Analysis of the behavioral impact of code modifications

Jorge Veiga
jorge.s.veiga@tecnico.ulisboa.pt

Instituto Superior Técnico, Lisboa, Portugal
September 2020

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

1. Introduction

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. This means that a code reviewer plays a very important role, as he’s responsible for rejecting contributions that would hurt the project, while accepting contributions that would be a positive addition to it. This concept can be applied in many different ways in the real world, one of them being GitHub, in the form of pull requests.

In a GitHub pull request, a developer adds his own contributions to a project in his own branch, independent of the one where the working version of the project itself is located. Then, when those contributions are complete, the developer can ask for them to be added to the project’s code base. In order to guarantee the quality of these contributions, these changes must be reviewed. The code reviewer can choose to accept the contributions an merge them onto the project, reject them, or request for the developer to make some changes before reviewing them again.

As such, it’s clear that code reviews are a fundamental part of software development, as accepting and shaping positive contributions while rejecting negative ones is hugely beneficial.

Traditionally, this code review process is based on the textual change of the code and the status of the execution of the tests. On GitHub specifically, a standard diff between the base version of the project and the one with the pull request changes is displayed, along with some other views, one of which displays all the commits in the pull request individually, for example.

In this project, we intend to explore a potential way to improve this code review process, by assessing the impact of the code changes in a pull request and presenting them to the developer. The goal is for these insights to be presented alongside the traditional diff in order to give the code reviewer extra information that can help them to more efficiently and quickly analyze a code review and decide whether a pull request should be accepted or not.

These insights displayed will mostly be based on coverage, test times, and execution traces. In this document, we will explain the related work, the solution itself and all different packages that it’s com-
posed of, as well as the results of our evaluation which aimed to assess whether the solution we developed has a positive impact on the code review process or not.

2. Related Work
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. [12] surveyed 88 Mozilla core developers and 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.

In an attempt to find what constitutes a good code review, Bosu et al. [4] 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. Further expanding on the issue of large changes affecting code reviews, di Biase et al. [6] 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.

In regards to pull requests on GitHub, Rahman and Roy [13], 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. Gousios et al. [7] 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.

In the realm of visualizing programs’ executions, Gregg describes flame graphs [8], 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.

The visualization of large traces can often be problematic. In order to address that problem Trümper et al. [14] 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.

For situations where concurrency is involved, Karran et al. [10] 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.

When trying to find the differences between two executions, Bezemer et al. [3] present differential flame graphs, which allow the depiction of the differences between two flame graphs [8] 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.

Danglot et al. [5] 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 [9] present a tool called WeCode, which detects conflicts on behalf of the developers, introducing the concept of continuous merging inside an IDE, essentially allowing developers to resolve conflicts while the changes are still fresh in their minds.

3. Code Review Helper
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 Code Review
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 developed. 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.

In order to accomplish this, Code Review Helper three main functionalities: one to connect to GitHub and get the information of a pull request, one to analyze the behavioral impact caused by the changes made by that particular pull request, and finally, one to present the insights gathered to the reviewers.

As such, we split our solution 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 call them Service, Core and Presentation, respectively.

3.1. 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.

This package consists of two classes, serviceApp and changedLines. The serviceApp class serves as the main class in this package and contains most of the functionality, while the changedLines class is an auxiliary class that stores the changes of each file in the pull request.

When running this class, its first step is to authenticate (this is simply because authenticated users have lower rate restrictions when using the Git API, but also allows us to analyse private repositories). Then we can start the analysis itself.

This begins by downloading the project in question to our machine. This is necessary because we need to run its tests. We download both the stable version of the project, as well as one containing the changes introduced by the pull request. 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 run our analysis on.

The service package is also responsible for filtering which lines were altered by a pull request. This is 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 parses the changes made by a pull request, finds which lines were added, and saves them in a structure where the line numbers are associated with the content that was added. All this work is performed in the serviceApp class by the filterChangedFilePatchAdditions method.

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 present in it. 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.

Our system also identifies which files were changed by the pull request. 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.

3.2. 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 work of the core package rests on the assumption that the project being analyzed already contains a working and thorough test suite, as the insights generated by our tool come from the execution of those tests. Our tool does in some way provide insights regarding this, 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 [15], including in GitHub projects [11], 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.

The first step in the core package execution is to run the maven tests on both versions of the project. 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. When the tests finish running, the tool will have all the information required for its analysis. This analysis can be broken down into a few different topics: test times, flame graphs, and coverage.

3.2.1 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 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's 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 core package returns 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.

3.2.2 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 them.

The coverage metrics gathered are the ones from the JaCoCo maven plugin: instructions, branches, cyclomatic complexity, lines, methods, and classes. The output from this section of the analysis includes all the coverage information for the project, gathered by parsing its respective XML files.

The more detailed analysis starts after, as our tool gathers all the lines that were changed by the pull request being analysed and stores coverage information specifically pertaining to those lines. The parseChanges method is responsible for going through the generated XML files, and 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. Each specific line is saved to a json file with the following information for each line:

- **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 information will later be used by the presentation package. This analysis also gathers all the coverage changes in areas of code that were not altered by the pull request, along with the GitHub links to those files. This allows us to assess whether the pull request being analysed is affecting other areas of the code, in a potentially unintended way.

3.2.3 Flame Graphs

Lastly in the Core package section, we have the Flame Graph implementation. First, the tool gathers the required information from the execution of
the projects in order to build a flame graph. To do this we simply configured our project in a way that allows for the production of a Java Flight Recording 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 for the purpose.

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.

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 two files we generate are compatible with the external flame graph tool we use to generate the flame graphs. 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.

3.3. Presentation Package

The presentation package is the package responsible for displaying the information gathered by the Core package to the code reviewer. In order to do this, we created a an application using Spring Boot and Thymeleaf, with some of the information being displayed using the D3.js library.

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’s also where the logic regarding the D3 library is present.

The application we developed can be summed up as a series of different views, each providing a different insight on the pull request being analysed, which we describe next:

3.3.1 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.

3.3.2 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 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.

3.3.3 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’s a much less in depth view.

3.3.4 Test Times

Regarding test times, we take this 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. Flame Graphs

Finally, we display both of the created flame graphs 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. Online availability

The code for Code Review Helper is available on github at https://github.com/JV-23/code-review-helper

4. Evaluation

Our evaluation consisted of a user study to assess the effectiveness of our tool, as described in the following subsections.

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 allowed us to characterize the group that is performing the experience.

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. 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, which was a very simple pacman game, heavily based off of the jpacman repository. 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

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.

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, with an average score of 3.4 out of 5 (where 1 was not easy, and 5 was very easy). We also asked whether the views were an improvement, and most found it a moderate to good improvement. 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.

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 be relevant in the context of a code review. In terms of ease of use and understanding, responses were split.

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.

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.

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.

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.

5. 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’s 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 shown some possible ways to improve them and potentially make them a more efficient and accurate process.

References


