An Approach to Aid Developers Understand Code Change

David José Calçada Lourenço

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. Nuno João Neves Mamede
Supervisor: Prof. Rui Filipe Lima Maranhão de Abreu
Member of the Committee: Prof. António Manuel Ferreira Rito da Silva

October 2020
Acknowledgments

First of all a huge thank you to my father who provided me all the conditions and support to be able to fulfill my life goal of having a superior academic degree, and to my aunt who gave me a place to live during this long journey.

A huge thank you to all my family and friends who gave me the emotional support and motivation to keep going, without them this journey could’ve been a lot harder.

Thank you to all my Instituto Superior Técnico (IST) friends and colleagues who took the same path as I did. For all the late nights and challenges we faced together, thank you for helping me grow, learn and overcome every academic problem that crossed our way.

Special thanks to Professor Rui Maranhão for being my advisor. Thank you for all your knowledge and guidance, for dissipating any doubts that came along the way during this project, without you this thesis wouldn’t have taken shape, thank you!

And last but not least, a thank you to all IST professors that crossed paths with me during this academic journey. All of you contributed somehow to make this final achievement possible!
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
Resumo

Revisão de código é uma prática comum em engenharia de software de importância prática para a redução de defeitos no software. Hoje em dia as revisões são feitas com a ajuda de ferramentas especializadas como o Gerrit. No entanto, mesmo numa revisão de código auxiliada por este tipo de ferramentas, ainda é necessário uma quantidade significativa de esforço humano para perceber as alterações efetuadas no código, uma vez que as informações necessárias para analisar as alterações efetuadas podem estar distribuídas por vários ficheiros com os quais o revisor não é familiar. Alterações de código são geralmente organizadas em commits para serem revistas. No entanto, em commits, os ficheiros são ordenados alfabéticamente, criando situações em que ficheiros muito grandes ou com alterações complexas possam ser mostrados apenas no fundo do commit, fazendo com que a sua revisão e compreensão sejam mais difíceis uma vez que o revisor pode já ter dispendido muito tempo e concentração em ficheiros com menos impacto ou relevância. Nesta tese apresentamos uma solução de ordenação baseada no tipo de alterações realizadas em cada ficheiro. Juntamente com a nova ordenação fornecemos uma vista simples em HTML baseada no GitHub com o objetivo de tornar a revisão de código e sua compreensão mais fácil para os revisores e menos dispendiosa em termos de tempo.

Palavras Chave

code change, code review, pull request, commit, github
# Contents

1 Introduction ................................................................. 1
   1.1 Objectives .................................................................. 5
   1.2 Contributions .......................................................... 5
   1.3 Structure of the Document .......................................... 6

2 Related Work ..................................................................... 7
   2.1 Pull based development .............................................. 9
   2.2 Code review ................................................................ 10
      2.2.1 Code review tools ............................................... 12
      2.2.2 Code smells ....................................................... 14

3 Implementation ................................................................ 17
   3.1 Improved Commit Views .............................................. 19
      3.1.1 Plugin vs Standalone Tool .................................... 19
      3.1.2 Improved Commit Views Architecture .................... 20
      3.1.3 Improved Commit Views Repository ....................... 20
   3.2 API Package .............................................................. 20
      A – Repository files’ paths: ........................................... 21
      B – Repository Pull Requests: ...................................... 21
      C – Pull Request commits: .......................................... 22
      D – Commit changed files: .......................................... 22
      E – Changed files patch: ............................................ 22
      F – Changed files content: ......................................... 22
   3.2.1 GitHub Java API .................................................... 22
   3.2.2 Authentication ........................................................ 23
   3.2.3 Technical Information ............................................. 24
   3.2.4 Retrieving repository files’ paths .............................. 26
   3.2.5 Retrieving Pull Requests ........................................ 26
   3.2.6 Retrieving Commits ................................................ 27
# List of Figures

1.1 GitHub's pull request view. ................................................. 4  
1.2 GitHub's *diff* mechanism .................................................. 5  
2.1 Gerrit Web UI change screen .............................................. 12  
2.2 Gerrit Web UI side-by-side diff screen ................................. 13  
2.3 Cluster Changes - Tree view displaying a change ..................... 13  
3.1 Improved Commit Views Architecture ..................................... 21  
3.2 Starting page of our tool, asking for a repository to scan .......... 43  
3.3 Page showing the open state pull requests of a requested repository 44  
3.4 Page showing the commits of a Pull Request ............................ 45  
3.5 Page showing our produced view of a patch ............................ 46  
3.6 Page showing our produced view of a patch with both code smell boxes 47  
3.7 Page showing our produced view of a patch with non Java files present 47  
4.1 User’s code review importance given to software quality .......... 57  
4.2 GitHub’s code review easiness as perceived by users ................ 58  
4.3 User’s perception of helpfulness of orders other than the alphabetical one. 58  
4.4 User’s perception of our view’s improvement when compared to GitHub’s. 59  

ix
List of Algorithms

3.1 High level example of a Improved Commit Views execution ............................................. 48
3.2 API retrieveRepositoryFilesNames recursive function pseudo code .................................. 49
3.3 API getRepositoryPRsInfo function pseudo code .......................................................... 49
3.4 API printRepositoryPRCommits function pseudo code .................................................... 50
3.5 API getRepositoryPRCommit function pseudo code ......................................................... 50
3.6 Core processCommit function pseudo code ................................................................... 51
3.7 Visitor process function pseudo code .............................................................................. 51
3.8 NodeType>Process class sample process function pseudo code ........................................ 52
3.9 SmellChecker process function pseudo code .................................................................... 52
# Acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>OSS</td>
<td>Open Source Software</td>
</tr>
<tr>
<td>IDE</td>
<td>Integrated Development Environment</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>AST</td>
<td>Abstract Syntax Tree</td>
</tr>
<tr>
<td>HTTPS</td>
<td>Hypertext Transfer Protocol Secure</td>
</tr>
<tr>
<td>HTML</td>
<td>Hypertext Markup Language</td>
</tr>
<tr>
<td>JSON</td>
<td>JavaScript Object Notation</td>
</tr>
<tr>
<td>JVM</td>
<td>Java Virtual Machine</td>
</tr>
<tr>
<td>CSS</td>
<td>Cascading Style Sheet</td>
</tr>
<tr>
<td>IST</td>
<td>Instituto Superior Técnico</td>
</tr>
</tbody>
</table>
Contents

1.1 Objectives ............................................................... 5
1.2 Contributions .......................................................... 5
1.3 Structure of the Document ........................................ 6
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 [1].

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 [2] 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 [3]. 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. 1.1. 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. 1.2. 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 [5] 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.

---

**Figure 1.1:** GitHub's pull request view.
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 creation 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.

1.3 Structure of the Document

Chapter 2 provides some background and related work that has been done regarding pull-based development and code review. In Chapter 3 we describe our solution and its implementation, a tool that aims to improve the identified problem. We go through the details on how we implemented it and the choices we made and why. Chapter 4 describes our evaluation of the tool, how we conducted the tests, the data we acquired from them and an inspection and analysis of such data. In Chapter 5 we wrap up the document with the conclusions we could infer from the work done.
2 Related Work

Contents

2.1 Pull based development ............................................. 9
2.2 Code review ......................................................... 10
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 [6]. 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. [7] 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. [8] 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. [9] 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. [10] 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 [8], 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. [10] 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. [11] 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 changeset 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. [5] 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. [12] 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. [13] 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. [14] also acknowledged that modern code review is widely used and that developers may dispense 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 [5, 13], 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 [15]. Gerrit is a web-based code review tool that integrates with Git [16]. 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.2.1 and Fig.2.2) 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. [17] 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.2.3), 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.
Figure 2.2: Gerrit Web UI side-by-side diff screen

Figure 2.3: Cluster Changes - Tree view displaying a change
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 [18]. 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. [18] 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. [19] 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. [20] 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. [21] 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

Implementation

Contents

3.1 Improved Commit Views ............................................. 19
3.2 API Package ......................................................... 20
3.3 Core Package ....................................................... 28
3.4 Display Package ................................................... 40
3.5 Algorithms ......................................................... 48
3.1 Improved Commit Views

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.

3.1.1 Plugin vs Standalone Tool

To fulfill our main goal of creating a different kind of order on the listing of the changed files of a GitHub commit, we had two possible ways of implementation.

The first possible solution was to create a plugin for some Integrated Development Environment (IDE) (IntelliJ or Eclipse) that could make use of already existing GitHub integration mechanisms on such IDEs and that would then display in a window our listing of a requested commit.

The second solution was to simply create a standalone tool that would create a view and display our results through a Hypertext Markup Language (HTML) page, requiring the reviewer to use the browser and not only the IDE where he might be working. We chose this approach.

The solution through the creation of an IntelliJ plugin was very attractive but it was abandoned mainly because of two reasons. The first one was version control. IDEs are always evolving and we may have one or two new versions of them every year. With new versions comes new features and updates to the IDE API and many times plugins that work on version X stop working when version Y comes out, so the created plugin could become useless. The second reason was the amount of GitHub integration that IntelliJ alone already provides. IntelliJ is one of the most complete IDEs out there and since we last checked out, it had its own GitHub integration with already so many features like being able to retrieve the repository pull requests, open them and check commit by commit, file by file. As this feature was quite new, we didn’t find a way to use it through a plugin because there was no API for it yet.

So we followed the solution of creating a standalone tool. This solution may not be as attractive as the solution of an IDE plugin, as reviewers and programmers spend most their time on a IDE, but when it comes to review pull requests it is almost inevitable not to open the browser and check the pull request
there, since most of the time there is a need to comment the changes and create a discussion and this is only doable through the website.

Besides the creation of a different order in the listing of the files, another goal was to be able to create a display, or a view, as close as possible to the one provided by GitHub but that could have some extra information about the files that were changed or some quick access links to relevant files that could be linked to the patch. With this goal in mind we wanted to show that having such new elements (or other ones) along the new order of the files could help and improve code review as well as make it less time consuming through the existence of quick access links. Having a tool that creates such HTML page may be useful to give ideas to the creators and maintainers of sites like GitHub, maybe our elements or views may inspire someone and help improve such sites, and that was something that was always in our mind as we decided to follow this approach.

3.1.2 Improved Commit Views Architecture

As mentioned in the previous section, we chose the approach of creating a standalone tool as a solution for our problem. We wanted Improved Commit Views to have a simple design and consist only of a few modules that were self explanatory and easy to maneuver.

So in order to better describe and understand the functionality of it, we will, in the following sections, go over each of its created modules one by one, explaining how they work, the choices we made on their implementation, known issues, etc. As seen in Fig. 3.1 the tool is split in three different modules named api, core and display.

3.1.3 Improved Commit Views Repository

As we wanted our work to be used by anyone that finds it useful or for future work, we made a functional version of Improved Commit Views available on GitHub [4] for public use, through the following link https://github.com/davidjcl/improved-commit-views

3.2 API Package

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 [22].

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.

Creating a view capable of identifying code changes’ impact across the classes of the repository required that we accessed and used information related to the repository’s pull requests, files and commits. After studying and testing how the API works, we were able to successfully retrieve the following data needed for the next step of our algorithm.

A – Repository files’ paths: this is the full path of a file present in the master branch of the repository. Having each file’s path will be necessary for us to infer some relations between files during our process of each changed file patch. It also gives us insight of the whole structure of the repository. Since our tool is running locally and working with information requested from the online repository, the information about each file’s path will be stored in a temporary structure. This task can be performed by using the Repositories → Content section of the API.

B – Repository Pull Requests: this represents a whole GitHub pull request, with all the information we can retrieve from it. Since we want to create a view over the pull request mechanism, we’ll need the existing pull requests’ content. Through the API we can retrieve pull requests based on their status, open or closed. As the view is aimed at helping code review, it only makes sense that we retrieve the
pull requests that are marked as 'open'. Pull Requests of a repository can be retrieved through the Pull
Requests section of the API.

C – Pull Request commits: this represents the commits present in a certain pull request, which can
be one or more. For each existing pull request we want to be able to access all of its commits and
retrieve them as they are requested by the user. A commit contains the information that will be analysed
by us. Pull Requests section of the API is again used for this task.

D – Commit changed files: after getting the commits of a pull request we can inspect their data and
among that data is the list of changed files. Code reviews are made upon these files, they are the most
important objects for our tool as our analysis will be done over their contents. To get the list of changed
files of a commit we can use the Repositories → Commits section of the API.

E – Changed files patch: each changed file contains a patch field. This patch represents the
changes made to the file, that is, the deleted and added lines of code like they are shown in the GitHub
view. All the previous work done through the API was necessary just to retrieve this field as our analysis
algorithm will work upon these lines of code.

F – Changed files content: this represents the content of a changed file, that is, the entire source
code of a java file with the changes present in the patch. This information is needed because, to perform
a static analysis, it is useful the use of source code parsers. However such parsers only work properly if
the input content is a well formatted file, and that's why we retrieve this information from a changed file,
to help us in the posterior analysis.

3.2.1 GitHub Java API

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. This means that if we want to request the API for the information mentioned in the above
section, we would need to create HTTPS requests and manipulate (parse) all the involved JSON data in
the best suitable way for us.

We could adopt this approach and code all the necessary classes to create the HTTPS requests
and then parse the information received and so on, but doing so is time consuming as there are many
fields and variables coming from the responses and we would need to fully understand the structure of
the response and extract the relevant information, and by doing so, we could get delays in our schedule
because of the many bugs and errors that may occur in our code while creating them. So, with this in mind, we searched for easier alternatives to access and extract the data we need from the API.

From the API homepage we found a section that contains libraries written in multiple programming languages that abstract the HTTPS and JSON work required, all we need to do is make simple calls directly to the library for the information we want. We now need to understand the available method calls and functions of the library so we can invoke the necessary methods to retrieve such information, but this simplifies our work a lot as it is simpler to do than to create HTTPS requests and parse JSON data. Usually these libraries contain good documentation and implementation comments we can check.

Since our tool was written in Java we wanted a library in that language that is kept up to date with the latest version of the GitHub API. One of the available libraries was the GitHub Java API [23]. We downloaded and tested how this library works, and since we were able to easily use it to retrieve the information we needed, we opted to use it in our solution.

3.2.2 Authentication

According to GitHub's developer page, there are limits to the amount of requests that can be performed to their API. The limit varies whether the performed request is authenticated or unauthenticated. For unauthenticated requests there is a limit of 60 requests per hour, whilst the limit for authenticated requests is 5000 per hour.

Since our tool may easily exceed the limit of 60 requests per hour, based on the size of the requested repository, we wanted to make sure that the requests performed through the Java API are authenticated. We can achieve this authentication using a basic authentication, by requesting a username and a password to the user and passing them to the java API library so it creates an authenticated user and performs requests with it, or, in a more secure way, provide the library an OAuth Token that will be used to create the authenticated user.

The simpler way to implement authentication in our tool, from a user's perspective, was to request the user's GitHub username and password alongside the repository's info and perform the API requests from there through the Java library. However, GitHub is deprecating password authentication, meaning that, using this authentication method in our tool would render it useless at some point in time and we don't want that. With that in mind, the only way to perform authenticated requests to the API was through the usage of an OAuth Token.

These tokens are represented through a big string, which makes them hard to be memorized, thus making no sense to be asking users for them all the time to create the authentication. To solve this problem, our tool reads the token from a specific file, display\src\main\resources\application.yml. By default we provide a simple working token in such file. However a user may edit the file and replace the existing token for a token of his own before running the application. If no token is found in the file, the
application will still be able to run but will make unauthenticated requests to GitHub’s API.

3.2.3 Technical Information

This module of our tool is composed from the following relevant classes:

• API - main class responsible for the requests performed to the online GitHub API
• PR - class containing a pull request data
• PRCommit - class containing the data of a commit present in a pull request
• ChangedFile - class containing the data of a changed file present in a commit

**API Class:** this class is the main class of the module. Here relevant structures will be stored and the main functionality implemented. Since our tool will perform API requests to GitHub and such requests are 'repository dependant', each object of this class will represent one GitHub repository in our solution. To create such representation, the class will have several key attributes as shown below:

• String repoToScanOwner - Name of the owner of the repository
• String repoToScanName - Name of the repository
• List<PR>prs - Structure containing the open state pull requests, represented by the PR class.
• Map<String, String>repositoryJavaFiles - Structure containing the repository java files’ paths.

Some other attributes are present, but they are 'connectors' between our class and the used Java API library.

**PR Class:** this class contains the information relative to a single pull request of the repository. Its a simple pull request representation created for our solution, its key attributes are:

• String prTitle - Title of the retrieved pull request
• String state - State of the pull request (should be 'open')
• long id - unique identifier of the retrieved pull request
• int number - number of the retrieved pull request
• List<PRCommit>prCommits - Structure containing the pull request commits, represented by the PRCommit class
**PRCommit Class:** this class contains the information relative to a single pull request commit of the repository. Its a simple commit representation created for our solution, its key attributes are:

- String sha - Identifier of the retrieved commit
- String message - Message of the retrieved commit
- List<ChangedFile>changedFiles - Structure containing the changed files present on the commit, represented by the ChangedFile class
- boolean processed - Boolean showing if the commit was already analysed by our algorithm or not

**ChangedFile Class:** this class contains the information relative to a single changed file of a commit. Its a simple changed file representation created for our solution, its key attributes are:

- String name - Name of the file, which is its path in the repository
- String patch - Changes performed over the file
- String content - Entire source code of the file with the performed changes
- String fileMasterUrl - Repository URL of the master version of the file
- String fileCommitUrl - Repository URL of the commit version of the file
- double impactValue - Assigned value after our analysis of the file changes
- Map<String, String>relatedJavaFiles - Possible repository’s Java files related to the patch of this changed file
- List<String>patchSmells - Patch smells found by CheckStyle [24]
- List<String>patchSmellsPmd - Patch smells found by PMD [25]

The motive behind the creation of such data structures was to minimize the amount of API requests performed through the Java API to GitHub. If a reviewer wants to view one pull request or commit more than once, having such information stored in structures will minimize the amount of time needed to show it to the reviewer, as performing a get over a structure is less time consuming than performing an HTTPS request and all the related processment. Also, as all the information through the ChangedFile objects is stored, and since our analysis algorithm will work mainly with such kind of objects, this will save our tool unnecessary processment as we make sure that each commit (and its changed files) are analysed/processed only once by our core module.
3.2.4 Retrieving repository files’ paths

The retrieval of the repository files’ paths is mandatory for the analysis being done by our analysis algorithm. With this in mind, such paths are the first information being requested through the API class of our package. To get the paths of all of the repository’s files, a recursive method, named `retrieveRepositoryFilesNames`, was implemented to make sure that no file of the online repository is left behind. Such method will make calls to the imported Java API library that will then return information about each repository directory or file. This information is filtered by our method and the paths of the Java files existing in the repository will then be saved in a structure present in our API class.

This information will be used later by the core package, to infer if some classes that may emerge in a changed file patch belong to the project by being searched for in the created structure. Since we opted not to download the repository to our local machine before performing our analysis, this was the best solution we found, as knowing which Java classes are present in the repository is a need to know for our analysis.

3.2.5 Retrieving Pull Requests

After our API class retrieves and stores the paths of the files that exist in the repository, the next logical step in our tool execution is to display and retrieve the pull requests that are marked as ‘open’ in the online repository.

To display the existing pull requests is quite easy, we make a call to the GitHub API asking for all the pull requests that are in the ‘open’ state and then process the returned data. Through a new method named `getRepositoryPRsInfo` we invoke such API call, with the help of the Java API library, and save each returned pull request’s number and title in an entry of a SortedMap that is returned by our method. The returned map will then be used by our display package to display the information to the user.

Now we have the information that identifies each pull request, but we will need to access them and be able to retrieve their commits. We didn’t want our tool to be overloaded and take much time between user requests so we implemented it in a way that it will only retrieve the information that the user requests regarding pull requests and commits. This means that when a user asks for a pull request $X$, through the displayed information by the display package, our API class will only retrieve that pull request’s relevant information to the public API, that is, the existing commits of such pull request but without retrieving their contents (changed files), that is another request that will need to be made by the user. By retrieving each kind of information separately one by one, and not in a ’bulky’ way like automatically retrieving a pull request’s every commit and every file of each commit, we made our tool more responsive and faster between each user’s requests.

We were able to implement such methodology because the Java API library contains methods able
to request the public API for each pull request and each commit individually through their identifiers (pull request number and commit sha).

### 3.2.6 Retrieving Commits

To retrieve the commits of a pull request, we adopted the same methodology as the retrieval of pull requests. After the selection of a given pull request, the next step would be to display its existing commits. To retrieve such information we created a method `printRepositoryPRCommits` that given the pull request number as an argument would retrieve and save each of its commits’ sha and message in an entry of a SortedMap that is returned by the method. The returned map will then be used by our `display` package to display the information to the user.

Now that the information regarding each commit is displayed, comes the last and most important functionality of our API class, retrieving and storing the content of a selected commit. Through the pull request number and the commit sha we implemented a method, `getRepositoryPRCommit`, that retrieves the changed files of the identified commit. Not only this method requests such information to the public API, it will also populate the structures that compose our API class. If the pull request does not exist in the `prs` structure of our running API object, it will create one PR object and populate such structure. After that if the requested commit does not exist in the `PRCommit` structure of the PR, it will create a `PRCommit` object that represents the requested commit and finally populate such object with `ChangedFile` objects representing each changed file retrieved from the requested commit.

In sum, this is the main method of our API class, responsible for returning a pull request commit. If such commit already exists in our API object, it simply returns it. If not, it will ask the public API for it and create or populate all the relevant structures.

### 3.2.7 Known Issues

From the performed tests we observed that this module worked as intended in most cases. However we found 1 specific case that even though the module wouldn’t crash it would propagate one error that would compromise our core module functionality. When retrieving a changed file patch this field could come with a null value and that is a problem for our analysis algorithm. We couldn’t discover why this happened, but we’re certain that it is some limitation present in the Java library we used or even of the public GitHub API. This problem was observed only a couple of times, so we consider it a minor issue.

Another issue that exists in the package is its startup time. When creating an object of our type of API, one of its first steps is to retrieve the repository files’ paths. However, since our algorithm will sweep the whole repository, this step may take some minutes to finish, as it will take longer the bigger the repository is. This is a one time operation when requesting a repository for the first time during the
runtime of our tool, so we see it as a minor issue as it is almost like the configuration load that some tools that exist nowadays have.

### 3.3 Core Package

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...

Table 3.1 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.

### 3.3.1 Technical Information

This module of our tool is composed from the relevant classes:

- **Core** - main class of the module, responsible for inter-module communication and requesting the analysis of a patch
- **Visitor** - class responsible for the startup of a patch analysis
- **NodeType>Process** - class responsible for the analysis of each specific *NodeType*. Package contains *N* classes of this type based on the number of relevant nodes
- **SmellChecker** - class responsible for the detection of code smells during the analysis process

**Core Class:** this class is the main class of the module. It handles data that comes from the *API* package itself, that is, it communicates with such package. Besides that, this class is responsible for
Table 3.1: Table of processed events

<table>
<thead>
<tr>
<th>Level</th>
<th>Case</th>
<th>Example</th>
<th>Impact Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>Declaration/Change of a class or interface</td>
<td>public class RegisterProcess</td>
<td>0.8</td>
</tr>
<tr>
<td></td>
<td>Verification of whole new class</td>
<td>public class RegisterProcess {</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Declaration/change of a class constructor</td>
<td>public RegisterProcess(...)</td>
<td>0.6</td>
</tr>
<tr>
<td></td>
<td>Verification of whole new constructor</td>
<td>public RegisterProcess(...)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Method</td>
<td>Declaration/Change of method declaration</td>
<td>public void hello()</td>
<td>0.6</td>
</tr>
<tr>
<td></td>
<td>Verification of whole new method</td>
<td>public void hello() {</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Invocation of changed class methods OR</td>
<td>generateServerResp(..);</td>
<td>0.7</td>
</tr>
<tr>
<td></td>
<td>over objects of such class (ex = object A)</td>
<td>this.generateServerResp(..); A.generateServerResp(..);</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Invocation of methods from a class B</td>
<td>B.getMessage();</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>over objects of such class (being B a</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>class from the project/repository)</td>
<td>System.out.println(…);</td>
<td>0.4</td>
</tr>
<tr>
<td></td>
<td>Invocation of methods from Java Lang</td>
<td>LinkedList&lt;?&gt;.add(…);</td>
<td></td>
</tr>
<tr>
<td></td>
<td>or imported libraries</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variables</td>
<td>Declaration of non-primitive type variables</td>
<td>Message m = …;</td>
<td>0.3</td>
</tr>
<tr>
<td></td>
<td>Declaration of primitive type variables</td>
<td>int i = …;</td>
<td>0.1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>boolean b = …;</td>
<td></td>
</tr>
<tr>
<td>Fields</td>
<td>Field Declaration</td>
<td>public static final int N = 4;</td>
<td>0.5</td>
</tr>
<tr>
<td>Loops</td>
<td>Method calls inside ‘if’ conditions</td>
<td>If(amount &lt;= 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Method calls and variable declarations</td>
<td>for(Message m : responses)</td>
<td>Based on previous processing values</td>
</tr>
<tr>
<td></td>
<td>inside a ForEach loop condition</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Method calls and variable declarations</td>
<td>for(int i = id – 1; i &lt; pendingOperations.size(); i++)</td>
<td>Based on previous processing values</td>
</tr>
<tr>
<td></td>
<td>inside a ‘for’ loop condition</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Method calls inside a ‘while’ loop</td>
<td>while(app.getReceivedAcks() &lt; Server.N – (Math.floor((Server.N – 1)/3)))</td>
<td>Based on previous processing values</td>
</tr>
</tbody>
</table>
preparing the data that will be processed. It will filter the classes that come from the API package structures (usually in a PRCommit object), prepare the relevant data from such classes in a format that will help our process and feed such information to a Visitor object that will startup the data analysis. Key attributes of this class are:

- List<ChangedFile>javaFiles - Structure with the filtered files that will be processed
- List<ChangedFile>notJavaFiles - Structure with the filtered files that won’t be processed
- Visitor visitor - Visitor object for the analysis

Visitor Class: this class is responsible for the startup of the patch analysis. It will receive from the Core class the changed file that will need to be analysed, parse the relevant content and iterate over the parsed output. The class adopts a Visitor pattern behaviour, so it will, based on the type of the produced parsed output, call specific classes for its analysis.

<NodeType>Process Classes: these classes are responsible for processing each type of nodes that are visited by the Visitor class. So each class is specialized in its own node and more classes may be added for new desired nodes. Key attributes of these classes are their own IMPACT_VALUEs that match the specified Table 3.1 ones.

SmellChecker Class: this class is responsible for the code smell detection phase of our analysis process. It will perform code smell analysis with the help of two different tools, CheckStyle and PMD. The class was created with the intention of making the code smell phase easy to be modified by an experienced user as pleased.

3.3.2 Parsing the patch content

In order to analyse and process the content of a patch, that is, the changes that were made to a certain file, we will need to be able to read each line and understand the symbols and the content that comprises such line, or string to be more precise, that means that we need to parse such strings.

Since we are performing our analysis over Java projects, it is crucial for us to understand and know how the syntax of such programming language works as we will need to identify the language’s meaningful symbols and rules present in a string, filter, map and access them and then work over the ones that we find important.

To perform the parse of the content we wanted, we had two possible approaches, to create our own code able to read and identify every symbol and rule of the Java language on each string, or to find
existing code (a parser) that could do such job for us and from whom we could access all the information we needed for our tool in a simple and organized way.

The first approach would be very difficult and time consuming to follow, as creating a parsing tool is no easy feat and that was not the main goal of our tool as we wanted to avoid spending as much time as possible developing code that would be auxiliary (or secondary) to the main purpose of the tool.

Following the approach of finding a parser we would need one that could parse Java code but also that could be easily integrated in our tool, and by that we mean finding a Java parser written in the Java language would be the best possible solution. Most of the existing parsers are written in a language different than the language they parse, that is done because there are some programming languages where it is easier to code all the rules and find all the symbols present in a string making them the go-to languages to implement such parsers.

Fortunately we found an open source parser for Java written in Java with a well documented API and regular updates that follow the evolution of the Java language. That parser is simply called javaparser [26] and with it we could simply create a dependency in our tool project for it and invoke the parser how we pleased.

3.3.2.A Javaparser tool

As mentioned above javaparser is a tool made to parse Java code. By passing the source code to the parser it will analyse its contents and produce an Abstract Syntax Tree (AST) of the file. An AST is a tree that represents the syntactic structure of source code where each node of the three denotes a construct occurring in the source code. A construct may take many forms, in Java for example, it may come in the form of a if-condition, a method call, any kind of expressions (delimited by ;) and so on.

Since the produced tree will have many nodes in the form of every construct present on the source code given to the parser, we will be able, through the API of the javaparser, to traverse the entire tree and extract all kind of information about each node, thanks to the processment done by the parser over the source code.

Just to give an example, if we want to look a bit deeper into a method declaration, we may ‘select’ such node and extract information about it. The information may take many forms like, the method’s arguments, its name, the type it returns, its body, among other things. The great advantage about this is that, we can then individually ‘select’ and extract all this type of information given in the example because they are not only nodes of the method declaration but also nodes of the entire tree.

Since we want to filter and select only the type of rules and symbols that we find important for our future analysis, the job done by the parser will simplify our task as we may traverse the tree, visit only the type of nodes that match our interests and extract the information we need and do our processing over it.
Most parsers simply produce an AST from the given input and that's it, they usually don't give us a deeper insight about declarations and connections that exist in the source code. However the parser we used has such functionality. It is able to associate a certain symbol (like a field or a local variable) to its declaration. This means that we have a whole new kind of information about many constructs. Now we'll be able to identify the class of a variable regardless of where she was declared, know the type/class of an object that is being called and so on, providing us the kind of information that will be really useful for our analysis processment while saving us a lot of time as we won't need to write new code to create such complex functionalities.

3.3.3 Impact Value

The creation of a value to identify which classes of the commit were the most 'important' ones was a must. However there was one problem with the implementation of such value, namely, which impact value should each case have?

We didn't find a plausible way to answer this question or to assign values to each case because the importance that is given to them is always a subjective topic, it depends on the perception that each reviewer has regarding them. Even after some research we didn't find studies that could help us define our order of importance. The best way to draw some conclusions could've been to perform an inquiry before our implementation phase, asking users on the importance of every defined case for them. However, its not easy to recruit users to perform such inquiry in our academic environment, and since we wanted to perform user tests later on our tool, we decided that we could ask these questions about the ordering at that stage and, taking into account the responses, adjust the impact values that each of our cases have.

So, based on our intuition and perception, we created our own order, that would be used on the user tests, and assigned values that range from 0 to 1. The choice on this interval was just to make the final impact value of the file not too big.

The value of each case is declared on its own <NodeType>Process class, so an experienced user can create his order, change the values on the source files, recompile the project and the tool will create an order based on the new values. However, this may not be the best approach for all users, so adding a new functionality that would give them the possibility to change these values on the startup of the tool could've been a better choice from us.

3.3.4 Analysing a patch

Our analysis process starts with the Core class of this package. As the process was implemented in a way to analyse commit by commit, as they are requested, we implemented a method in our Core
class, *processCommit*, that processes a single *PRCommit* object that came from our *API* package. The method will filter the commit’s changed files to be processed, prepare all the analysis’ needed data, and then feed it, along the files, one by one, to our *Visitor* class to process them.

Since we are performing a static analysis, we needed to parse the patch of each changed file to understand and identify which of our defined cases are present in it. To parse source code, we need to give to our parser a well formatted input so he can create an *AST* out of it. Most parsers require the entire, and error free, source code of a class to perform such task, which makes it impossible for us to parse the patch, as it may represent only a portion of a class and sometimes even portions of statements of a class. Nonetheless, the parser we used has so many functionalities that it gave us the opportunity to parse only a single statement of the patch and to analyse it by itself. This approach had some problems in it. The major one was that if we parsed the patch line by line we could be splitting statements that were declared in more than one line leading to parsing errors and thus to unprocessed patch content and inaccuracy in our analysis.

We had to follow a different approach, an approach in which we wouldn’t need to parse the patch but instead we would match the patch with the version of the file of that commit, parse that file and analyse from there.

### 3.3.4.A Matching patch with source file content

With the impossibility of parsing the patch to analyse its contents, we had to find a workaround for that problem. The solution we came upon was to retrieve changed the file’s source code in that commit version, as that version contains the new lines present in the patch and those are the lines we are interested in, parse that code and then traverse the produced *AST* and match its nodes with content present in the patch. With this approach, instead of analysing source code straight from the patch, we would analyse, and take advantage of the parser functionalities, an *AST* node that corresponds to code in the patch.

To match the code in the patch with a node of the source file *AST* we tried to verify if their contents were the same, that is, if the string content of a line of the patch was equal to the content of the node, that was also a string. This could produce false positives and also false negatives. For example if we had a statement in the patch that was split into multiple lines and only 1 of those lines was changed, we would try to match that changed line with the content of a node and we would get no correspondence because the node that represented that statement had in its content string all the lines and not only the changed one. We could also have cases in which multiple nodes had the same content string across the file, and we could match the changed line with the node in the wrong position and that was an inaccurate match.

So instead of looking at the strings of the contents we decided to try to match through their line po-
sitions. One of the advantages of parsing source code to an AST is that we can get multiple information about each produced node. With the library we used, we could retrieve the range of each node, that is, the source code lines’ numbers where the node begins and ends. This way we compared the line number of a changed line in the patch with the interval produced by the start and end of each node. If the line number of the patch was inside the interval of a node, that meant that that node corresponded to that change in the patch, so we could isolate him and analyse that entire node in our own way. But to implement this approach we had to number the patch lines’, as that information isn’t retrieved through the API. However, when retrieving a patch there are some fields that indicate where the changed lines are starting and the number of new lines and so on, so with this information we were able to implement a method, `addChangedPatchAdditionsLines` that received a patch and would return a Map where each of its entries are the line number and the line content. Now we could easily match each changed line to a node in our AST.

3.3.4.B Searching and processing a node

The best way to iterate, search and add some new functionality without changing the nodes’ structure in the AST produced by the parser is to take advantage of the visitor design pattern in Java. ASTs may contain an enormous amount of nodes of different types and scenarios may arise when, for some specific ones, we want to perform some action that is not implemented in their core structure, that’s when the pattern is very powerful.

In our implementation the AST produced by our parser contains several different types of nodes, each of them representing a structure or an element present in the Java language like an ‘if statement’ or a method declaration. From all these types of nodes, we wanted to visit the ones whose type is compatible with the cases we defined and perform our process based on such node type.

The parser’s library had already some defined visitor’s for the AST structure, so we could take advantage of that and use such implementation. With this in mind we created a Visitor class responsible for parsing a ChangedFile object and extending the functionality of one of the library’s visitors in such a way that, when the visitor visits a node of our interests it will then call an object of `<NodeType>Process` to perform a specific analysis for that type of node. With this implementation we then had to create a number of `<NodeType>Process` classes equal to the number of unique nodes that interested us. This approach makes it easier to perform bug tracking and node analysis improvement, as when a node processment isn’t working as intended, we’ll know exactly where to find the problems. It also makes it easier to add new types of nodes in our analysis and its respective process, simply need to visit the new AST node type in the Visitor class and then crate a new `<NodeType>Process` class with the specific analysis for that new node.
3.3.4.C <NodeType>Process classes

These are the classes responsible for analysing each type of existing nodes. Their creation was needed because each node of the AST may have multiple and different fields than the other ones making each of them unique, thus making their data access and processing different from each other. Also, the verifications we performed may also vary from node to node. Table 3.3 contains the classes we needed to create and a brief description of their behaviour.

3.3.4.D Method calls and related classes identification

One of the most important actions of our analysis is method calls processment, as method calls constitute the majority of the code of any Java class. Since we are in the object oriented paradigm, these calls may be done over objects of a class different than the one we are implementing and, in these cases, we want to be able to identify which class the object that is being invoked belongs to.

In our design we separated the method call category into three different cases based on the type of object that is being invoked, as we believed that different types of classes may have a different impact in a code review and in the understanding effort required by the reviewer, so they should be separated and prioritized.

The parser we used had an extension to it that created the context and the code flow of a source file, an ability that most parsers don’t have as they simply create an AST, allowing us to retrieve a lot more information about a node of the tree than usual. As this extension could create connections and correlations between the multiple declarations and elements present in the source code, we could retrieve, through it, information about the caller of a method call and its class type, just as we wanted.

However for this functionality to work without problems and as accurate as possible the library needs the entire project’s source code in the local machine, so it is able to analyse all the files and correctly identify the class of the object that was invoked. This was something we didn’t have, as we didn’t download the repository where we are performing the review because we feel that, for a static analysis and code review, the reviewer shouldn’t be forced to download into his machine the version he is reviewing. We had to create a workaround for this situation, even knowing that it might not be always accurate, because the library wasn’t producing outputs that could be of use for us from our requests.

We had two possible outputs when asking the library to identify the type of the invoked object, it would answer nicely and correctly if the call was made over an object belonging to the Java language or running in that Java Virtual Machine (JVM), or it would throw an exception from which we could retrieve some information.

The first output matched one of our defined scenarios so we had no problems with it.
Table 3.3: Table of <NodeType> Process classes

<table>
<thead>
<tr>
<th>Class</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>ClassOrInterfaceDeclarationProcess</td>
<td>Increments this class impact value to the impact value of the ChangedFile object. Also, checks if the patch represents a whole new class, if so, calculates and increments the bonus value to the impact value of the ChangedFile object.</td>
</tr>
<tr>
<td>ConstructorDeclarationProcess</td>
<td>Increments this class impact value to the impact value of the ChangedFile object. Also, checks if the constructor is new, if so, calculates and adds the bonus value to the impact value of the ChangedFile object.</td>
</tr>
<tr>
<td>FieldDeclarationProcess</td>
<td>Increments this class impact value to the impact value of the ChangedFile object.</td>
</tr>
<tr>
<td>MethodCallExprProcess</td>
<td>Increments this class impact value to the impact value of the ChangedFile object. The value added depends on the three possible invocation cases defined on Table 3.1. Performs checks on the invoked object to see if he belongs to the repository. If so, also adds such file name to the relatedJavaFiles structure of the ChangedClass object.</td>
</tr>
<tr>
<td>MethodDeclarationProcess</td>
<td>Increments this class impact value to the impact value of the ChangedFile object. Also, checks if the method is new, if so, calculates and adds the bonus value to the impact value of the ChangedFile object.</td>
</tr>
<tr>
<td>VariableDeclarationExprProcess</td>
<td>Increments this class impact value to the impact value of the ChangedFile object. The value added depends on the possible variable cases defined on Table 3.1. Performs checks on the variable object to see if he belongs to the repository. If so, also adds such file name to the relatedJavaFiles structure of the ChangedClass object.</td>
</tr>
<tr>
<td>ExpressionStmtProcess</td>
<td></td>
</tr>
<tr>
<td>ForEachStmtProcess</td>
<td></td>
</tr>
<tr>
<td>ForStmtProcess</td>
<td></td>
</tr>
<tr>
<td>IfStmtProcess</td>
<td></td>
</tr>
<tr>
<td>ReturnStmtProcess</td>
<td></td>
</tr>
<tr>
<td>TryStmtProcess</td>
<td></td>
</tr>
<tr>
<td>WhileStmtProcess</td>
<td>Implements a visitor for more specific nodes present in this node type.</td>
</tr>
</tbody>
</table>
For the output that produced an exception, we then looked at his message and were able to retrieve the name of the class of the object that was invoked, the class that the library was unable to find, thus throwing the exception. From the name of the class we could do some manipulation and try to infer which specific class is that. Looking at the imports present in the source code of the ChangedFile being analysed we looked for the ones that matched the class name we found. From there we isolated the match (or possible matches, as we may import classes with the same name but from different packages, even though its something rare), and searched our API object repository:JavaFiles structure for a file that matched the import we isolated. If a match was found, we assumed that the method call was performed over an object of the existing repository and incremented the value from that scenario, if no match was found, we assumed that the call was made over an object of an imported library.

For the imports that matched files existing in the repository, we then added such file path to the related:JavaFiles structure present in the ChangedFile object so this information could later be shown to the user in our view.

3.3.4.E Impact value formulas of new whole level structures

As shown in table 3.1 most of our defined cases have a fixed impact value assigned to them. However, there are three cases whose impact value is calculated through a mathematical formula making their value dynamic. The reason behind this decision is because, in these cases, we wanted to give an extra bonus to the impact they generated. For example, while analysing a whole new method, our tool will find in such method all the method calls and all specific cases defined in our table, however, since this method is completely new, we thought that it should be given an extra impact on its ChangedFile impact value because new methods are relevant and may require an extra effort from the reviewer in their understanding. The same concept applies to whole new classes and constructors.

Nonetheless, we didn’t want to give a bonus impact value too big, as the specific cases are still being accounted for and we needed to have a fair balance in our analysis of such cases.

Whole new method and constructor formula: we believe that bigger methods or constructors may be harder to understand, as the more lines of code they have the more operations, calls and complexity may be present in them. However this is not always the true because blank lines and comment lines may increase the lines of code present in a method’s body. Even with the ‘noise’ introduced by those kinds of lines we decided to give an impact value based on the body size of such methods or constructors, by dividing their line count by 10 so the produced value isn’t too big compared to our scale of defined impact values. This approach gives more importance to bigger methods and constructors as they are more prone to require more understanding effort.
Whole new class formula: following the same concept applied to the methods and constructors, whole new classes should be given a bonus *impact value* according to their size. However in the case of whole new classes, all their methods and constructors are also new and their bonus are already being calculated and incremented in the ChangedFile impact value, so we wanted to mitigate this factor a little to not take it into account twice. So our formula took into account the number of lines of the class’ body, its size, and the number of methods and constructors defined in the class. With these chosen values, we divided the number of lines of the class by the sum of its methods and constructors plus 1, to avoid divisions by 0, and then divided such value by 10 just like the previous case. This gave us a decent bonus value for new classes.

### 3.3.4.F Code Smell Detection

Despite the subjective aspect of the code smell characteristic, and because we are parsing and working over source code, we thought that displaying some information about existing code smells in the patch of a changed file could help code reviewers assess the quality of the patch making easier their decision of approving it or asking the patch developer for some changes in the ‘smelly’ or problematic zones of the patch before approval. If repository integrators follow the principles we test the patch against and see the enforcement of such principles mandatory before the approval of a patch, displaying such information straight away will save reviewers a lot of time, as they won’t need to inspect the patch looking for such problems and can focus on the change understanding or other particularities.

After we finish analysing the patch and have assigned the ChangedFile object its final *impact value*, we then perform a code smell inspection over the patch through a new class named SmellChecker. This class will use two code smell detection tools, PMD [25] and Checkstyle [24], to inspect the source code of the changed file in search of code smells, and will give us an output with their detections.

The produced output identifies in which line the smells exist and information regarding their type, so in order to match the smells with the patch, we use the line numbers present in both data.

Each smell detection output was saved in temporary text files. After that, we read and analysed each produced file line by line. For each line we isolated the source code line number of the smell and then inspected the patch for a line with the same number. If the patch contained a line with the same number as the one we isolated, that meant that the patch contained such smell, and so we saved the smell information in a structure present in the ChangedFile object being analysed. Such information was then displayed to the user in our view.

**PMD tool**: PMD is a static source code analyzer that finds some types of code smells like unused variables, empty catch blocks and so on... These kind of detections are configured in a standard rule set defined by the tool, however, a skilled user may configure and write new rulesets for the tool so it
can detect other type of code smells desired by him. We used the standard rulesets already defined as we thought they detected the type of smells we wanted to display to the user, the ones that may contain useful information regarding the patch.

**Checkstyle tool:** Checkstyle is a static source code analyzer that verifies if the written code adheres to a certain standard, that is, unlike PMD, it looks to the way code is written, like if variables’ names match a certain pattern predefined by the project owner, line length, and so on. For this tool to work correctly we also have to give it a file with the ruleset for the standard we want to test against. In most Java projects the followed standard is Google Java Style Guide [27] and it's this one we ask Checkstyle to test our patch against.

We used both these tools in our code smell detection phase because we feel that they identify different types of smells that may be useful for the reviewer. PMD detects more technical code smells present in the source code while Checkstyle detects problems in the way the code is written like a beauty checker.

By giving the reviewer both of these data we aimed to save him the time required for inspecting such type of aspects, in case they are a must and these standards are followed by the project.

### 3.3.5 Known Issues

A static analysis may not always be completely accurate and ours is not different, so we identified some cases that, even though rare to occur, they still might happen once in a while and have a small impact in the calculated *impact value* or in the information showed to the user.

**New method or constructor check:** there is a case that even though a method or a constructor may be completely new our tool will miss it. This might happen because when performing this check, our tool asks the parser for the beginning and the ending of a method, and the ending corresponds to the line where its closing brace is. However, we found some cases of patches where the whole method was new but the closing brace of the method was already present in the source file, because an old method could’ve been completely deleted and its brace used, which would make this brace not to be present in the patch new lines that we use in our analysis. When this happens our tool considers that the method wasn’t entirely new as there was one line from the method that wasn’t present in the patch’s new lines.

**Class and method size:** when calculating the size of a class or method, the blank and comment lines present in their bodies are taken into account. This may give more importance than is intended to a certain class or method, as its size may return a big value due to this ‘noisy’ lines and not due to the amount of useful lines of code present.
**Variable Assignments:** variable assignments are not being taken into account. So if an object or attribute is initialized through a "new ....()" in an assignment, it is not detected and it may be of value to detect if the new object is of a class present in the repository.

**Method calls object type identification:** we’ve already explained the workaround required for our tool to detect the type of the object that is being called as the parser doesn’t work completely accurately unless we have the entire repository in our local machine and feed it its path to analyse. In a multi-call statement we will only be able to retrieve the type of the first object, that is, if multiple calls are made and over different types of objects (because of the return value that comes from each call), the calls are all detected but only the first caller is taken into account and its type discovered.

**Related files:** when identifying the related files present in a changed file patch there are cases in which we can’t accurately determine the right file. If the repository contains classes with the same name and the same package but with a slightly different folder path, we won’t be able to tell which of those classes is the imported one. In this case, we display all the files that match the import present in the changed file.

Another issue is when a new class is created in a pull request or commit being analysed and such class is used in another changed file present in that same pull request or commit, we won’t be able to identify it as a related file because the new class isn’t present in the master branch of the repository, as we test against the data present there.

### 3.4 Display Package

This module is responsible for the startup of our tool and its function will be to interact with the user and create a HTML view based on his requests. The module will communicate and retrieve data from the API package and with the help of a framework will create dynamic HTML pages able to use and display the Java information that is running through our tool.

#### 3.4.1 Spring and Thymeleaf Frameworks

The final goal of our tool was to display to the user an HTML page with the relevant information created and processed by our core module. Since 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. Fortunately there are frameworks able to do such work for us and some of the most used and known ones are the Spring Framework [28] and the Thymeleaf Framework [29].
With the integration of Spring in our tool we were able to transform the tool in a web application that was easy to implement, run and interact with.

The Spring Framework does all the heavy work for us, runs a webserver in our machine and takes care of all the configurations needed for our tool to run as a Web Application. We simply need to write Java code that conforms to the implementation and requirements of the framework and it will take care of creating all the defined webpages (endpoints) and bind the right Java objects to such endpoints and so on.

Thymeleaf is a HTML template engine and its usage allowed us to create dynamic HTML pages quite easily, as it lets us to access a Java object's data and fields quite easily and display that information in multiple ways inside HTML elements.

We chose these frameworks because we already had previous knowledge about them, acquired through the Software Engineering course, and because, thanks to their wide usage in our field of work, they have a lot of documentation and are also an ever changing and evolving frameworks with lots of features and security.

3.4.2 Diff2html tool

As we wanted to create a view as close as possible to the one that displays GitHub's patch information, one of the main challenges we faced was to recreate and implement some of the HTML elements of the page containing the changed files and their patches, more precisely the patch box displaying the diff information.

Before starting the attempt of recreating such elements we did some research looking for libraries or HTML templates that matched GitHub's website. As far as HTML templates go, we didn’t find one that could replicate the GitHub one so we searched for libraries that we could use on our own HTML and that were able to create a diff element box similar to GitHub.

Fortunately we found one library that could do such work. Diff2html [30] is an open source library that given a patch as input can generate HTML similar to the one provided by GitHub, plus giving us some options to configure the way the patch information is displayed, some of them useful to help code review and change understanding. The diff2html library may be used from the command line to create entire HTMLs for the inputs provided but can also be invoked from a Javascript library through our own HTML to return a HTML element (div) containing the patch box.

3.4.3 Technical Information

This module of our tool is composed from the following relevant classes:

- Display - main class of the module
• APIData - data class containing information regarding one API object of the tool

• APIInterface - class responsible for storing and managing APIData objects.

• <ViewType>Controller - classes responsible for requesting and processing information regarding a certain request from the user and managing the HTML views and endpoints

**Display Class:** this class simply contains one main method for starting up the entire tool as a Spring Application.

**APIData Class:** this class is responsible for the creation and storage of an API object and its information when a user asks for an online GitHub repository. The creation of this class made it easier for processing and exchanging data between this module of the tool and the other 2 modules without adding too much complexity on other classes. Key attributes of this class are:

- String repoToScanOwner - Name of the owner of the requested repository
- String repoToScanName - Name of the requested repository
- API api - API object created
- Core core - Core object used for the patch analysis

**APIInterface Class:** this class is responsible for storing all the APIData objects that are created. Since our tool doesn’t use a database and we need to store the objects that are running so we can retrieve their information to create the views that will be displayed to the user, we decided to create a class responsible for such storage and management of the data. Key attribute of the class is:

- List<APIData> apiData - containing all APIData objects

**<ViewType>Controller Classes:** these classes are responsible for requesting and processing data depending on the type of request the user performs. The controllers match a certain HTML endpoint, that the Spring Framework will create, with a certain HTML page file. The data they request and process, through the use of the API and Core objects present in the APIData object, will then be used on a HTML file. These controllers are the bridge between the Java data of our objects and such HTML page files.

Besides these Java classes, the module also contains HTML and Cascading Style Sheet (CSS) files. The HTML files are a must so our framework can use them and work as a web application. We created as many files as we needed and implemented them in a way that we thought that would make it easy for the user to navigate through them on the browser. More information regarding the displayed HTML
elements explained below. The CSS files were just a complement to our HTML pages to separate the logic between the skeleton of the page and its customisation elements.

3.4.4 HTML pages

As we have already mentioned, some modules of our tool were implemented in a way that they perform 1 request at a time to the public GitHub API or process 1 commit at a time so the response time of our tool doesn’t greatly increase between requests performed by the user, making his waiting time as little as possible.

With this in mind, our webapplication is composed of a few HTML pages that will guide the user step by step till he reaches the final processed patch he wants to review. This step by step behaviour is achieved by having each HTML page display only one type of information from which the user can then perform a more specific request regarding the displayed data. By performing such request, the user will then move to a new page where the data retrieved from his request will be shown.

3.4.4.A Repository Webpage

This is the first page presented by our tool. Here we ask the user, through a simple form, for the name and the owner of the repository he wishes to analyse as shown in Fig. 3.2.

The transition from this page to the next one may take some time, if the information provided by the user is correct, we are creating and setting up the API object and retrieving the file’s paths that exist in the repository, and as already mentioned, this is a known issue of the API package. With this in mind, we provide a warning in red to make the user aware that this step may take some time to process.
3.4.4.B Pull Requests Webpage

This is the page shown to the user after the initial repository scan performed by our tool. Since we are interested that our tool is used to perform code reviews over pull requests, in this page we will only show which pull requests are present in the repository and in the open state, meaning, the ones that still require review. Fig. 3.3 is an example of our pull request webpage, we display the pull requests in a table containing their identifiers, the number and the title, in descending order, making the newest pull requests stay at the top.

Besides this table, we also provide a navigation bar at the top of our webpage so the user can easily move to another page (step) of our tool. From the pull request page we can only move to the initial page and scan another repository.

When the user clicks on the pull request he wishes to review, the tool will redirect the user to a new page containing all the commits present in that pull request.

3.4.4.C Pull Request Commits Webpage

This is the page shown to the user after he chooses one of the pull requests displayed in the pull request webpage. As we can see in Fig. 3.4 we provide a simple table containing all the commits present in the pull request. We show their identifier, the commit sha, and their message, so the user can easily distinguish each commit.

Again, the top navigation bar is present and now with an option to return to the pull request page, the previous step of the tool.

Clicking on a commit sha will trigger the actions described in Sections 3.2.6 and 3.3.4 and redirect the user to a new page where he can see the results produced by our analysis. We chose to perform this actions only at this stage to avoid unnecessary processing and high response times between previous
steps, as when the user reaches this stage he most likely has the certainty of what he wants to review, and so we can spend processing resources on this final request.

3.4.4.D Commit Patch Webpage

In this page we display the patch of each changed file, with the help of the diff2html library to create a patch box similar to GitHub’s, along with other small elements that provide information regarding the code smells and some quick access links. The files are already ordered according to our performed analysis.

Fig. 3.5 illustrates the beginning of our page, the navigation bar is present, now with another option to return to the pull request commits. We can see part of the patch of the file with the highest impact value of the commit and three lateral boxes containing information regarding such file and patch.

File Info box: in this box we provide to the user two quick access links of the file, one for its version present on the master branch, which should be the version from which this patch derived, and another link for the version containing the present patch. The link to the master version of the file may lead to an inexistent page in case the file is new and was added in this pull request.

Related Repository Java Files box: here we provide quick access links to the master version of the files that were used in the changed file’s patch. This information is given to the reviewer to minimize context switching time, as sometimes he may need to read and see what a certain method from the
invoked class does, and these links reduce the search time for the file.

**Patch Code Smells (according to Google standard) box:** contains the code smells present in the patch that were identified through the use of the *Checkstyle* tool.

**Patch Code Smells (according to PMD) box:** contains the code smells present in the patch that were identified through the use of the *PMD* tool.

Fig. 3.6 contains a patch with code smells detected by both tools and aims to demonstrate the difference between the type of smells each of them find. We can see that the code smells detected through *Checkstyle* using the Google standard looked at patterns on how the variables are written and the existence of Javadoc documentation, it also finds smells like line length and indentation standards. While the *PMD* analysis gives us a little more technical information like bad practices in implementation as shown in the figure, empty constructors or methods, unused variables and so on.

Lastly in Fig. 3.7 we can see that in commits containing files other than Java, such files will simply be put at the end of our view without any type of analysis performed over them.
Figure 3.6: Page showing our produced view of a patch with both code smell boxes

Figure 3.7: Page showing our produced view of a patch with non Java files present
### 3.5 Algorithms

In this section we present some of the most important algorithms that may directly impact the performance of our tool. Their pseudo code will be shown and we make some observations regarding each one.

**Algorithm 3.1**: High level example of a Improved Commit Views execution

<table>
<thead>
<tr>
<th>Result: Final Commit View</th>
</tr>
</thead>
<tbody>
<tr>
<td>String repositoryName; // user input</td>
</tr>
<tr>
<td>String repositoryOwner; // user input</td>
</tr>
<tr>
<td>prs = Map{Key: prNumber, Value: prTitle};</td>
</tr>
<tr>
<td>prCommits = Map{Key: commitSha, Value: commitMessage};</td>
</tr>
<tr>
<td>commit;</td>
</tr>
<tr>
<td>API api = new API(repositoryName, repositoryOwner);</td>
</tr>
<tr>
<td>Core core = new Core();</td>
</tr>
<tr>
<td>prs = api.getRepositoryPRsInfo();</td>
</tr>
</tbody>
</table>
| <displays view with "prs">
| prCommits = api.printRepositoryPRCommits(<PR inside prs>.number); |
| <displays view with selected PR commits>
| commit = api.getRepositoryPRCommit(<PR inside prs>.number, <COMMIT inside prCommits>.commitSha); |
| core.processCommit(api, commit); |
| <displays final view with our file listing and informations>

Algorithm 3.1 represents a pseudo execution of our tool from the time the user inserts the repository he wants to analyse, and through his choices of pull request and commit, till he reaches the final commit view created through our processing algorithm. The next algorithms are the "deep dive" of this main
execution to better understand what our tool is doing behind the scenes.

**Algorithm 3.2: API retrieveRepositoryFilesNames recursive function pseudo code**

<table>
<thead>
<tr>
<th>Input:</th>
<th>String contentPath</th>
</tr>
</thead>
<tbody>
<tr>
<td>Result:</td>
<td>List inside an API object containing all the Java files present in the requested GitHub repository</td>
</tr>
</tbody>
</table>

boolean hasDirectories = false;
List<RepositoryContent> rootContent = null;

// verification of contentPath and retrieval of repository contents to rootContent;
repositoryContent; // object that can be any file or directory of the online repository

for repositoryContent in rootContent do

  if repositoryContent is directory then // if content is of type directory

    hasDirectories = true;
    while hasDirectories do
      hasDirectories = false;
      retrieveRepositoryFilesNames(repositoryContent.Path);
    end
  end

else

  // checks if Java file and saves it inside API object;
end

Algorithm 3.2 is executed along the creation of the API object on the main execution. It is responsible to recursively retrieve and save, through the Java API library, all the Java files' names that exist on the repository. Depending on the number of files that the repository being retrieved has, this algorithm may take a while to execute, his time to finish grows depending on them, creating a performance problem during the startup of the tool. Also, the more Java files the repository has, the bigger will be the Lists inside the API object.

**Algorithm 3.3: API getRepositoryPRsInfo function pseudo code**

| Result: | Sorted Map containing the identifiers of all the open pull requests of the repository |
| SortedMap<Integer, String> prs; |

// retrieval of GitHub repository open pull requests to variable auxprs;

for PullRequest pr in auxprs do

  prs.put(pr.number, pr.title);
end

return prs;
Algorithm 3.4: API printRepositoryPRCommits function pseudo code

**Input:** int PRNumber  
**Result:** Sorted Map containing the identifiers of all the commits of a pull request

```
SortedMap<String, String> commits;
for RepositoryCommit rc in auxcommits do
    commits.put(rc.sha, rc.message);
end
return commits;
```

Algorithms 3.3 and 3.4 are both mentioned because they also make API calls through the Java library. With that in mind, their execution time varies depending on the amount of calls that need to be made, that is, the more pull requests or commits the repository has, the more calls will be made increasing processing time.

Algorithm 3.5 is one of the most important algorithms inside section 3.2. He is responsible for retrieving and saving the data of a requested commit by the user. He also calls the function responsible for retrieving and saving the information of a pull request. This algorithm runs only once per commit to
improve the performance of the tool. When the same commit is requested more than once by the user he is simply returned. If the commit information has yet to be retrieved through the API library, its run time will vary depending on the amount of changed files that are present inside the commit and their size, as each file data has also to be retrieved and saved in new objects.

**Algorithm 3.6: Core processCommit function pseudo code**

**Input:** API api  
**Input:** PRCommit prCommit  
**Result:** Processes the given commit by analysing each changed file patch and changing their impact value.

<checks if the commit is already processed. if true, simply returns so impact values are not calculated more than once>;  
<filters commit all Java files to be processed>;  
for ChangedFile cf in javaFiles do  
  Map<Integer, String> patchAdditions;  
  <filters all the changed file patch added lines and calculates their source code line number into patchAdditions variable. Map{lineNumber, lineContentString}>;  
  core.visitor.process(cf, patchAdditions, api.repositoryJavaFiles);  
end

Algorithm 3.6 is responsible for the startup analysis of the commit. Its runtime is influenced by the amount of changed files inside in the given commit, and also by the time it will take our Visitor to fully analyse each file.

**Algorithm 3.7: Visitor process function pseudo code**

**Input:** ChangedFile cf  
**Input:** Map<Integer, String> patchAdditions  
**Input:** Map<String, String> repositoryJavaFiles  
**Result:** Processes the given changed file by parsing it, visiting desired nodes and applying specific analysis over each node.

<parses the ChangedFile cf entire source code to produce an AST "cfTree">;  
Traverses the entire tree "cfTree" visiting the desired nodes and invokes their NodeType>Process class process method>;  
SmellChecker.process(cf, patchAdditions);

Algorithms 3.7 and 3.8 are the ones where the static analysis of a changed file will take place. In 3.7 the entire source code of a changed file is parsed into an AST. We will always traverse such AST once and analyse the node types that are of interest to us, calling their respective analysis class. The runtime of this algorithm depends on the amount of nodes the AST has, as bigger trees will take more time to visit, and also on the time our code smell inspection libraries will take to analyse the changed file.
Algorithm 3.8: \(<\text{NodeType}>\)Process class sample process function pseudo code

Input: NodeType node
Input: ChangedFile cf
Input: Map<Interger, String> patchAdditions
Result: Processes the given node type characteristics and contents. Changes cf impact value based on analysis findings.

\[
\text{for } \text{int } \text{key } \text{in } \text{patchAdditions do}
\]
\[
\begin{align*}
\text{if } & \text{node.begin } \leq \text{key } \&\& \text{key } \leq \text{node.end } \text{ then } \\
& \text{apply node specific processment, changing cf impact value}; \\
& \text{patchAdditions.remove(key)}; \\
& \text{return}
\end{align*}
\]

Algorithm 3.8 is a sample example of how a node is analysed. Through the attributes of the node being analysed, we try to match the number of its source code lines with the number of a changed line of the patch, as if the line numbers match, we know that the changed line represents the node being inspected. When a match occurs, we apply a number of operations, depending on the node type, to calculate the impact value to be incremented to the changed file. If a match occurs, and because it can only occur once, we remove the key from the patch, so in future node checks, there will be less keys to compare, improving the performance of our algorithm. The runtime of this algorithm depends on the size of the \(\text{patchAdditions}\) map. The more keys he has to compare, the more time our process may take, however, when the map becomes empty, the remaining nodes of the tree that will be visited will just enter and exit without performing any unnecessary checks or operations.

Algorithm 3.9: SmellChecker process function pseudo code

Input: ChangedFile cf
Result: Processes the given changed file source code looking for code smells according to a certain standard. Writes the found code smells into cf object.

\[
\begin{align*}
\text{creates temporary system text file containing cf source code, } & \text{"processfile"}; \\
\text{creates temporary system text file containing CheckStyle analysis output, } & \text{"processout"}; \\
\text{creates temporary system text file containing PMD analysis output, } & \text{"processoutpmd"}; \\
\text{PMD.process(processfile);} \\
\text{CheckStyle.process(processfile);} \\
\text{filters output files' content into cf object}; \\
\text{return}
\end{align*}
\]

Finally, algorithm 3.9 is responsible for the code smell detection of our tool. Its runtime is highly dependent of the calls we make to the PMD [25] and Checkstyle [24] libraries.
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1 Methodology</td>
<td>55</td>
</tr>
<tr>
<td>4.2 Data</td>
<td>56</td>
</tr>
<tr>
<td>4.3 Data Analysis</td>
<td>59</td>
</tr>
<tr>
<td>4.4 Threats to Validity</td>
<td>60</td>
</tr>
</tbody>
</table>
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.

In the following sections we will describe how we conducted our tests and how the received data backed our purpose or not.

4.1 Methodology

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.

Experiment preparation: one of the objectives of the experiment was to see if reviewing through our tool view was quicker than reviewing through GitHub’s view. To be able to draw some conclusions regarding this question, we divided our subjects into two different groups, one group that started the experiment reviewing through our tool, and timed the review, and another group that also timed their review but started the experiment reviewing through GitHub’s view. By having these two different beginnings of the experiment, we had some credible review times to compare.

The experiment was conducted always on the same repository. For that, we created a repository [31] that had a very simple and very unfinished base version of the \texttt{jpacman-framework} [32]. We then created
two different and independent pull requests that our subjects would review based on their IST student ID. Each subject only reviewed one of the pull requests, but we decided to have at least two to create a bit of diversity and to see how the review time was affected by different pull requests, as pull request size and difficulty may vary.

The parity of the last two digits of the IST student ID assigned the subject to the right pull request and whether the user had to start and time his review through our tool view or GitHub’s.

After the timed review, we asked the user to make a non timed review on the opposite interface so he could have all the necessary insight to answer our final form.

**Targeted users:** for the test subjects we wanted people that had some kind of experience with GitHub, software engineering or code review. Ideally we would’ve ran the tests with people that are already in the working industry or in an academic environment, so that we could have more diversity in our users sample, however, targeting users that are in the working industry was harder than the others. As so, we asked for the help of some ex-colleagues and colleagues that are enrolled in MEIC course, as they had already Git experience from previous courses and certainly are familiar with code reviews.

### 4.2 Data

In this section we present the data gathered from the user tests performed remotely by our subjects. The data is divided into two different subsections, each of them referring to the first and second part of a user test, respectively.

#### 4.2.1 Recruited User’s group characterization

As mentioned in the previous section, the first part of our evaluation consisted in the online recruitment of users to test our tool and the filling of a form that allowed us to characterize them.

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.4.1. 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.4.2, and that GitHub’s patch interface doesn’t need improvements.

### 4.2.2 Experiment Results

We had 2 users perform a timed review through GitHub’s webpage and 3 users through our tool, according to the adopted assignment method explained on section 4.1. 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.4.3. Fig.4.4 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 scenarios found that led to
Figure 4.2: GitHub's code review easiness as perceived by users.

Figure 4.3: User's perception of helpfulness of orders other than the alphabetical one.
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. Because our user group wasn’t as big as we wished it to be, we couldn’t form the different subgroups big enough to compare the timers in an objective way that allowed us to detect noticeable patterns on review time, making us unable to confirm if our tool helped reducing review time or not.

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 regard-
ing 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.

These observations occurred on a mainly academic group of users, so they may not reflect the same opinion that experienced industry users have, for that we needed more users in our group already in the job market.

### 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.

Also the fact that the experiments were conducted remotely and without our supervision prevented us to have a perception on how the users interacted with our tool and if they were doing everything as expected from our point of view, we had to blindly trust that everything went alright on their side and with the obtained results.
## 5 Conclusion

### Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1 Conclusions</td>
<td>63</td>
</tr>
<tr>
<td>5.2 Future Work</td>
<td>63</td>
</tr>
</tbody>
</table>
5.1 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.

5.2 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.
Bibliography


66


