example ........................................... 36
3.7 Online availability ............................................... 39
3.1 Overview

In order to achieve our goal of reviewing code modifications based on the impact that they have in the behavior of the program, we present a new way to visualize the changes made in a pull request on a GitHub project.

GitHub is one of the most popular websites to host code, and it supports pull based development, making it a perfect fit for the work we are developing. Currently, the process of reviewing a pull request on GitHub takes into account the commits, the changed files and the differences (diff) between the files in the base branch compared to the pull request branch. Our solution adds a new way to visualize the changes based on the behavioral impact that they have on the system.

To do that, we worked with the current version of the official GitHub API (GitHub REST API v3) [28]. This API allows developers to perform a number of different operations on pull requests, some of which are of particular interest to us.

Firstly, it allows us to access any of the pull requests in a project by providing their number, which is how we will interact with the pull request we want to review. Given the pull request number, we can get all the details of that specific pull request, including the files altered, and which specific lines were altered, and how. This will be the crucial step for our tool, and where our usage of the API will be the most useful, as it allows us to get all the changes we want to analyze to our local machine and work from there.

Given this, it is clear that the GitHub API was crucial for the work we developed, essentially making it possible for there to be a connection between our system and the projects and code hosted on GitHub in order to extract the information we require.

The project was developed using the Java programming language, and we focused on analysing Java projects hosted on GitHub, particularly the ones that use maven. Further, and when it comes to presenting the information gathered, we use spring-boot, thymeleaf, and the D3.js library. We will expand on how these are useful later.

Code Review Helper has three main aspects to it: it connects to GitHub and gets the information of a pull request, it analyzes the behavioral impact caused by the changes made by that particular pull request, and it presents the insights gathered to the reviewers.

As such we split it into three main packages: one to connect with git and retrieve all the required information, one to run an analysis on the behavioral changes introduced by the pull request, and one to present the most relevant information to the reviewer. We called them Service, Core and Presentation, respectively. A very basic overview of how they interact is present in figure 3.1

The rest of the Solution chapter of this document is structured so that each of the packages has its own section, where we go in depth into how they’re implemented and how they work in connection to each other.


3.2 Service Package

The service package is the package responsible for all the interactions with GitHub. It contains all the functionalities relating to git itself, essentially working as a "bridge" between our project and the git repositories. The logic for all communication required between our work and git is implemented here.

3.2.1 Technical Details

To interact with the GitHub API we utilized the egit Java library (org.eclipse.egit.github.core). [29] The library is composed of three main packages: Core, Client and Service. The Core package models the resources available through the GitHub API, the Client package contains the classes responsible for the communication with the API itself, over HTTPS, and it models the JSON responses into the appropriate Java model classes, and the Service package contains the classes that invoke the API calls.

This library supports the GitHub v3 API (the API we're using), with the goal of 100% support. This made it a good choice for this project, and allows the connection between our code and the GitHub API to be easier and more intuitive than it otherwise would be.

3.2.2 Overview

This package consists of two classes, serviceApp and changedLines.

- The serviceApp class is the main class in this package and contains the majority of the logic that's
run here.

It stores authentication details as well as the files that were changed and a list of all repository files in its attributes.

- The `changedLines` class is an auxiliary class that simply stores a file's information regarding how it was changed by a pull request.

It stores the file name in a String, and the actual changes in a Map where the key is an Integer referring to the line number, and the value is a String containing the changes that were made.

### 3.2.3 Authentication

A non-authenticated user has stricter rate limits when making requests to the git API. For comparison, an unauthenticated user can perform 60 requests per hour, while an authenticated user can perform up to 5000, in the same time period [30]. An unauthenticated user will also have no access to private repositories.

In order to deal with this, our original solution proposal included the possibility to authenticate, and so did earlier versions of our tool. On a technical level, this was allowed by the egit library, which has a `setCredentials` method, which receives the username and password. We would set this upon the creation of our Service object, which contains an object of the type `GitHubClient`, which is where these credentials were stored.

However, this method of authentication relied on the user inputting their user/password combination, which will unfortunately no longer be supported by Github, starting later this year [31].

In order to make our application future proof, we replaced this with an OAuth2 token. For simplicity purposes, we use our own personal access token whose only permissions are read access to public repositories.

### 3.2.4 Download Git Projects

The first step in our analysis is to download the project in question to our machine. This is necessary because we need to run its tests. Downloading the stable version of a project (the one on the main branch) is basically the same as doing a standard `git clone`, so we do it by issuing a standard command line process. Then, we need to download a version that includes the changes introduced by a pull request. First, we copy the version we already downloaded to a new folder (called PRVersion, short for Pull Request Version) using robocopy [32] (which comes standard with all Windows distributions), and then we apply the pull request changes by providing its number which allows us to fetch them from GitHub.
By the end of this process we will have two local folders: stableVersion and PRVersion. One has the main branch version of the project, while the other has the pull request changes applied. These are the folders where we’ll be running our analysis on.

### 3.2.5 Filter Lines Changed by a Pull Request

The service package is also responsible for filtering which lines were altered by a pull request. This was necessary for us due to the coverage analysis run by the Core package, which will check whether the changes added by a pull request are covered by tests. In order to do this the Service package needed to be able to parse the changes made by a pull request, find which lines were added, and save them in a structure where the line numbers are associated with the content that was added.

The changes introduced by a pull request are usually presented in the standard git diff. When accessed through the API the diff is as seen in figure 3.2. This is readable for a human, with file names presented first, followed by line numbers, and then the changes themselves, with additions in green and deletions in red. However, this needs some treatment in order for a computer program to deal with it. Our service package parses it into a local structure that only contains the information we’re interested in (the line numbers that were added, and their respective content). The result is then used to check whether these changes are covered (more on this in the Core package section).

All this work is performed in the `serviceApp` class by the `filterChangedFilePatchAdditions` method, which takes a git diff like the one shown and the filename, and returns an object of the type `changedLines` (a class we created for this project) which contains the information we described in a Map of Integers and Strings, where the Integer is the line number that was added, and the String is the addition that was made.
3.2.6 Retrieve File Names in a Repository

Another function programmed in our Service package is the ability to retrieve all file names in the repository that's being analysed. We need this information because we will be comparing the impact on all files in the project (particularly the coverage), not just the ones that were altered, as sometimes a pull request can have a far reaching impact. So the Service package traverses the repository recursively and gives us information regarding all files in the repository.

When designing the solution we realized that this could also be done locally, because we download the project to the user's machine, and doing so might actually be faster than querying GitHub for every file. However, when thinking about the solution as a whole, we realized that presenting links to a file's GitHub location would be a good idea, as we'll be showing execution differences, but a user might want to go to GitHub in order to get the full context of the file. So when getting the files’ information from GitHub, we actually get and store their respective links (knowing that the file name is located after the last ‘/’ separator), which will be useful for the Core and Presentation packages, as we'll explain more in depth later.

This operation is performed in the `retrieveRepositoryFilesNames` method, and the output is a simple list of strings, which get stored as the attribute `repoFiles` of the `serviceApp` class.

3.2.7 Find Pull Request Files

As an addition to the last subsection, our system also identifies which files were changed by the pull request. This is also done when analyzing the git diffs, as explained earlier. In essence, we end up with the entire repository being categorized in two ways: one the files that were altered, and the other is the files that weren’t.

The reason for the way our Service package classifies the files and the changes will become clear in the next two packages.

We also create a method `findFile`, to be used by the Core package, which receives a file name and returns its GitHub link, which we stored as explained in the previous section.

This also concludes our Service package section. Next, we’ll explain the Core and Presentation packages, and how they relate to each other (and with the Service package).

3.3 Core Package

The core package is responsible for the majority of the work in our system. This includes executing both versions of the project that's being analyzed, recording their execution profiles and analyzing the outputs in regards to coverage and execution times. In short, it runs all the logic necessary to assess how a pull
request changes a project and stores all the information gathered so that the Presentation package can take it and present it to the reviewer. The following subsections will explain this in detail.

3.3.1 Test Cases

One of the assumptions we make is that the project being analyzed already has a good test suite, as our insights come from the execution of these tests. In an early proposal of our solution, we considered implementing test generation, but found that it would become too complex, and that the problem of test generation alone could be its own thesis.

Our tool does in some way provide insights regarding this, though, as we show coverage across the project and in the areas that were altered by the pull request. Considering that good code coverage is associated with good test suites and better fault detection [23], including in GitHub projects [24], it stands to reason that our analysis, which relies on the existing test suite will be more effective if the general coverage of the project is higher.

This is something to keep in mind when using the system in a real world situation or even in our Evaluation step - the choice to assume the existence of a good test suite could skew the results when analyzing projects that aren't being properly tested.

3.3.2 Running Tests

The first thing our core package does is run the tests on both versions of the project. For this we have to take a few things into account:

Firstly, we run the tests using the following command: `mvn test -fn`

You’ll notice that it is a standard maven test command, with the added `-fn` flag. This flag means "fail never", and it allows the tests to continue running even if one (or more) of them fails. This is useful for us, as we want to see the behavior of all tests, and it then allows us to inform the user on how the tests that succeeded behaved, even if one (or more) of the tests fail. Without this flag, if one of the earlier tests fail, the execution is completely stopped, which means that our analysis would be incomplete, because we’d only have the results of the tests until that point. This also ensures that even if the stable version of a project on GitHub isn’t currently passing all tests, we can still run an analysis over the pull requests.

Before running these tests, it is also important to know that two plugins will be required for our analysis: the Maven Surefire Plugin [33], and JaCoCo [34]. The reason for this is that some of our analysis will be based on the outputs of these two plugins, so it is required that the project we’re analyzing has them in their Project Object Model (POM), in both versions, in order for the required outputs to be generated. In a lot of cases this won’t be a problem, as these plugins are widely used, which includes many GitHub projects. For the ones that don’t use them, the developer should add the required
information to the POM before running this step of the program.

With all this taken into account, the process of running the tests is fairly straightforward, and is entirely automated, as our core package will first navigate into the folder containing the stable version and run those tests, and then it will do the same for the pull request version. Everything explained above so far in this section is executed in the \texttt{runStableVersionTests} and \texttt{runPRVersionTests} methods. At the end of this step, we’ll have gathered all the required outputs for our analysis.

This analysis can be broken down into a few different topics: test times, flame graphs, and coverage. We’ll explain how each of these work in the following subsections of the document.

### 3.3.3 Test Times

After running each versions’ tests, the first thing we analyze is each version’s tests times. The role of the core package here is to find the test times for all the tests in both versions and calculate the differences for each specific test. For instance if a test called \texttt{exampletest} takes 1 second to run in the stable version, and 3 seconds to run in the pull request version, the core package would be able to tell us that the difference is 2 seconds.

It is possible that certain tests are added/removed in between versions, or that certain tests pass in one version and not the other. In this case, there are no time differences and the core package returns a ???, which signifies to the developer that something potentially went wrong and it might be worth investigating that specific test case.

The reason for us to run this is fairly straight forward: a look at the test times is the most basic type of analysis possible that can still identify performance discrepancies. If one or more tests present a significant difference in their test times between versions, it is a potential sign that there’s a big behavioral difference and that it is worth it for the reviewer to look at that specific test and attempt to understand the reason behind that slowdown/speed up.

Of course, test time discrepancies don’t always mean that there are significant behavioral differences. Differences in performance, especially smaller ones, can be explained by many factors. One of the more common ones we found were tests that accessed the cache: we run the stable version first and those times are slow, but then the pull request version is quicker because many necessary values are already present in the cache. To alleviate this, we give the developer the option to run the tests more than once and then compute the average times, which should give a more accurate representation of the performance.

In order to get the test times, we need to parse the outputs of the Maven Surefire plugin, which contain them. These are usually in the \texttt{target/surefire-reports} folder, which our core package finds by running through the project recursively (there can be one of these folders for each package, for example) and then it runs through all the Extensible Markup Language (XML) files, finds the relevant lines that contain
Figure 3.3: Excerpt of a json file detailing the time differences for each test between versions

The following is an excerpt of a json file detailing the time differences for each test between versions:

```json
{
  "name": "testServer[1: firefoxDriver]",
  "status": "-0.4910000000000001"
},
{
  "name": "testTarget[6: 3.14361]",
  "status": "0.005"
},
{
  "name": "testWebDriver[1: class org.openqa.selenium.firefox.FirefoxDriver]",
  "status": "-0.09"
},
{
  "name": "testMockProx",
  "status": "-0.3710000000000044"
},
{
  "name": "testWrongVersion[0: class org.openqa.selenium.chrome.ChromeDriver]",
  "status": "4.25100000000001"
}
```

Figure 3.3: Excerpt of a json file detailing the time differences for each test between versions

Test times, and stores the name of the test and the time (in seconds). Finding all these times would be highly time consuming for a human, but can be done fairly quickly by a machine. We implemented this in the `stableVersionTestPerformance` and `pullRequestVersionTestPerformance` methods.

Once this is done for both versions, it is time to compare the times. We match the tests in both versions by their names, and calculate the difference in time for each test. This, again, would be highly time consuming for a human but is done quickly here. The `compareTestTimes` method is responsible for this. As both versions’ test times are stored in Maps, this method simply has to match their keys that have the same name, as they refer to the same test, and then calculate the difference between the values, which correspond to the time it took to execute the test.

With this done, the core package has all the information necessary in order to produce the outputs that the presentation package will use. These come in the form of two json files. One details the test times for the stable version, and the other details the time differences between versions, where negative values are improvements after the pull request modifications, positive values are slowdowns, and a `??` signifies that the test only ran properly in one of the versions. The work required to accomplish this is performed by its own class, which we named `OutputTestTimes`. This is a very simple class that only contains the logic to parse the Maps containing the results into the specified json file. It gets called twice, as each execution of this method returns one of the json files.

An excerpt of one of these json files is present in figure 3.3.
These json files are already fairly readable for a human, but will later be processed by the Presentation package in order to make their analysis even easier.

### 3.3.4 Coverage

Something else we analyze is the project’s coverage. We split this into two different analysis: one where we simply look at the lines that were added or altered in a pull request and whether they’re covered, and another where we look at the coverage of the project in general and see if there are differences between the stable version, and the pull request version. This gives us an insight into what areas of the code were and weren’t executed, which allows us to understand whether the new changes are being tested, and whether the execution of other areas of the code changed because of the them.

It is important to note that a part our coverage analysis is based on the outputs generated by the previously mentioned JaCoCo plugin. These are the outputs we’ll be parsing in order to get part of the information we’ll be looking for, and for that reason it is important to understand what their coverage metrics are. Their coverage counters are available in their documentation page [35] and can be summed up as follows:

- **Instructions**: the smallest units counted, they refer to single Java bytecode instructions.
- **Branches**: these refer to switch and if statements, and can have no coverage, partial coverage (when only a part of the branch is executed), or full coverage.
- **Cyclomatic Complexity**: refers to the number of unit test cases required to cover all paths of a method.
- **Lines**: information for each individual line, and can have no coverage, partial coverage, or full coverage. This is where most of our analysis will center around.
- **Methods**: a method is considered to be covered when at least one of its instructions was executed.
- **Classes**: a class is considered to be covered when at least one of its instructions was executed.

With this established, we can now describe how the parsing of the coverage XML files works. This is performed in our `coverageAnalysis` class, which, as the name suggests, is responsible for most of the coverage operations described in this section. The first method we use is called `parseReports` and its purpose is to give us a general overview of the project. It simply goes through the generated XML files relating to coverage and gives us a report of the aforementioned coverage counters for both versions and also calculates the differences. This is later exported to three json files (one for each version and one for the differences) to be used by the presentation package. Figure 3.4 is an excerpt of a json file.
Once this is done, the next step in the execution is the `checkIfChangesAreCovered` method, which is also part of the `coverageAnalysis` class. This method crosses the information from the Core package and the Service package in order to understand whether the changes that were added by a pull request are being covered by the tests. First, it uses the `getPullRequestChanges` method, which returns a list of the lines that were added by a pull request (this method calls the aforementioned `filterChangedFilePatchAdditions` method). Then it takes these lines and uses them as input in the `parseChanges` method.

The `parseChanges` method is then responsible for going through the generated XML files (similar to `parseReports`), but it specifically looks for the lines that were added by the pull request. Due to the way the XML outputs are formatted, this requires three `for` cycles: one to find the right file, one to find the right method, and then the last one to find element containing the right line. This element contains information regarding the instructions and branches of the line, and whether they were covered.

This information is put into a list of objects of the `ChangedLine` class. This class contains all the information regarding a line that was changed by a pull request. Its attributes are:

- `filename`: Which file was changed.
• change: The actual textual change.
• missedInstructions: How many instructions aren’t covered in this line.
• coveredInstructions: How many instructions are covered in this line.
• coveredBranches: How many branches are covered in this line.
• missedBranches: How many branches aren’t covered in this line.
• lineNumber: The number of the line that was changed.

This list of changedLine objects containing all the information regarding the lines added by the pull request is then stored alongside the git diff of the pull request, to later be used by the Presentation package.

Finally, we run the method findDeadCode also from the coverageAnalysis class. This works similarly to other described methods in regards to how it parses the coverage information, and its goal is to compare the coverage of both versions in the areas of the code that weren’t altered by the pull request, and also to find the specific lines where there are differences. This means that we can identify which lines were being executed but no longer are, or vice versa, and allows us to potentially find unintended consequences of the pull request changes.

In addition to this, this method utilizes information from the Service package, as it saves the names and github links of the files where the coverage changed despite the pull request not altering their code. More information on how the Service package does this is present in section 3.2.5.

At the end, we’ll have two more json files: relatedAreas.json and deadCode.json. The first one contains the aforementioned information taken from the Service package regarding the names and links of the identified files, while the second expands on that information by detailing the specific lines where coverage changed, their content, and how the coverage changed. The reason for this separation will become clear when explaining how the Presentation package works. Figure 3.5 is an example of a relatedAreas.json file and figure 3.6 is an excerpt of a deadCode.json file.

3.3.5 Flame Graphs

Lastly in the Core package section, we describe the Flame Graph implementation. This is comparatively less complex, as we rely on an external tool [5] to generate the graphs themselves. As such, the requirements here are:

• Get the required information from the execution of the projects in order to build a flame graph

• Structure that information in such a way that it can be dealt with by the tool we’re using to build the graphs
Figure 3.5: Example of a relatedAreas.json file

```
{
  "name": "Preferences.java",
  "link": "https://github.com/bonigarcia/webdrivermanager/blob/master/src/main/java/io/github/bonigarcia/wdm/Preferences.java"
},
{
  "name": "HttpClient.java",
}
```

Figure 3.6: Excerpt of a deadCode.json file

```
{
  "name": "Preferences.java",
  "children": [
  {
    "name": "Covered Instructions",
    "value": -1
  },
  {
    "name": "Missed Instructions",
    "value": 1
  },
  {
    "name": "Covered Branches",
    "value": 0
  },
  {
    "name": "Missed Branches",
    "value": 0
  },
  {
    "name": "Line Number",
    "value": 70
  }
  ]
}
```
To satisfy the first requirement we simply need to configure our project in a way that allows for the production of a Java Flight Recording [36] containing the data relating to the profile of the projects' executions at specified intervals. This can be done by creating a .mvn folder containing a jvm.config file containing the line:

\[-XX:StartFlightRecording=delay=20s,settings=profile,filename=recording.jfr\]

This means that the events that happen during the execution of the project we’re analyzing will be recorded. The result of these recordings will be two files called recording.jfr, one of them referencing the execution of the stable version, and the other will reference the execution of the pull request version. These .jfr files are then parsed by our program by using the jdk.jfr library [37].

We do this in the generateTraces method of the coreApp class. First we utilize the aforementioned jdk.jfr library to extract the information of the stack traces that was recorded, by calling the getStackTrace method inside a for cycle that goes through all the RecordedEvent objects present in the file. Then, we write them to a file called stableProfile (for the stable version, the pull request version we call prprofile).

These files are formatted so that they’re compatible with the external tool [5] we use to generate the flame graphs, satisfying the second requirement we had identified. This means that we simply need to run the information we generated through the tool and we’ll have the flame graphs we wanted. These will be used by the Presentation package, as explained in the following section.

### 3.4 Presentation Package

Finally, we have the presentation package. This is the package responsible for displaying the information gathered by the Core package to the code reviewer.

#### 3.4.1 Overview and Technical Details

For the presentation package, we created an application using Spring Boot [38] and Thymeleaf [39], with some of the information being displayed using the D3.js [40] library.

Their roles in the package are as follows:

- Spring Boot is an open source Java-based framework used to create Spring-based Applications. One of the big advantages of Spring Boot applications is that they need very little configuration, as they’re preconfigured already, meaning that Maven configurations regarding POMs, or tomcat integrations for example, are all available from the go.

This is perfect for us, as we’re not building an overly complicated application that requires specific configurations. The goal of our presentation package is simply to display information to the code

29
reviewer in a simple and perceptible way. We also had previous experience using this library that told us it was a good fit.

• Thymeleaf is an open source java library that functions as an Hypertext Markup Language (HTML) and/or XML template engine. It essentially applies a set of transformations to HTML views in regards to how the data is displayed.

  Thymeleaf is fully compatible with HTML, making it a good choice for us. Further, previous experience with this library in the Software Engineering course, gave us the knowledge required to use it without a learning curve, and the confidence that it could help in realizing the vision we had for the Presentation package.

• D3.js (also known as simply Data-Driven Documents (D3)) is a JavaScript library for producing interactive data visualizations with HTML, Scalable Vector Graphics (SVG) and Cascading Style Sheets (CSS). D3 emphasizes web standards in order to give developers the full capabilities of modern web browsers.

  Our need to visualize information, along with the fact that we’re creating a web application, makes D3 a compelling choice, as it is a powerful visualization tool compatible with modern web standards. Furthermore its website has an extensive gallery of examples and its GitHub page has extensive documentation, making it a good choice for us due to lack of experience utilizing any library intended for data visualization.

Regarding the actual application we’re developing, it can be summed up as a series of different views, each providing a different insight on the pull request being analysed. This is where we fulfill our goal of creating views that add to the traditional information displayed by GitHub (commits, changed files, "diff" between branches). The description of each of the views we’ve created are in later subsections. Further, appendix A features a screenshot of each view. But before getting into their details, it is important to first describe the structure of the package, including what files are present and how they work with each other. This is done in the following section.

3.4.2 Structure

Our presentation package consists of a few different files. First, we can discuss the controllers. These are the java classes responsible for the creation of the views. In here we have a controller whose sole purpose is to start the Spring application, one controller that accesses the local json files that were created by the Core package, and then a controller responsible for the actual views themselves.

In addition, there’s some other complementary files that are necessary for the package to display the views correctly:
• One HTML file for each view, as expected for a web application

• Three CSS files, called codeTables.css, coverageTables.css and main.css. As the names suggest, the first two are responsible for stylizing specific tables we present, while main.css contains general styling options that we use across all views.

• Several Javascript files, responsible for some elements of the views. This includes the logic for the creation of tables detailing the difference between test times as well as the coverage trees, for example. It is also where the logic regarding the D3 library is present.

In the following sections, we will be detailing each of the views we created. As mentioned before, a screenshot of each of the views can be found in appendix appendix A.

3.4.3 Main View - Coverage on Changes

The first view we display to a reviewer contains the coverage on the lines that were altered by the pull request. This view also contains a list of files whose coverage was altered despite there being no textual changes made. In order to facilitate the analysis, each of the files on the list is a link that leads to its GitHub page where the full content of the file is available to be checked. The information detailing the exact coverage changes is in a different view, because we found that putting it in this view would have made it too cluttered and hurt the usability of our tool.

The information on the coverage for the lines that were altered contains a counter for instructions and branches of each line, as well as a standard diff comparing the pull request and main branch versions, similar to what you would find on GitHub. In order to produce this diff we utilized the aforementioned diff2html [41] generator.

For that reason, we chose this as the first view to display. Not only is the information regarding associated files immediately relevant, but the display of a diff means that even if the analysis on coverage doesn’t produce meaningful results, our tool will be no worse than the already existing solution on GitHub, where the same diff is displayed.

3.4.4 Coverage in Other Areas of Code

Another view the reviewer can choose (using a navigation bar we present at the top of each page) displays all the changes in coverage in areas of code that weren’t altered by the pull request. As explained before, the files affected are shown in the main view. Sometimes no files are affected, and in this case, this view will display a message saying that there are no changes in other areas of the code, and because of that there is nothing to display.
In the case that there are some files affected, this view takes the `deadCode.json` file (explained in section 3.3.5) and produces a view containing the information. Note that despite the file name, this doesn’t only identify code that stopped being used, it can also find code that is only now starting to be used.

As such, this view displays the excerpts of code along with the information on how the coverage of each line changed. In order to provide the necessary context, we also show a few lines before and after the lines where the coverage changed (like what GitHub does when they show a few lines of context in their diffs).

### 3.4.5 General Coverage

The last view regarding coverage, is split into three, each with a simple tree that displays the total coverage information of the project - one for the stable version, one for the pull request version, and one for the difference.

This takes each java file as a node of the tree and the coverage information for that file is inside the node. Each node is collapsible making it possible to choose which packages to visualize. Further, areas where the coverage is high are color marked as green, and low coverage areas are color marked as red, to further help the visualization. This is meant as a complement to the previous two views, as it is a much less in depth view.

### 3.4.6 Test Times

Regarding test times, we take the information present in the two json files described in section 3.3.3 to produce two tables: one with the test times on the stable version, and one with the differences in times between versions.

In order to make this easy to read, the table with the differences will be color coded: in red we’ll display tests that had slower performance, green will be tests that were quicker, and yellow will be tests that had no significant differences. This makes it easy to identify tests whose performance changed at a glance, which is especially useful for projects with a very high amount of tests, where carefully reading the table would be much harder than simply looking for the red/green squares to find the tests where performance changed.

### 3.4.7 Flame Graphs

Finally, we display both of the created flame graphs (as we described in section 3.3.5) in their own view. First, we display the flame graph that corresponds to execution of the stable version of the project. Below it, we display the flame graph corresponding to the execution of the pull request version.
Both of these graphs are fully interactable, which allows the user to expand certain areas of the stack by clicking on them, in order to better evaluate specific areas of the graph. It also allows the user to type in a method name that he’s looking for, which is then highlighted on the graph, if it exists.

3.5 Algorithms

In this section, we present the algorithms for certain more complex areas of a Code Review Helper execution, and expand on their complexity.

First, algorithm 3.1 details how Code Review Helper finds and compares times between both versions. This is a simple parsing of two files and comparing integers taken from both, the time and space taken is simply dependant on how many files and tests there are, as each of them implies an extra operation.

Algorithm 3.1: Finding and comparing test times between versions

```
Result: List of differences between test times on both versions - testTimeDifferences
stableTestTimes = Map{Key: testName, Value: testTime};
prTestTimes = Map{Key: testName, Value: testTime};
testTimeDifferences = Map{Key: testName, Value: testTime};
for file in stableVersionFolders do
    if file is testTimesFile then
        for node in file do
            add test to the stableTestTimes map
        end
    end
end
for file in pullRequestVersionFolders do
    if file is testTimesFile then
        for node in file do
            add test to the stableTestTimes map
        end
    end
end
for entry in stableTestTimes do
    if prTestTimes contains entry.getKey then
        compute the difference and add it to testTimeDifferences
    else
        testTimeDifferences.put(entry.getKey, "???")
    end
    // "???" because the test was only found in one of the versions of the project;
end
return testTimeDifferences
```

Algorithm 3.2 is how Code Review Helper filters the added lines from a GitHub diff. This is a more complex operation to understand due to the specificity of the contents of a GitHub diff, which means that
parsing it isn’t a straightforward operation. In particular, understanding which lines are additions, which aren’t, and keeping a count of the line number (GitHub only provides the starting number of a diff) are the most complicated aspects. Regardless, each line from the diff is only parsed one time, meaning that the algorithm scales linearly.

Algorithm 3.2: Filtering the added lines and their content from a GitHub diff

Result: GitHub diff output parsed into an object containing a Map with a relation between each added line number and its content

```
patchLines[] = patch.split("/n"); // splits the git diff by /n;
int lineNr;
boolean b = false;
linesAdded = Map<lineNr, lineContent>;
for line in patchLines do
  if line.startsWith("+") then // if line is an addition
    add line to linesAdded map;
  end
  if (!(line.startsWith("-"))) then
    lineNr++ // only increments if line is not a removal;
  end
  if line.startsWith("@@ -") then
    String lineNumber = "";
    for i = 0, n = line.length(); i < n; i++ do
      char c = line.charAt(i);
      if c == "," or c == "" then
        b = false;
      end
      if b = true then
        lineNumber += c;
      end
      if c == '+' then
        b = true;
      end
    end
    lineNr = parseInt(lineNumber);
    lineNumber = 0;
  end
end
return linesAdded
```

Algorithm 3.3 relates to how Code Review Helper finds the differences in coverage between both versions. This is a simple comparison of both maps, but having to iterate through both means that this operation will not perform as well for larger maps (larger number of classes in the project), as the number
of iterations will be the number of classes multiplied by itself.

Algorithm 3.3: Finding differences in coverage between versions

Result: diff - Differences in coverage

diff = Map<String, coverageResults>;
remove classes that only show up in one of the versions from being compared;
for prEntry in prVersionClasses do
  for stableEntry in stableVersionClasses do
    if the classes match then
      compare entries;
      if entries are different then
        add differences to diff;
    end
  end
return diff

Algorithm 3.4 is how Code Review Helper finds if the pull request changes are covered. First, it calls
algorithm 3.2 the find out what the changes are, and then compares them line by line with the coverage
output file, jacoco.xml. The bigger the pull request, the longer this will take, as each line added by the
pull request requires an operation to find it on the output file and then add it to the result of the algorithm.

Algorithm 3.4: Finding if the pull request changes are covered

Result: result - coverage information on pull request code changes
result = List<ChangedLine>;
get lines added from GitHub diff (algorithm 3.2);
for file in prVersionFolders do
  if file is jacoco.xml then
    for each line added by pull request do
      find it on jacoco.xml and add it to result
    end
  end
end
return result

Finally, algorithm 3.5 finds if areas of code that wasn’t altered had coverage changes. It compares
both versions of all the files that weren’t altered line by line, meaning its performance is largely dependant
Algorithm 3.5: Finding if code that wasn’t altered had coverage changes

Result: result - coverage changes in areas of code that weren’t altered

result = Map(filename, coverageInformation);
coverageChanged = all files that had coverage changes (algorithm 3.3);
remove the files that were altered by the pull request from coverageChanged;

for stableFile in coverageChanged (stable version) do
  for prFile in coverageChanged(prVersion) do
    for line1 in stableFile do
      for line2 in prFile do
        if line numbers match then
          compare coverage;
          if coverage is different then
            add to result;
        end
      end
    end
  end
end

return result

3.6 Motivation Example

To better illustrate the execution of Code Review Helper, we will go through a sample execution, step by step, where we will analyse pull request number 2 from the jpacman-simple [42] repository, which we also use in the Evaluation step of our thesis. The first thing a user will want to do upon downloading the tool is to configure the authentication method. This can be done by simply going to the config.yml file in the core package and inputting their token, which the tool will use from then on.

With authentication set up, a user would install the Service package, and compile the Core package, before running it for the first time. When running the Core package, a user is faced with the option to either download the projects being analysed, or to run the analysis (this is simply so a user doesn’t have to do both in one sitting). Of course, in the first execution the projects haven’t been downloaded yet, so a user will choose the download option. Then Code Review Helper asks the user for the repository being analysed, clones it to two folders (stableVersion and prVersion), and then asks the user what pull request is being analysed, and applies its changes on the prVersion folder. Figure 3.7 shows what this step looks like.

Then, when the user wants to run the analysis, he simply runs the Core package again and chooses that option. After inputting the url of the repository and the pull request being analysed, this step is completely autonomous. The first thing Code Review Helper does is the execution of the tests on both versions, this shows up on the terminal as a standard mvn output. After this first step, the user will notice that the files stableTestTimes.json and timedifferences.json are now present in the Core folder of Code
Then, the user will see a “Processing recording...” message on screen. This means the trace of the stable version’s execution is being parsed so it can later be used to generate the flame graph. Once done, a “Recording processed, writing to file” message shows up, and this process then repeats for the pull request version. This will generate the stableProfile and prprofile files.

Next, Code Review Helper does all the logic relating to coverage. This means the user will see a standard mvn output on screen as the coverage reports are generated. Then the tool runs the analysis, and when this is done, four more files have been generated: stableCoverage.json, pullRequestCoverage.json, coverageDifference.json and diffs.

Finally, Code Review Helper searches for related areas of code, that weren’t altered by the pull request but had coverage changes. The user will see a few Loading repo files... messages on screen, before a list shows up with all the files that had coverage changes despite no code changes (this list can be empty, as it is in the case of this specific pull request we’re analysing). After this step is done, two more files will have been output: relatedAreas.json and deadCode.json.

At this point, the core folder will look something like Figure 3.8.

There’s only one more step before being able to run the views, which is to convert the stableProfile and prprofile files into flame graphs. To do this, we simply use the FlameGraph tool, available on
Figure 3.8: Core folder after analysis step is run
GitHub [43]. This step can be seen on Figure 3.9. After the flame graphs are created, the user simply places them in the same folder as all the other generated files and the analysis is complete. Then, the user goes to the presentation package, compiles it, and runs its. After this step is done, the user can simply open a web browser on localhost:8080 and be presented with the home page of Code Review Helper, as seen on Figure 3.10. This contains some instructions on what each view represents, and the user can then use the navigation bar to go between them. An explanation of each of these views is present in section 3.4, along with screenshots for each of them in appendix A. Using these views, the user can now perform the code review.

### 3.7 Online availability


This concludes the description of our solution. In the following chapter, we describe the evaluation process and its results.
Analysis of pull request

Instructions

Changed Lines Coverage - Displays the lines that were altered in the pull request (same way as GitHub) and whether they are covered by the test suite. Also has direct GitHub links to any other files whose coverage has changed (if any)

Related Coverage - Displays the coverage information in files that were not altered but had coverage changes (could be empty if no changes are introduced by the pull request)

Test Times - Displays the test time differences between both versions. Table entries are color coded for easier recognition. Green means improvement, red means the test was slower, yellow means no significant change.

Flame Graphs - Displays flame graphs representing the execution profile of the tests of both versions

All Coverage Info - Displays a tree containing the coverage of both versions and differences

Figure 3.10: Code Review Helper's homepage
In this chapter we present the evaluation stage of our project. This was done by letting users use the application developed to analyse a pull request and compare the views presented to the ones currently available on GitHub, followed by a form to assess the time taken to review the code as well as the users’ opinions on each of the views presented.

This chapter is split into two sections: the first one is where we’ll describe the methodology we used in detail, and the second one is where we will present the results and their respective analysis.

4.1 Methodology

The evaluation was all done remotely due to the ongoing pandemic. The first step in our evaluation process was for the users to answer a form asking them questions about their experience with code reviews, GitHub, and GitHub’s pull request interface. This allows us to characterize the group that is performing the experience. In this phase, we assess:

- Whether the user has entered the job market, and if so, how long ago they did
- Whether the user has performed a code review before, and if so, how it was done
- The user’s opinions on the importance of code reviews for software development
- How much experience the user has with GitHub
- What the user thinks about the current GitHub pull request views and whether they can be improved, and if so, how.

After this is complete, the user moves on to the experimental phase itself. First, depending on their IST number, they are placed into one of four groups. Each of these groups performs a slightly different version of the experience, by analysing different pull requests, and by some using our tool first, followed by GitHub’s views, while others do it the other way around. More specifically, the way we placed users into the groups was based on the two last digits of the IST number, and whether each was even or odd. Groups 1 and 2 would analyse pull request number 1, while groups 3 and 4 would analyse pull request number 2, and groups 1 and 3 would use our tool first, while groups 2 and 4 would use GitHub’s pull request views first. This was simply to ensure that our data wasn’t biased due to us using a specific pull request that worked particularly well/poorly, or due to the order that the users use the tool/GitHub affecting users’ opinions.

After being placed into a group, the user starts the experiment by following the instructions given to them. For example, if a user is in group 1, these instructions can be summed up to running the tool on the specified pull request, opening the views in a browser (on the address localhost:8080), and using them to review the changes while timing it (we specifically asked users to “try to understand the code
changes, their impact, potential problems with the implementation, and whatever else [they] find relevant in the code presented"). We also included a short description of how the tool works, although most of it is detailed when opening the tool itself, so we didn’t go in depth in our guide.

Then, the user does the same using GitHub, followed by answering our final form.

The pull requests being analysed were from a simple repository we built for the purpose [42], which was a very simple pacman game, heavily based off of the jpacman repository [44]. They involved adding a score (pull request number 1), as well as ghost NPC logic (pull request number 2).

In this final form, we assess a few things about the experiment:

• How much time it took for the user to review the pull request
• Whether the extra information presented by our views helped the code review
• Whether the views in general were easy to understand
• Whether the tool was easy to use
• Whether each of the views was relevant and easy to understand
• How much of an improvement (if any) our views were when compared to GitHub’s views
• An open ended question where the user could write whatever feedback the other questions didn’t capture

After filling out this form, the user has concluded the experiment. In the next section, we will present and analyse the results gathered.

4.2 Results

In this section, we present the results of our experiment. This is split into two subsections: first, we present the results of the group characterization, and secondly, we present the results of the experiment itself and how our tool fared.

4.2.1 Group Characterization

Five people completed both steps of our evaluation process. In the first step, where we simply characterize the group performing this experiment, we found that all five people have a Bachelor’s degree in an engineering field, with two having already entered the job market and having been there for more than two years. All participants said yes when asked if they were familiarized with the code review concept, but only two had ever performed a code review. We also asked users about their opinion on the importance of code reviews for software quality, and all found it to be important or very important.
Two of our users also had experience performing code reviews, with one doing it through a code inspection in a team meeting, and the other doing it remotely. Both users who had performed code reviews said they didn’t think GitHub’s patch interface could be improved. All five users had hosted projects on GitHub before, with one having hosted two projects there, and the other four having hosted three or more.

4.2.2 Results regarding our tool

After performing a code review with both our tool and GitHub’s interface, all 5 users said that the pull request changes and their impact was made easier to understand by our tool, with 4 of them saying that the extra information presented by our tool helped the code review. Users also found the tool relatively easy to use, as seen in figure 4.1. We also asked whether the views were an improvement, and most found it a moderate to good improvement, figure 4.2. Originally we were also going to assess whether our tool helped the speed of code reviews, by telling some users to use our tool before GitHub and other users to do it in the reverse order, but with only 5 users in total we were unfortunately able to gather enough data for representative results.

Next, we asked users for their opinion on the information we presented regarding code coverage. Four of the five users found it to be relevant and said it made it easier to understand the code changes with one user saying no in both instances. All users agreed that the information presented was relatively easy to understand, as presented in figure 4.3.

The next section of our form regarded our views presenting the test times. Here, four of the five users found it to make the code changes easier to understand, but only 3 of the 5 users found them to
**Figure 4.2:** Users opinions regarding whether the tool is an improvement when compared to GitHub

**Figure 4.3:** Users opinions regarding how easy the code coverage views were to understand
be relevant in the context of a code review. In terms of ease of use and understanding, responses were very split with two users found it to be very easy, but the other three being split across the scale, figure 4.4

Regarding the flame graphs, 3 of the 5 users found them to be relevant in the context of a code review, but only 2 of the 5 said our presentation helped make the code review process easier. Regarding ease of use and understanding, responses were once again split, with one user finding it very easy, one finding it not easy at all, and three classifying it in the in-between, as seen in figure 4.5

Finally, we asked the users for any extra feedback or opinions that weren’t caught by the previous questions. Three users left this blank, one user talked about the homepage, saying “Put a little more styling in the homepage, between each paragraph and put bold in each tab menu listing explanation, for easy reading.”, and one user said they would have liked “More information regarding what each of the graphs, especially test times which I didn’t understand what time was it testing. Most of my test times are positive (meaning slower) which means the resulting pull request is slower.”

4.3 Discussion

The results mostly show that our analysis and views can be helpful with regards to code reviews. While the remote nature of the experiment along with the small number of participants limited what we could accomplish with the experiment, the feedback we got from the participating group was mostly positive. In this section, we discuss the insights we gathered, as well as the limitations of our experiment and how they affect the results.
4.3.1 Insights Gathered

The results mostly show that our analysis and views can be helpful with regards to code reviews. While the remote nature of the experiment along with the small number of participants limited what we could accomplish with the experiment, the feedback we got from the participating group was mostly positive.

All 5 of our users said that the information presented by our tool made the pull request changes and their impact easy to understand, with 4 of them agreeing that this helped the code review. This was one the biggest questions our work was trying to answer, and given the results we can conclude that yes, the insights provided by a semantic analysis on a pull request can be useful for improving the code review process.

Regarding the utility of each of the views, the ones regarding code coverage seem to be the most useful, as 4 of the 5 users agreed that they were relevant and made it easier to understand the code changes, which is a more positive result than the views containing test times or flame graphs. We theorize that this is because our coverage views already have similarities to the GitHub views, with extra information appended to it. This makes them easy to use for someone unfamiliar with our tool, and it means that our information is not a replacement but simply an extension of what already exists, meaning that unless the new information got overwhelming, they would be unlikely to have a negative impact on the code reviews.

Still, there is some room for improvement here, especially in ease of understanding. The average score here is 3.4, from a scale of 1 to 5, where 5 is very easy and 1 not easy at all, which suggests there are still improvements to find. Although no one suggested any specific improvements, we think that the coverage on code changes could potentially be improved by adding the coverage information on
the same table as the code changes themselves, rather than two different tables which cause the user to go back and forth.

Second in terms of usefulness, are the tables containing the test times. 4 of the 5 users found them to make the code changes easier to understand, but only 3 said this was helpful in their code review process. Unfortunately this result is a bit limited, as the users were reviewing a pull request from a repository that they had just become familiar with, meaning they weren’t totally certain of what each of the tests did and how meaningful the time changes were. Still, it seems clear that the idea of presenting test times themselves can be useful, and we suggest that the results would show this even more so if users were to test the tool outside of a controlled environment and in one of their GitHub hosted projects.

In terms of ease of understanding, the test time tables scored an average of 3.8 out of 5, which makes them the easiest to understand out of all the views presented. This was not a surprising result, as they’re simply comparing numerical values and are even color coded to make it easier to identify positive or negative changes. Still, one user found them difficult to understand, as he didn’t understand what was being tested (which we already talked about is likely due to an unfamiliar repository), as well as confusion regarding positive test times. These are mostly easy problems to fix and would simply require a help dialog on the page. Overall though, test times were still an easy to understand addition that had a positive impact on most users’ code reviews.

We also evaluated our flame graph views. These were the least useful in the context of a code review, with only 2 out of the 5 users considering that they helped in making the code review process easier, but 3 out of the 5 users thought they were relevant in the process of a code review. This indicates that while a display of execution traces can be useful, the one we used wasn’t the most appropriate. This could potentially be fixed by using a different approach that makes it more friendly to these types of analysis, such as Flamegraphdiff [16], for example, as it overlays two different traces and can maybe make the differences pop out more immediately for the user.

In terms of ease of understanding, the flame graph views scored an average of 3 out of 5, which makes them the hardest to understand out of all the views presented, with one user even classifying it with a 1. This again supports the idea that a different approach when presenting execution traces could prove to be more fruitful than the one we used.

Lastly, and due to one user mentioning that the homepage could have more styling for easier reading, we also believe that there’s a possibility that an improvement here could prove to be an improvement for the entire tool, as the homepage features information and instructions regarding all the other views.

4.3.2 Limitations

Due to the difficulty we had recruiting participants for our remote evaluation, there are some limitations in our data that mean some aspects weren’t evaluated. One of these aspects is how the time taken
to review a code change with our tool compares to the time taken when using GitHub’s views. This is because the methodology for this would involve a group of participants using our tool first, with others using GitHub’s views first. With an already small group of 5, analysing smaller splits of that group would be unrepresentative.

This also applies to other areas where we could have gathered some insights, for instance regarding whether certain characteristics of a user make them more or less likely to find our tool useful. An example of this would be comparing the data from people who have already entered the job market to people who haven’t and seeing if there’s a difference in the results and whether each group found our tool more or less useful. Unfortunately, each group of people with specific characteristics would be too small to draw proper conclusions.

Another possible limitation is the fact that we tested our tool with only two pull requests. This was done to limit the variables in the experiment and ensure that it was kept at a reasonable length for people to perform. We also intentionally used a repository that’s often used for educational purposes and the pull requests were real features from that repository to ensure that they’re representative of a real life situation and that we didn’t introduce a bias in the results. Still, it is possible that with different pull requests, the results would differ, and we would require a much more extensive evaluation process with many more people and many more pull requests, which was unfortunately unrealistic for us.

Still, despite these limitations, we believe the insights gathered from our evaluation are valid, as discussed in section 4.3.1, but in this section we wanted to mention possible limitations and threats to the validity of our results.
5
Conclusions

Contents

5.1 Conclusions ................................................................. 53
5.2 Future Work ............................................................... 53
5.1 Conclusions

This thesis had the goal of analysing the impact of code changes on a project and showing the insights gathered to developers who are in the process of a code review, in an attempt to make the code review process as efficient as possible.

This goal was met, as we developed a working solution that automated most aspects of running the two different versions of a project, gathered a series of insights, and presented them to developers. To do this, we settled on a system architecture that consisted of three packages (Service, Core and Presentation), each with its own specific role in the analysis, which proved to be a good way to structure our tool. In separating the logic of our tool into these three packages with each having its own defined responsibilities, with Service doing all the communication with GitHub, Core doing all the analysis of the pull requests, and Presentation doing all the work related to the views, we were able to perform all the tasks we had decided upon in a fairly straightforward way. Further, normalizing the outputs of the Core package and having the ability to store them for later uses proved to be a plus, as it meant that we didn’t need to perform the code reviews immediately after the analysis were done, as all the files were stored locally and could be visited at any point.

Then, we evaluated how our solution performs, by having developers use it to perform a code review and compare it to the standard GitHub views. Results were mostly positive, with most users reporting that our views helped in the understanding of the code changes that had been made. Despite this, there are still areas to improve on, both in terms of usability and what information is presented, as some views we created proved to be more useful than others. We found that a simple counter showing coverage of the lines that were altered can have big potential benefits, as well as a simple table displaying test times (with this becoming even more useful if the developer is well familiar with what each test does), while the display of execution traces showed some promise, but was found to be harder to interpret than other views, and less useful on average. We theorize that it is possible a different way to show them could alleviate these issues.

In light of how important code reviews are in software development nowadays, we hope our contribution has showed some possible ways to improve them and potentially make them a more efficient and accurate process.

5.2 Future Work

We identify some further work that could be done in this area to expand on our contribution.

First, we consider that such an analysis can be expanded by implementing some sort of test generation, allowing our tool to work even if the project being analysed does not have tests, increasing the number of projects our tool can work with. In addition to this, applying similar concepts to other
programming languages could be of interest, as our system only works for Java, which also limits its usefulness.

 Further, we consider that performing a larger scale evaluation would be valuable, with more participants and more aspects being evaluated, such as time taken to perform the reviews when compared to existing solutions. A larger number of participants would increase the viability of the results, and the evaluation of more aspects would expand our knowledge in regards to how this type of analysis can help code reviews.

 It could also be interesting to expand on the ways to present the information we gathered, to alleviate some of the ease of use issues identified by the users. We suggest some of the potential solutions in the Evaluation chapter, and exploring them, along with other new ways to present similar information, could be of interest, as it could improve the results found with our tool.

 Finally, we consider that a different way of evaluating Code Review Helper could be helpful in terms of better understanding how well it functions. This could be done by analyzing real pull requests present in real GitHub repositories (which would be trivial, as Code Review Helper automates this process), and assessing whether the views introduced contribute to a better understanding of the pull request, for example, whether our coverage views would be quicker in identifying potential issues with the pull requests than the standard github views.
Bibliography


Presentation Package Screenshots
Figure A.1: Main view (Section 3.4.3)
If there is no table being displayed, it means that the pull request did not affect the coverage of the rest of the code. If there is, the left side shows the specific lines that changed, and the right shows how it changed.
**Figure A.3:** General Coverage (Section 3.4.5)
### Test Times on Stable Version:

<table>
<thead>
<tr>
<th>Feature</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>splitOutOfBounds</td>
<td>0.002</td>
</tr>
<tr>
<td>splitWidth</td>
<td>0.015</td>
</tr>
<tr>
<td>noStartSquare</td>
<td>0</td>
</tr>
<tr>
<td>registerThirdPlayer</td>
<td>0.008</td>
</tr>
<tr>
<td>verifyHeight</td>
<td>0</td>
</tr>
<tr>
<td>connectedEast</td>
<td>0.799</td>
</tr>
<tr>
<td>spriteHeight</td>
<td>0.015</td>
</tr>
<tr>
<td>startStop</td>
<td>0.183</td>
</tr>
<tr>
<td>testJava</td>
<td>0.01</td>
</tr>
<tr>
<td>registerSecondPlayer</td>
<td>0.05</td>
</tr>
<tr>
<td>registerPlayerTwice</td>
<td>0</td>
</tr>
<tr>
<td>registerPlayer</td>
<td>0</td>
</tr>
<tr>
<td>testOrder</td>
<td>0.008</td>
</tr>
<tr>
<td>spriteWidth</td>
<td>0</td>
</tr>
<tr>
<td>connectedNorth</td>
<td>0</td>
</tr>
<tr>
<td>connectedSouth</td>
<td>0</td>
</tr>
<tr>
<td>verifyXY2</td>
<td>0</td>
</tr>
<tr>
<td>verifyXY0</td>
<td>0</td>
</tr>
<tr>
<td>verifyWidth</td>
<td>0</td>
</tr>
<tr>
<td>verifyXY1</td>
<td>0</td>
</tr>
<tr>
<td>connectedWest</td>
<td>0</td>
</tr>
</tbody>
</table>

### Differences in the Pull Request Version (negative values are improved):

<table>
<thead>
<tr>
<th>Feature</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>splitOutOfBounds</td>
<td>0.015</td>
</tr>
<tr>
<td>splitWidth</td>
<td>-0.01</td>
</tr>
<tr>
<td>noStartSquare</td>
<td>0.008</td>
</tr>
<tr>
<td>registerThirdPlayer</td>
<td>-0.002</td>
</tr>
<tr>
<td>verifyHeight</td>
<td>0</td>
</tr>
<tr>
<td>connectedEast</td>
<td>-0.32</td>
</tr>
<tr>
<td>spriteHeight</td>
<td>0.001</td>
</tr>
<tr>
<td>startStop</td>
<td>-0.08</td>
</tr>
<tr>
<td>testJava</td>
<td>0.002</td>
</tr>
<tr>
<td>registerSecondPlayer</td>
<td>-0.00</td>
</tr>
<tr>
<td>registerPlayerTwice</td>
<td>0</td>
</tr>
<tr>
<td>registerPlayer</td>
<td>0</td>
</tr>
<tr>
<td>testOrder</td>
<td>0</td>
</tr>
<tr>
<td>spriteWidth</td>
<td>0.016</td>
</tr>
<tr>
<td>connectedNorth</td>
<td>0.008</td>
</tr>
<tr>
<td>connectedSouth</td>
<td>0</td>
</tr>
<tr>
<td>verifyXY2</td>
<td>0</td>
</tr>
<tr>
<td>verifyXY0</td>
<td>0</td>
</tr>
<tr>
<td>verifyWidth</td>
<td>0</td>
</tr>
<tr>
<td>verifyXY1</td>
<td>0.011</td>
</tr>
<tr>
<td>connectedWest</td>
<td>0.002</td>
</tr>
</tbody>
</table>

**Figure A.4:** Test Times (Section 3.4.6)
Figure A.5: Flame Graphs (Section 3.4.7)