An Approach to Aid Developers Understand Code Change

David José Calçada Lourenço
david.j.c.lourenco@ist.utl.pt

Instituto Superior Técnico, Lisboa, Portugal

October 2020

Abstract

Code review is a common software engineering practice of practical importance to reduce software defects. Review today is often done with the help of specialized tools, such as Gerrit. However, even using a tool-supported code review, it still involves a significant amount of human effort to understand the code change, because the information required to inspect code changes may distribute across multiple files that reviewers are not familiar with. Code changes are often organized as commits for review. However, in commits, files are sorted alphabetically creating situations where larger files or files with more complex changes may be displayed at the bottom of the commit, making their review or understanding harder, as the reviewer may have already spent a lot of time and concentration reviewing less impacting or relevant files. In this thesis we present a new sorting solution based on the type of changes performed in each file. Along the new sorting we provide a simple HTML view based on GitHub with the goal of making code review and understanding easier for reviewers and less time consuming.

Keywords: code change, code review, pull request, commit, github

1. Introduction

Code review is a practice used in Software Engineering in which human reviewers perform manual assessment of source code made by other developers. Such practice was adopted with the intention of identifying and fixing defects and quality problems during the development of a system so they don’t make it to a live release, which can be problematic [11].

Even though code review can be effective on its purpose, it also can be quite expensive in terms of time and effort required to perform it. In the past, Fagan's introduced us to his variant of code inspection [14] which was effective in the improvement of the quality of the developed software, however its formal requirements and cost were a big downside to modern development teams, so they don’t usually adopt it anymore. A more lightweight variant was then adopted by such teams which is usually called modern code review [8]. This variant provides informal requirements and the help of code reviewing tools, meaning that the creation of team meetings for code reviews is no longer needed and that each member of the team may do a review remotely. The feedback of the reviews is usually done through comments.

Besides the adoption of modern code review, many development teams have also adopted modern development methodologies like the distributed software development. In this methodology the team members may not see each other as their work on the project is done remotely. To provide such remote access to the project, such teams may use code hosting sites like GitHub [4].

But GitHub is more than just a site where we can store the code of our project as it provides many features that may greatly improve the development phase of a distributed project. For instance, it lets members of a team clone the project’s repository into their local machines so they can work on their own on code changes and then easily send them back to the online repository. As GitHub has version control mechanisms over each repository, this kind of work methodology is very effective as it gives the development team power to easily keep track of every change and version of the project.

Another important and core feature of the site is the pull request mechanism. Sometimes when a developer changes code that he is not sure is the best for the project, it may be useful to not automatically integrate the changes in the repository, so he may create a pull request.

Pull requests are sets of changes that are present in the developer’s local repository and are
sent to the main repository to be compared and reviewed by other members of the development team. If the changes are agreed upon, they can then be merged to the main repository of the project.

The pull request mechanism is greatly used by Open Source Software (OSS) projects as the repository of such projects is public, meaning that any person is able to clone and send code changes to it, so an "approval" mechanism is needed to ensure the changes are indeed good for the project’s goal.

Given the importance of code review in software development and the increasing usage of code hosting sites like GitHub and their pull request mechanisms, our goal is to provide an alternative or a view that complements the features for code review that already exist in such sites.

Taking a closer look at the information that is displayed in a pull request and knowing that much of that information is used to do code review over the proposed change, we think that providing extra information could improve the quality of a code review as the reviewer would be able to perform the review faster and understand the changes more easily.

Right now the information provided by a GitHub pull request is present in the Conversation, Commits, Checks, and Files Changed tabs as shown in Fig. 1. For the sake of code reviewing, the Files Changed tab contains the most important information for us, as the changes done to the source code are displayed in it. As for the Conversation tab, it’s through it that reviewers give feedback and exchange ideas relatively to the changes being made.

The Files Changed tab displays the files that were changed using a simple diff mechanism, that is, it shows us which lines of code were added or deleted in each changed file as can be seen in Fig. 2. Such way of giving information, even though simple and certainly useful, may sometimes not be good enough for understanding more complex pull requests.

As understanding code changes is indispensable in software development and engineers do ask for more tool support for code reviewing [22] we find that working the information that is provided to us through the diff mechanism and then display more information in a new tab (or view) in the pull request may be a way to improve code reviewing on sites like GitHub.

In this thesis we developed a tool that created a view with a new file listing scheme as well as with elements providing information regarding each changed file. With the final user tests performed, we concluded that new file listing schemes, that take into account the performed changes, and the extra file related information helped the flow of the code review and change understanding by our reviewers.

1.1. Objectives

We wanted to give developers an alternative way of displaying code changes such that it makes their code reviewing tasks easier and less time consuming.

To achieve that, our goal was to create a tool that provides more information other than the one currently available on GitHub’s pull request mechanism. Instead of looking only at the diff content of each file, we want to be able to display the impact that such changes generated throughout the rest of the project. By displaying which changed files generated a bigger impact and which other project files were used by them, we hope to be providing information capable of helping reviewers better understand the code changes being reviewed.

With this in mind we want our tool to be able to complete the following objectives:

- Create a view that displays changes’ impact information in a easy and understandable way.
- Be as accurate as possible in its impact analysis.
- Reduce review time.
- Be considered useful by reviewers.

Even though some of the objectives are subjective because they depend on the opinion of each individual, we will be able to tell if they are fulfilled based on the amount of positive feedback received, during the testing phase of the tool, by our user subjects.
1.2. Contributions
In this thesis our contribution to the code review paradigm was to try to provide an improvement to the way changed files are sorted in a commit page of online repositories sites. Alongside the new sorting, we felt that giving some new small elements to these pages, that are non-existent on such sites, could contribute to a faster code review and easier understanding of the implemented changes. As code reviewers have been pointing out some of the problems they face while code reviewing as well as the lack of features on existing code review tools, we hope that the creation of our tool could be a new step in the imagination and implementation of more and new utility mechanisms to support code review, one of the most used approaches for code quality assessment in today’s software development.

2. Related Work
2.1. Pull based development
Pull-based development is an emerging paradigm for distributed software development. Previous analysis work on GitHub has shown that pull request usage is increasing in absolute numbers among shared repositories [15]. Many of these shared repositories are from OSS projects that receive multiple code contributions from the community via pull requests. But pull requests need to be assessed and approved or rejected. The study shows that the majority of pull requests are processed in less than a day and that code reviews as well as the part of the code that was modified (if it modifies new or recently modified code or not) affect the time needed to merge a pull request into the main branch.

Rahman et al. [20] conducted a study that analysed the factors behind the success, or failure, of pull requests on GitHub[4]. Factors like the repository language, application domain, project age and developers experience were taken into account on the study. From the study they concluded that some languages have a higher failure rate than others, but this may be due to the repository having a high amount of forks. Projects from certain domains have a higher success rate than others based on the pull request activity. As the project gets older, the amount of forks grows and so does the success and failure rates of pull requests. Finally, the amount of pull requests increases with the amount of participating developers, however this also increases the amount of unsuccessful pull requests. Developers with more experience tend to be more productive and have a higher acceptance rate of pull requests.

However, GitHub pull request mechanism can be used for more than receiving contributions.

Gousios et al. [17] analyzed how members of OSS projects core team (integrators) assess contributions through pull requests and how they use them. They found out that, besides being the main mechanism to get contributions from the community, pull requests were also used by integrators to successfully accommodate code reviews and discuss new features for the project. The contribution acceptance is based on its quality and degree to fit the project’s roadmap and some of them are also prioritized, depending on the integrator performing the assessment. Such prioritization is based on the criticality (bug fixes) of the contribution, its urgency (new features) or size.

Yu et al. [25] also studied which factors may influence the latency time to accept a pull request. By analysing a sample of GitHub[4] projects that heavily use the pull request mechanism and continuous integration mechanisms, they concluded that the acceptance latency of a pull request is affected by many and complex factors that are not easily predicted and studied. Some of these factors include the size of a pull request or the delay to the first human response and the availability of the continuous integration pipeline.

Understanding how integrators work and use the pull request mechanism, provides the opportunity to mold the information pull requests provide in a way that facilitates their jobs.

On the other hand we got the contributors, people who contribute to a project through pull requests. Gousios et al. [16] conducted a study to understand the challenges and practices contributors undergo when using the pull request mechanism on GitHub. Before making a contribution, contributors try to understand the current state of the project and assess its needs. Such understanding is often acquired through the exploration of the repository’s issue tracker and, mostly, by examination of current project contributions which are done, precisely, through pull requests. Contributors decide to make contributions for various reasons but the most verified one is the fact that they use the project they are contributing to. However some of them feel that the responsive time from the integrators to their contributions is often big (in some cases the contributions even lack a response) which can make their contributions obsolete or not relevant in the long term, making their effort feel unrewarded. However a justification for this scenario may be, as pointed out by integrators [17], that some contributors also don’t respond to feedback received from the integrators on their pull requests, giving birth to “hit-and-run” pull requests. Such pull requests make the integrator’s job hard on how to handle and prioritize the right pull requests on repositories with a large amount of contributions.
The asynchrony that exists between the production of a contribution, its evaluation, and its integration may be one of the most impactful characteristics of the pull-based development that leads to the challenges both integrators and contributors face when working under such methodology.

2.2. Code review

Code review is not an easy task. Gousios et al. [16] point out that code understanding and reviewing is simplified if code changes pertain to a single, self-contained task. However, such tasks are difficult to create by contributors. Another need reported by both contributors and integrators is that knowing the impact of the proposed pull requests beyond the changed code could also be useful and that having some tool’s results (or some other kind of information) integrated into the pull request interface could help the reviewing process.

Di Biase et al. [13] conducted an experiment to study how some quality factors vary when reviewing code with tangled changes versus untangled changes. For that purpose the experiment assessed code review quality when it was performed over a single pull request, or commit, with many changes and of different types (tangled changes), with 2 smaller pull requests, or commits, each with less changes of only 1 type (untangled changes). Their results didn’t reveal difference in net review time between untangled pull requests and tangled pull requests but the group who reviewed pull requests with untangled changes recognized the benefit of such pull requests as the changset is more divided and better structured without many features. The experiment also revealed that reviewing untangled pull requests lead to more context-seeking steps as users open more related classes to review the changes.

Tao et al. [22] explored how important it is to understand code changes, the information required for developers to understand change’s quality and risks as well as the lack of tool support for such practices. In their study they found out that understanding a code change is indispensable in software development, specially in major development phases, during their code review process. Information regarding the code change’s quality and its risks is important for its understanding, however, such information is difficult to acquire in the current practice of code review.

Baum et al. [10] conducted a study regarding the optimal order of reading source code changes. Their objective was to assess how relevant is the order of changes presented by the review tool to the reviewers and discover better orders, other than the traditional alphabetical one, so reviewers have a more efficient understanding and checking of the code being reviewed. The study revealed that in the majority of the code reviews, the reviewers use the order in which the code changes are presented (usually the alphabetical order) but they find it to be sub-optimal for their task. The participants were either neutral or negative about the traditional alphabetical order. Some of them mentioned that intelligence or analysis behind the way files are listed on GitHub[4] would be a welcomed improvement. Based on these first results, some principles regarding a new file ordering were created by the authors.

With the increasing acceptance and popularity of modern code review processes in software development, assessing the quality of a patch and achieving a well-done code review are two key factors to improve the quality of the code base. Kononenko et al. [19] performed a study to understand how developers conduct a code review, how they perceive the quality of a patch and the key challenges they face when performing code review, one of them being, again, lack of features in code review tools. They discovered that some developers spend most of their time reviewing code than developing code which makes them a group of huge importance in the quality assurance of the project. The challenges they face may vary from technical to personal. Technical challenges are usually associated with review tool support, familiarity with the code as well as understanding its complexity. Personal challenges may be associated with time management, technical skills or even context switching while reviewing a certain patch.

Bosu et al. [12] also acknowledged that modern code review is widely used and that developers may dispend way too much time doing it, so it is necessary to find ways to make code review less time consuming. They conducted a study inside Microsoft with its developers with the objective to get insight regarding what leads to a high quality review. Their results showed that code review comments highly influence the quality of the review, comments pointing out bugs, suggesting improved ways of solving problems or pointing out violations in team practices may help the author of the change submit a higher quality patch. Experience with the code base, and thus understanding the code being changed, was also an important factor to increase the amount of useful comments in the review. They also mention that the review effectiveness decreases with the number of changed files, which is normal, as the understanding and reasoning for more complex, and bigger, patches requires more effort and time from the reviewer.
2.2.1 Code review tools

Modern code review has been adopting the use of review tools to help reviewers perform their jobs. Even though they seem to be lacking useful features [19, 22], they are not being completely ignored and reviewers still use some of them because they do improve review quality when compared to reviews performed without them.

When mentioning code review tools, one that instantly comes to mind by many developers is Gerrit[1]. Gerrit is a web-based code review tool that integrates with Git[2]. By having this integration, Gerrit is able to keep track of every change present on the repository and through a frontend interface it creates webpages (Fig.3 and Fig.4) in which developers of the team may review commits performed by their teammates as well as approving or rejecting them through frontend actions provided by Gerrit. Making a parallelism to better understand Gerrit, we may say that GitHub’s Pull Request mechanism is a simpler Gerrit that is already integrated in git repository sites.

Barnett et al. [9] developed a static analysis tool at Microsoft, ClusterChanges, for decomposing changesets into smaller and independent ones. They believed that understanding a code review is more difficult when the changeset is big and composed of multiple and independent code changes. In order to help reviewers understand code changes in a easier way, ClusterChanges uncovered relationships such as definitions, and their uses, and method calls present in a diff-region of C# files to perform the decomposition of a big changeset. By decomposing these changesets into two partitions, trivial and non-trivial (as shown in Fig.5), they thought it would make the understanding of the changeset a lot easier for reviewers. Indeed, their study’s results showed that changesets composed of unrelated changes may affect negatively change understanding as reviewers may need to switch context or separate unrelated changes multiple times to better review them. These claims are backed up by the participants that tested the tool as they found the decompositions correct, complete and useful for their reviews.

ClusterChanges was the closest work we could find to our objective and with the success and good feedback it received, we believe that our approach may be a step in the right direction to make a difference in code reviewing.

2.2.2 Code smells

When performing a code review, the reviewer may search for many factors in the code. The main one is, of course, the functionality of the code, that is, understanding what that portion of the code does or if it is correct, bug or syntax error free. But, besides this technical analysis, a reviewer may also assess the quality of the code, and is in this context that code smells enter the scene.

Code smells are not code errors or syntax violations, they are characteristics of the source code that may suggest issues with code quality, such as understandability and changeability, which can lead to the introduction of faults[24]. Since code smells have a subjective characteristic, as their definition depend on the source code quality standards or requisites defined by each developer, there are few studies about them yet.

Yamashita et al. [24] investigated the extent to which aspects of maintainability, that were identified as important by programmers, are reflected by code smell definitions. Their results came from a set of interviews performed with professional developers over a defined time period. During this period, developers were asked to perform maintenance actions and some changes over their assigned systems and then perform the interviews for data collection. Their results showed that some
code smells can provide insight on different maintainability factors which can be improved via refactoring.

Much like the previous work, Sjøberg et al. [21] also aimed to determine the relationship between code smells and maintenance effort by performing a similar approach of conducting interviews with professionals that are maintaining some system over a defined period of time. However their results showed that from the 12 code smells identified by them none increased the effort needed to maintain the system, which is a result that contradicts other studies.

In the literature some believe that code smells may hinder object-oriented software evolution. Khomh et al. [18] investigated if classes with code smells are more change prone than classes without them. They divide their research into three ‘phases’ to try to understand the impact of code smells. To draw some conclusions, they study change likelihood by comparing classes with code smells against classes without them, classes with a different amount of code smells and finally the relation between some code smells and change proneness. Their study was performed over some releases of two different systems (Azureus and Eclipse) in which they detect existing code smells on their classes and study the relationship between them and change proneness of the classes. The results have shown that code smells increase the number of changes a class undergoes and the more smells the class has, the more change prone it is. However the identified smells didn’t all have the same impact, meaning that certain smells lead to more changes than others. With such conclusions we should be more aware of code smells and avoid certain bad practices as they may indeed have a negative impact on the software quality and its evolution.

van Emden et al. [23] also conducted a study on how the presence of code smells may affect the quality of the code. For that, they created a tool to perform automatic code inspection and detection of certain types of code smells. Their work shows that code smells can be broken into smaller aspects that can be automatically detected by tools, thus providing a fast amount of information about code quality, quality that that may be improved if coding standards and good practices are followed to avoid the introduction of code smells.

3. Methodology

To tackle the identified problem and create a new way of displaying code patches to help code review, we needed an online repository that provides the pull-based mechanism for software development as well as a good Application Programming Interface (API) from where we can extract all the needed information for the creation of our tool. Keeping this in mind, we opted for GitHub, as it is one of the most popular/used repositories and meets all of our requirements.

GitHub hosts projects from a variety of different programming languages. Although many of these languages are much used worldwide and well known, each of them has its own syntax which poses a problem for us, as it’s not trivial to create an impact analysis tool capable of analyzing a huge variety of languages. As we want to create and prove that a new view capable of displaying the impact of the patch across the project helps developers in their code reviews, our solution will focus on evaluating Java only projects and be named Improved Commit Views.

To perform such evaluation we decided to create a standalone tool. We wanted it to have a simple design and consist only of a few modules that were self explanatory and easy to maneuver.

In order to better understand the functionality of such tool, the following sections will give a quick overview of each of the tool’s created modules one by one. As seen in Fig. 6 the tool is split in three different modules named api, core and display.

3.1. API module

To create a new pull request view we needed the repository’s data to work with and such data can be retrieved using GitHub’s API v3 [3].

GitHub’s API provides a list of requests that can be invoked by developers to retrieve information about all elements present in the site, from users’ information to repositories’ data and so on. This module is responsible to get all the information that is crucial for the implementation of our analysis algorithm. Its job is to interact with the GitHub API asking for that data and store it in defined structures. Such data can then be requested by the core or display module.

The GitHub API homepage specifies that all access to the API has to be done via Hypertext Transfer Protocol Secure (HTTPS) and that both the sent and received data is in JavaScript Object Notation
(JSON) format. To satisfy these requirements we used an already developed Java library that performs all GitHub’s existing API requests. GitHub Java API [5] is a Java library that is kept up to date with the latest GitHub API version and that was easy to integrate in our module.

3.2. Core module

From all the retrieved information, the most important one is the patch of the changed files, since its over that patch that developers will make a review and its from that patch that we can extract information relevant to assess how impactful or hard to understand the changes on a file are.

To assess the level of impact or difficulty to understand the changes of a certain file, we looked for multiple kind of events that may appear in the patch. Such events may come in the form of method calls, new methods, whole new classes, variable declarations and so on...

Fig. 7 shows the different type of events that we looked for and based on each type and the number of occurrences we created and assigned an attribute called impact value to our changed file. The creation of such attribute came from our need of having some information to provide to the module responsible for the view so it can understand how much impact or hard to understand each file is and display them according to our specification.

Each changed file processed by our analysis algorithm received an impact value field that starts at 0. Then, as we went through the steps to find our chosen events on each changed file patch, the impact value was incremented accordingly to the event found, this means that each event will have its own impact value. Bigger impact values means that the file has a bigger impact or is harder to understand.

3.3. Display module

This module is responsible for the startup of our tool and its function is to interact with the user and create a Hypertext Markup Language (HTML) view based on his requests. The module communicates and retrieves data from the API package and with the help of a framework creates dynamic HTML pages able to use and display the Java information that is running through our tool.

To achieve this goal, and taking into account that our tool was developed on an object oriented paradigm and in Java, we needed a way to display the data contained on our objects in an HTML environment.

For that we used two well known web application frameworks, the Spring Framework[6] and the Thymeleaf Framework[7].

3.4. Evaluation

For the evaluation of our tool we wanted to perceive how the created view helped reviewers on their code reviews, that is, if the view elements mainly helped in the understanding of the changes and allowed reviews to be performed faster.

To assess the helpfulness of the created view for a reviewer, we had to perform user tests, as they represent the best way of evaluating the goals of the tool’s produced view.

The user tests were split into two parts, the first one to characterize our subjects and the second one to perform an interaction between one user and our tool.

First part: in this part we recruited and characterized our users. Ideally we would recruit users in person throughout the campus but unfortunately that was not possible, so we had to recruit remotely through emails or social networks.

To each user that agreed to help in our tests, we sent a form with questions about their scholarship degrees, work experience, etc. By doing this we could later search for some correlations between user’s characteristics and the experiment results obtained through the usage of our tool.

Second part: this part of the experiment consisted in a practical interaction between a user and our tool. For each user that participated in the first part of the experiment, by answering our first form,
we then sent an email with the steps for this phase. Since all the user tests had to be performed remotely, and the user needs access to our tool, we sent them a zip folder containing the files needed to run our tool, alongside a read me file that explained how to perform the experiment step by step. After the execution of the experiment, the users were asked to answer another form with questions about their opinions and interactions with the tool.

4. Results & discussion

4.1. Recruited User’s group characterization

We were able to recruit a group of 5 people that performed the full experiment remotely in their machines.

All of the users had a Bachelor degree, were familiarized with the code review concept and had experience using GitHub through the contribution or hosting of projects. 1 out of the 5 users was already in the IT job market with an experience of 2 years.

Even though 2 of the users never performed a code review, all of them agreed that code review is somewhat to very important in software quality as shown in Fig.8. Of the 3 users that already performed code reviews, all of them reviewed through online repositories like GitHub, including GitHub itself as they acknowledged past reviews done through its pull request/commit mechanism. All of the users that had experience reviewing through GitHub felt like their experience was positive using the existing interface, Fig.9, and that GitHub's patch interface doesn’t need improvements.

4.2. Experiment Results

We had 2 users perform a timed review through GitHub's webpage and 3 users through our tool. When asked about the tool’s view itself, all of the users felt that the provided listing order helped the flow of the code review, the new provided elements helped in the understanding of the code review as well as making the code review less time consuming. All users also agreed with the impact value/ordering adopted by the processment mode of the tool.

Comparing our tool against GitHub, all users felt that new orders like the one provided are somewhat to very helpful when compared to the traditional alphabetical order, as seen on Fig.10. Fig.11 shows us how much of an improvement the provided view is when compared to GitHub's traditional view, according to all users.

When asked about improvements to our tool, one of the users mentioned adding language specific keyword coloring to ease readability, "Add more styling to certain keywords referred to the language being reviewed. Example: keyword class in java could have another color." Another user mentioned the implementation of a summary for each changed file regarding the amount of scenar-
ios found that led to the determined impact value, "display why the file has its impact value. for example saying that it has x changes in field declarations and y changes in the declaration of the class constructor."

4.3. Data Analysis
We asked our users to time their reviews, however the data obtained from that task was not enough for us to draw conclusions.

Regarding the order provided by our tool, 4 out of 5 users ranked its helpfulness on a level higher than 3, meaning that, orders created through the content of the changes and not only by the name of the file, may indeed help a reviewer better understand file changes. In our case this perception may be due to the fact that we display files with bigger and harder changes at the top of the commit when the reviewer still has its attention and predisposition levels at their highest values making their change understanding easier, as these levels are key factors during a code review.

Comparing the information provided by our view with the one provided by GitHub's view, our users' perception was that our view presented a decent improvement. As we got no negative feedback regarding our view's elements that provide extra information for each changed file, we may conclude that such improvement perception was directly linked with them and the new listing order, as these are the main features that are not present on GitHub's view.

Having no negative feedback on elements like the Related Files and Code smells boxes proves that having these types of quick access links or technical information may help the flow of a code review for a user when displayed by the file.

All users agreed that the developed tool was easy to use and understand, meaning that our goal of developing a user interaction friendly interface was met.

4.4. Threats to Validity
The major threat to validity in our experiment was the size of the user group. Having such a small amount of users made impossible to have a big enough review time sample to draw conclusions about one of the questions we wanted to address, "Does a different file listing order reduce review time?"

Besides that, a small amount of users may introduce a big amount of noise in the answers if the opinions of a few people diverge from the rest of the group. Fortunately when we assessed the subjective questions regarding each user's opinion and experiment experience, the majority of them had the same perception about the studied subject, but having a bigger sample of users would've been a lot better to draw further and more precise conclusions.

5. Conclusions
In this project we developed a tool that created a different type of file ordering and changed files view for repository hosting sites that use the pull request mechanism. We wanted to study if showing more information regarding each changed file alongside a different kind of file ordering, other than the traditional alphabetical order, helped code reviewers in their task of reviewing the changes implemented by a certain pull request, by facilitating change understanding and reducing review time.

We weren't able to conclude that our approach reduced the time spent reviewing a pull request but we were able to confirm that different types of file ordering, the extra information and quick access links provided, regarding each changed file, helped our users in the flow of their reviewing task when compared to the traditional GitHub view.

In sum we conclude that, as many code reviewers perform their reviews through the hosting site interface, adopting new ways of file ordering, that takes into account the changes performed on a file and the repository as a whole, as well as the provision of extra information regarding such files, is a step in the right direction to improve a hosting site's interface and the code review task performed by our reviewers in today's software quality assessment.

5.1. Future Work
The implemented tool focused on working and analysing Java only projects hosted on GitHub. Extensibility to the pool of analysed languages could be done by implementing language specific parsers and processment for the desired nodes produced.

Regarding the Java implementation itself, the static analysis processment performed with the help of the used libraries could be made more accurate if the pull request being reviewed is firstly downloaded to the local machine, as the processment done by the javaparser tool is more accurate if it has access to the source code of the whole project. The information displayed in the produced view, as well as its aspect, could be improved according to the feedback received by our users.

References


