12-2017

Approaches, Techniques, and Tools for Identifying Important Code Changes to Help Code Reviewers

Maneesh M. Mohanavilasam
Utah State University

APPROACHES, TECHNIQUES, AND TOOLS FOR IDENTIFYING IMPORTANT CODE CHANGES TO HELP CODE REVIEWERS

by

Maneesh M. Mohanavilasam

A thesis submitted in partial fulfillment
of the requirements for the degree

of

MASTER OF SCIENCE

in

Computer Science

Approved:

Young-Woo Kwon, Ph.D. Major Professor	Kyumin Lee, Ph.D. Committee Member
Tung Nguyen, Ph.D. Committee Member	Mark R. McLellan, Ph.D. Vice President for Research and Dean of the School of Graduate Studies

UTAH STATE UNIVERSITY
Logan, Utah

2017

ABSTRACT

Approaches, Techniques, and Tools for Identifying Important Code Changes to Help Code Reviewers

by

Maneesh M. Mohanavilasam, Master of Science

Utah State University, 2017

Major Professor: Young-Woo Kwon, Ph.D.
Department: Computer Science

Code review is a common software engineering practice employed widely in both open source and industrial context. It is tedious for code reviewers to keep track of code changes all the time. Especially since it is commonly recommended to have small commits frequently rather than having large commits. Manually browsing individual changes in multiple revisions makes code review difficult and inefficient. Software development is a highly time sensitive process and to make the code review process more efficient this research presents a code review tool that generates recommendations for the code reviewer and optimizes the code review process as a whole. The code review tool ranks the changes to be reviewed by the code reviewer, classifies each changes and labels them, provides additional refactoring information and lastly takes feedback from the code reviewer to readjust the ranking list dynamically in real time. To evaluate this research a total of 7 open source projects were analyzed and a single usability study was conducted. It was found that the code review tool: classifies changes as important and normal with an accuracy of 66.18%, summarizes refactoring types with an accuracy of 94.1% and detects incomplete refactoring with an accuracy of 98.4%. The results clearly show that the code review tool helps the code reviewers by providing recommendation that have a positive impact on their review and optimize the code review process.

(52 pages)

PUBLIC ABSTRACT

Approaches, Techniques, and Tools for Identifying Important Code Changes to Help Code Reviewers

Maneesh M. Mohanavilasam

Software development is a collaborative process where many developers come together and work on a project. To make things easy and manageable, software is developed on a version control system. A version control system is a centralized system which stores code and adds code from all other developers as an increment to the code base in the repository. Since multiple people work on the same code repository together, it is important to make sure that their contributions do not conflict with each other. It is important to maintain the quality and integrity of the repository. This is where the code review process comes into the picture. All the changes made to the repository by developers are reviewed by other, preferably senior developers, before it is integrated into the repository. This is done to maintain a high standard of development. The problem is that this is a manual and highly time consuming process. This research proposes a tool that tries to optimize the code review process. This is done by ranking the changes that the developers need to review: this makes it easier for the developer to decide which change he/she needs to review first. Also since every reviewer has their own preference and style, the tool takes feedback from the code reviewer after every change and readjusts the ranked change list according to his/her feedback. Adding to that, the tool classifies each change and tags it so that the code reviewers have a better understanding of the change that he/she is about to review. It also provides additional refactoring information about each change. Refactoring changes are very easy to miss, since they are not usually erroneous changes, but they erode the quality of the software overtime. The tool points out these changes so that these changes are not missed by the code reviewer. The research was evaluated on 7 open source projects and a usability study was conducted which prove that this tool does have a positive impact on the code review process.

ACKNOWLEDGMENTS

A special thanks to my thesis advisor Dr. Young-Woo Kwon at Utah State University. Dr. Kwon always had a solution to any road blocks I faced throughout my time at Utah State University. He consistently allowed me to explore my research interests and goals; but steered me in the right direction whenever he thought I needed it.

I would also like to acknowledge Genie Hanson, my graduate program coordinator for all the logistical support and Cora Price for being the second reader of my thesis. I am thankful for all her comments and feedback.

Lastly, I would like to thank my parents, Mohanachandran Mohanavilasam and Shaly Mohanachandran, for their encouragement and moral support. I could not have done it without all of you.

Maneesh M. Mohanavilasam

CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1
INTRODUCTION

Recent research [1] has revealed that over 30% of the total amount of code is repetitive mostly because of the copy-and-paste programming practice, the framework-based development, and the reuse of same design patterns or libraries, thus creating code change patterns (i.e., refactoring or bug fixing patterns). As code changes are repetitive, anomalous changes also could repeat either by a developer's own error or by other developer's fault unknowingly.

Firstly, it is well known that in software development process, the code review is a common software engineering practice employed in both open source and industrial context [2]. Same research also suggests that there is a huge scope for code review tools to make a positive impact on the quality of code churned out during the software development process. Hence, it is evident that code review is a process that has cemented its place in the software development life cycle. However, research also suggests that poorly reviewed code negatively impacts the software quality in large systems [3]. So there is a huge incentive to make sure that a code reviewer does a good job and maintains a high standard of review.

On one end we have repetitive code change patterns and on the other end we have a huge emphasis on manual inspection of code through the process of code review to detect and eliminate unwanted or erroneous code change patterns. All this suggests that it makes sense to build a tool to assist the code reviewers in the code review process so that they can have a positive impact and improve the overall quality of the system. However, code review is an expensive and time consuming process. It is tedious for code reviewers to keep track of code changes all the time. In particular, because it is commonly recommended to have small commits frequently rather than having large commits [4]. Browsing individual changes in multiple revisions makes code review difficult and inefficient. It is estimated that a code review can take on an average anywhere between 5 to 20 days [5]. In order to make a tool that actually helps the code reviewers review code more efficiently we must first understand the current challenges faced by them. According to the recent research conducted developers would rather postpone a review than rush through them, ignore review of issues they do not know enough about or stay away from issues that they think are uninteresting [6].

Due to the copy-paste nature of programming prevalent in the software industry tracking the evolution of code clones from a code review perspective is also very important. Many studies [7–9] illustrate the importance of code clones and its positive effects, however, there are other studies [10–12] that show that code clones are deterrent to the overall quality of the software. Code clones may be injected into a software repository with or without the knowledge of the programmers [13]. Also removing code clones completely from the software code base may not beneficial as illustrated by the studies Toomim et al. [14], Duala-Ekoko and Robillard [15], Hou et al. [16] and Nguyen et al. [17]. It makes more sense to manage clones in evolving software, track the changes as clone evolves in the code base and let the code reviewers make a decision about it. The idea of building an Eclipse plug-in for code reviewers is not new as illustrated by Lin et al. [18] that computes differences among clones and highlights the syntactic differences across multiple clone instances. The work in this research has a broader scope and is not just limited to tracking code clones. Integrated Development Environments (IDEs), such as Eclipse, enable the programmer with automated refactoring features. However, developers do not share a favorable opinion about letting an IDE automatically modify their code [19]. Thus this preference of developers to manually re-factor their code could lead to errors. This is supported by recent studies [20, 21] that clearly indicate an increase in the number of bugs due to the same. Furthermore, it is easy to omit incomplete refactorings as shown by Park et al. [21] which may affect the maintainability of the software as a whole.

To improve the productivity of the code review process, this research introduces an enhanced code review tool that ranks code changes, provides additional information about individual changes by classifying them and provides refactoring information: mainly detection of refactoring types or incomplete refactorings. Additionally, the described approach allows the code reviewer to express her preferences (i.e., feedback) during code review through a modern IDE (e.g., Eclipse), so that it is possible to customize the code review strategies.

First, a ranking model was developed that would rank revisions for the code reviewers in descending order of importance by extracting change information and code quality metrics from each revision. This would help them understand the severity of each revision to be reviewed. Second, a prediction model was built to tag each revision with a label. This labeling information provides more information and a little insight about the code to be reviewed. Third, a combination of a clone detection tool [22] and Abstract Syntax Tree (AST) pattern matching is used to generate refactoring information for the code reviewer. Finally, an avenue is left open to dynamically integrate feedback from the reviewer to customize these recommendations generated by the code review tool.

To demonstrate the benefits of this approach, the approach was evaluated with a total of seven third-party projects using the developed code review tool. The experimental result shows the effectiveness of the approach as the recommendation mechanism successfully: the tool ranked the revisions for the code reviewer, classified them meaningfully, identified refactoring types, and detected incomplete refactorings. As a result, the approach can reduce the effort of a code reviewer who has little knowledge of a code base. Overall, this research makes the following contributions:

- Ranking code changes: The approach described in this research has a well-defined model based on change information and software quality metrics that ranks changes for the code reviewer. A ranked list of changes gives the reviewer an understanding of what is important and helps the reviewer make more impact on the quality of the software through his review process.

- Classifying code changes: Code changes are also labeled by the tool giving the reviewer more knowledge about the change. As described in the section above, one of the major challenges a code reviewer faces is that she does not know enough about the change to be reviewed. This can help mitigate that.

- Identifying refactoring information: Most incomplete refactorings go undetected because they do not cause an error at run time. So providing refactoring information and refactoring anomaly detection can be a great help to the code reviewer who strives to improve the overall quality of the code base.

- A model for integrating user feedback: Different code reviewers have different review strategies and preferences. The tool takes feedback from the user and dynamically changes some of the recommendation according to the preference of the user.

- Empirical evaluation: Evaluation of the approach was done by conducting assessments on a total of seven third-party projects. The ranking model and prediction model was evaluated on one open source project while the refactoring type summarization and incomplete anomaly detection was evaluated on six open source projects.

The rest of this paper is organized as follows. Firstly, chapter 2 presents the approach of the code review tool. Next, chapter 3 presents the data collection strategy used to generate the data that helped in building the various models in the code review tool. Then, chapter 4 describes the implementation of the tools built to support the approach and conduct the evaluations. Lastly, chapter 5 empirically evaluates our approach and then we conclude this paper in chapter 6.

CHAPTER 2
APPROACH

Figure 2.1 shows the approach overview. The goal is to optimize the code review process by providing recommendations in form of a ranked change list, classification of individual changes, and clone refactoring information. The approach can be summarized as the assessment of the Project Repository by a Recommendation Engine with interactive user feedback. A code review tool aided by the Recommendation Engine acts as a medium between the code reviewer and the changes to the repository that need to be assessed. The code review tool is envisioned to be seamlessly integrated into a modern IDE so that is easily accessible to and accepts feedback from the code reviewer while at the same time minimizes any hindrance to her daily activities. The Recommendation Engine consists of four components; a Ranking Model, a Prediction Model, a Refactoring Information Generator and a Feedback Integration Model. These key components illustrated in the approach overview are explained below.

2.1 Project Repository

A Project Repository represents the modern version control system. A system in which multiple users add, modify or delete code from the code base contained in it. Each of these changes are tracked as an incremental addition to the code base and the users are given the ability to access, view or rollback these incremental changes. Along with recording code changes, a Project Repository also holds statistical information about the code change like: the time of the change made, the user that made the change and the message that was added along with the change describing it. The next section describes the Recommendation Engine which leverages these existing features of a software repository to provide valuable recommendations to the reviewer.

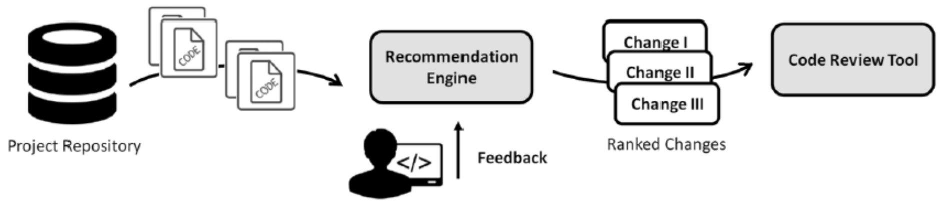


Fig. 2.1: General Approach Overview

2.2 Recommendation Engine

A Recommendation Engine analyzes the changes made to the repository across a specified number of revisions and provides recommendations to the code reviewer. It consists of four main components:

- Ranking Model
- Feedback Integration Model
- Prediction Model
- Refactoring Information Generator

Each of these components contributes to providing an array of recommendations to the reviewers. These recommendations are intended to give the reviewer a better understanding of the code base and help them in the process of code review. Each of these components is explained in detail in the sections below.

2.2.1 Ranking Model

The Ranking Model ranks the list of revisions to be reviewed by the code reviewer based on change information and code quality metrics of all the changes in the revision. An initial ranking is presented to the code reviewer when the code reviewer reviews a specified range of revisions for the first time. The Ranking Model dynamically changes the rank of these revisions in the presented list depending on the feedback from the reviewer. Thus the Ranking Model can adapt to different review strategies applied by the code reviewer.

The Ranking Model is a weight based model. Weight of each revision is computed and all revisions are ranked in descending order of the computed weight. The computation of the weight can be expressed as a sum of static and dynamic weight.

$$w = w_s + w_d$$

In the above equation, w is the weight of the revision, w_s is the initial static weight of the revision and w_d is the dynamic weight that gets added every time reviewer gives a feedback to the Ranking Model.

Initial ranking based on static weight

When the reviewer reviews the list of revisions for the first time a ranked list is generated which does not include any dynamic weight w_d since no feedback from the reviewer has been received by the Ranking Model. Hence the initial rank is computed just based on static weight.

$$w = w_s$$

The computation of the static weight can be expressed by the equation below:

$$w_s = \sum_{n=a}^{b} \mu_a * a_n + \mu_b * b_n + ... + \mu_z * z_n$$

For every revision, w_s is the initial static weight for all n changes in it. $a, b, c, ..., z$ are the code quality metric attributes used to generate the model and $\mu_a, \mu_b, \mu_c, ..., \mu_z$ are their respective importance factors. Importance factors can be used to set the impact of a certain attribute on the computed weight of the change because some attributes may be more important than the other. If all attributes are equally important then all their values can be set to 1. If so, the Ranking Model can be expressed as:

$$w_s = \sum_{n=a}^{b} a_n + b_n + ... + z_n$$

The attributes used in the model are selected by analyzing the data set described in chapter 3. The dataset has many code quality metric attributes each of which has been described in chapter 4. Gini index was computed for each attribute as illustrated in figure 2.2 and all the attributes that had an index score of more than 30 were selected. Some attributes like line, ncloc and statements were highly correlated to each other since they almost one and the same. In such cases only one among them was selected. The final list of attributes used for the model is:

- line: It is a count of the number of physical lines.
- comment_line_density: Describes the density of comment line. It is calculated by the formula:

$$Densityofcommentlines = \frac{Commentlines}{(Linesofcode + Commentlines)} * 100$$

- class_complexity: It is the average complexity by class.
- functions: It is the count of the total number of functions.
- sqale_debt_ratio: It is the ratio between the cost to develop the software and the cost to fix it.
- code_smells: It is the number of detected code smells.
- violations: It is the number of detected violations.
- open_issues: It is the number of open issues.
- duplicate_line_density: It describes the density of duplicate lines of code. It is calculated by the formula:

$$Densityofduplication = \frac{Duplicatedlines}{Lines} * 100$$

- reliability_remediation_effort: It is the effort to fix all bug issues.

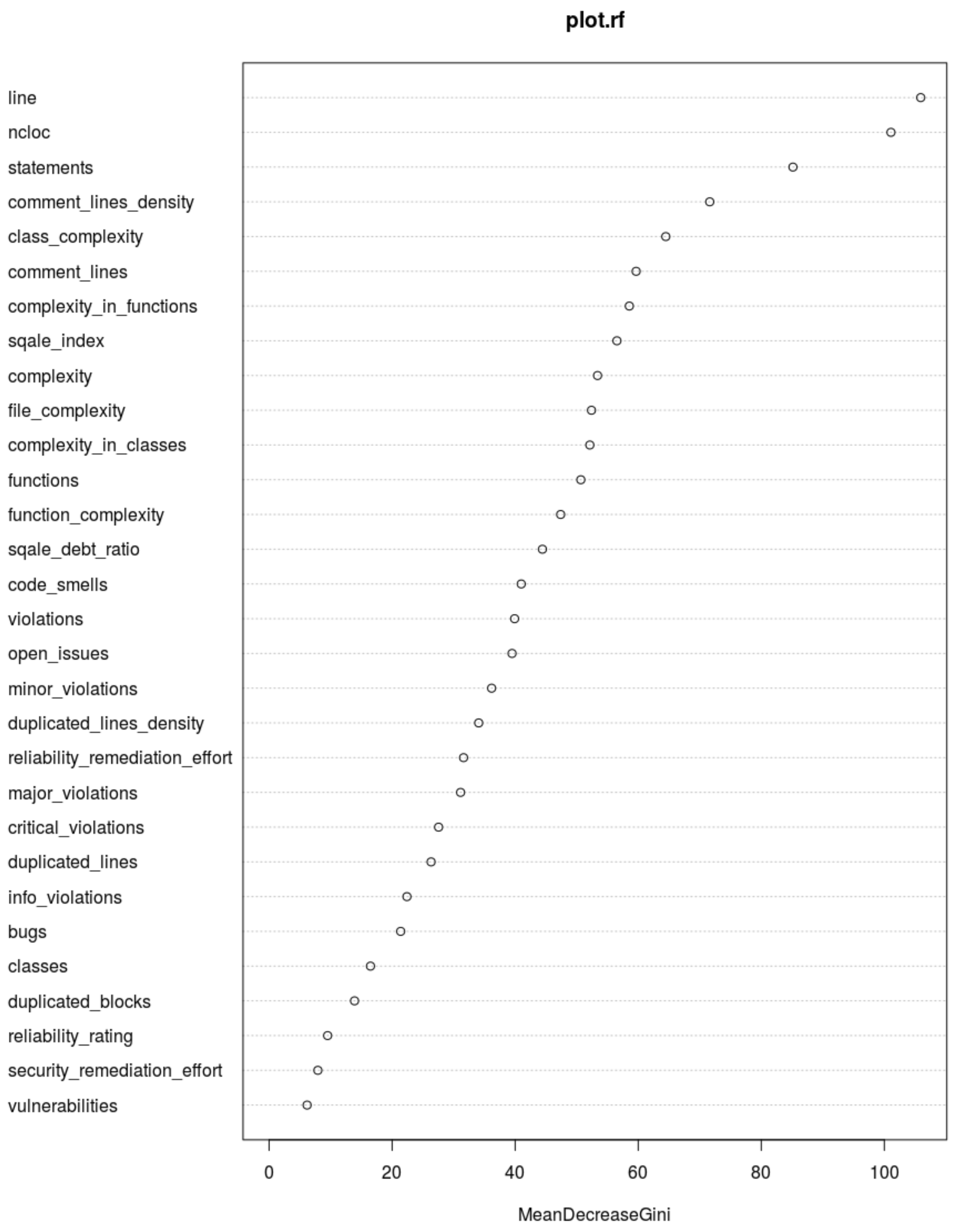


Fig. 2.2: Gini Index

2.2.2 Feedback Integration Model

The Ranking Model along with static ranking has a dynamic component. This dynamic component depends on the feedback submitted by the user which in this case is a code reviewer. This is the best way to leverage the expertise of the code reviewer. Also, since different reviewers have different styles of reviewing, a flexible Ranking Model would be more beneficial. The Ranking Model accepts user feedback by giving the user an option to up-vote or down-vote a change. When such a feedback is recorded from the user the Ranking Model dynamically changes the ranked list and rearranges the changes such that the changes similar to the one that the user has up-voted moves up the ranked list and if a user down-votes a change, all the changes similar to that change are moved down the ranked list. The dynamic weight for each change can be computed using the formula below.

$$\forall r \in R, w_d = a * A + b * B + ... + z * Z$$

In the above equation r is a revision in a set of ranked revisions R presented to the code reviewer. Each revision can be categorized based on different change information $A, B, C, ..., Z$ recorded by the Ranking Model. In this model these categories are used as factors that determine the similarity between changes. The value of these categories can either be a 1 or a 0 depending on whether two changes are similar or not. $a, b, ..., z$ are the importance factors of these categories.

The list of change information considered while building the Ranking Model are:

- Author Name : The author of the changes in the revision.
- Branch Name : The branch to which the changes were made.
- Type : The type of change that was made.

2.2.3 Prediction Model

The Prediction Model aids the code reviewer by giving more information about the change. Previous research has showed that one of the main reasons that a code reviewer has a hard time reviewing the code is when the reviewer has no prior understanding of the code to be reviewed [6]. The Prediction Model compliments the Ranking Model by tagging each change as important and normal. The Prediction Model has been trained to classify these changes. The Prediction Model can be trained to classify changes based on different strategies. The Prediction Model in this case is based on a specific labeling strategy. The labeling strategy is explained in the section below.

Labeling Strategy

Figure 2.3 gives an overview of the labeling strategy. The Prediction Model is built using a classification algorithm, Random Forest Tree, on a labeled dataset. So all the changes in the dataset had to be labeled as important or normal. One way of doing it would be to manually assess and label the changes. However, manual assessment can be vague unless based on strict parameters. In this case the dataset was labeled based on identification of issues and fixes applied to them. So all changes in the repository that had an issue associated with them or a fix to an issue were labeled as important and all other changes would be labeled as normal. The project management aspect of a repository of a project does a good job in keep tracking of issues and their resolutions. Issues are bugs but not limited to it. It can be performance problems, code smells, refactoring requirements. After the dataset has been labeled the Prediction Model tries to encapsulate the essence of the difference and provide future recommendations to the code reviewer.

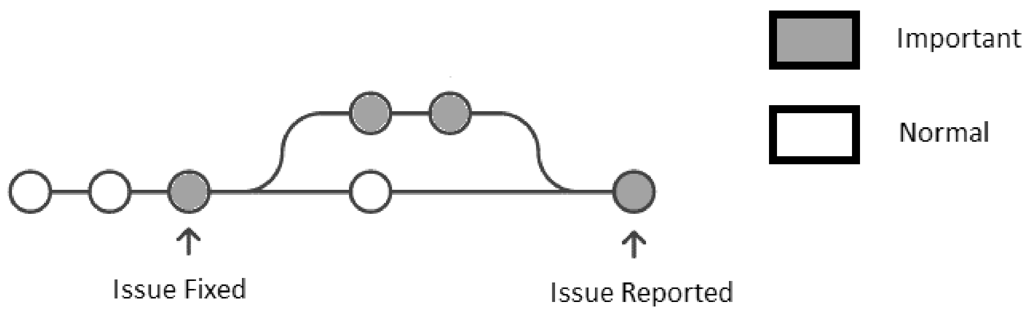


Fig. 2.3: Labeling Strategy

2.2.4 Refactoring Information Generator

This section presents Refactoring Information Generator consisting of the following two phases. Phase I summarizes clone refactorings using templates, which check the structural constraints before and after applying a refactoring to a program and encode ordering relationship between refactoring types. Phase II detects clone groups incompletely refactored, classifying the reasons.

2.2.5 Converting Clone to AST Model

The approach parses clone groups reported by Deckard [22] a clone detector. Refactoring Information Generator finds the set of AST nodes that contains reported clones:

$$n \mid offset(clone) \geq offset(statements)\ \Lambda$$

$$length(clone) \leq length(statements)\ \Lambda$$

$$n\ \in\ ast(statements)$$

In the above mathematical representation $offset(clone)$ is the starting position of clone instance clone from the beginning of the source file, and length(clone) is the length of the clone instance. The function ast(statements) finds an AST at the method level, and analyzes the AST to find inner-most syntactic statements (i.e., least common ancestor) that may contain incomplete syntactic regions of cloned code fragments. Refactoring information generator improves the performance of search by caching ASTs of syntactic clones after computing the above equation. The tool for parsing Java source code and generating the corresponding ASTs is provided with the Eclipse Java Development Tools framework.

2.2.6 Tracking Clone Histories

Refactoring Information Generator accesses each subsequent revision illustrated by $rj \in R = r_i + 1, r_i + 2, ..., r_n$, and identifies ASTs that are related to clone instances in an original revision r_i. Refactoring Information Generator presents a clone refactoring aware approach to check the clone change synchronization across revisions. It is integrated with Software Configuration Management (SCM) tools using an SCM library to analyze refactorings of individual clones and groups, helping developers focus their attention on any revision. To track the evolved clone group, Refactoring Information Generator automatically accesses the consecutive revisions ($n \geq 2$, where n is configurable) to search for the clone siblings (CI_1 and CI_2) by comparing two revisions. Then it extracts changes between two versions of ASTs. Refactoring Information Generator leverages the Change Distiller to compute AST edits to code clones. The Change Distiller was chosen since it represents concise edit operations between pairs of ASTs by identifying change types such as adding or deleting a method, inserting or removing a statement, or changing a method signature. It also provides fine-grained expression with a single statement edit. Refactoring Information Generator uses the Change Distiller to compute differences between the clone ASTs in revision r_i and the evolved clone ASTs in revision r_j.

When checking changes before and after clone refactorings, it is important to determine if references (e.g., method call and eld access) are preserved across clone instances. Therefore, a new reference binding checker was created, which is not provided by the Change Distiller, to assess reference consistency.

2.2.7 Matching Clone Refactoring Pattern Templates

Refactoring pattern templates are AST-based implementations that consist of a pair of pre- and post-edit matchers such as Mpre and Mpost. Mpre is an implementation for matching patterns before clone refactoring application. Mpost interacts with repositories, traverses ASTs of the source code in which the clones and their dependent contexts are modified, and extracts both a match between the nodes of the compared AST subtrees before and after a refactoring application and an edit script of tree edit operations transforming an original into a changed tree. After matching such clone refactoring patterns comprising a set of constraint descriptions where a refactoring can be performed, Refactoring Information Generator identifies concrete clone refactoring changes (e.g., re-factored revisions, refactoring types, locations, and restructuring descriptions). The change pattern descriptions are designed by using declarative rule-based constraints. A composite refactoring comprises a set of low-level refactorings.

2.2.8 Inconsistent Clone Refactoring

Refactoring Information Generator detects incomplete clone refactorings that are clone instances which are un-refactored inconsistently with other sibling in the group. It also classifies unrefactored clones if they are locally refactorable or not. Non-locally-refactorable clones means that a developer has difficulty performing refactorings to remove clones using standard refactoring techniques [23] due to limitations of a programming language or incomplete syntactic units of clones.

CHAPTER 3
DATA COLLECTION

Data collection is an important aspect of this research. The Ranking Model and the Prediction Model as described in sections 2.2.1 and 2.2.3 in chapter 2 are heavily data dependent. They depend on a selected list of attributes. This list of attributes was selected from an array of attributes and was used to train the Ranking Model and the Prediction Model. This chapter describes the approach used to acquire the dataset that contained all the initial attributes. Understanding the process of acquiring these attributes gives a better understanding of the ranking and the prediction model. The next section describes the approach used to generate the required dataset.

3.1 Approach

The Figure 3.1 shows an overview of the data collection approach used. The Change Analyzer and Metric Generator together generate an unlabeled dataset. The Change analyzer is used to generate the change information and the Metric Generator is used to generate the code quality metric information for all changes in each revision. The labeler is then used to label the unlabeled dataset. The Labeler works by analyzing the reported issues section of the project and labeling changes based on a predefined labeling strategy. The Change Analyzer, Metric Generator and the Labeler are further explained in the sections below.

3.2 Change Analyzer

The Change Analyzer is used to generate change information for changes in each revision. Every revision of a repository usually has some changes added by the developers. The Change Analyzer allows iterative analysis of the changes between consecutive revisions. It does this by keeping a record of all the changes, statistical information about these changes

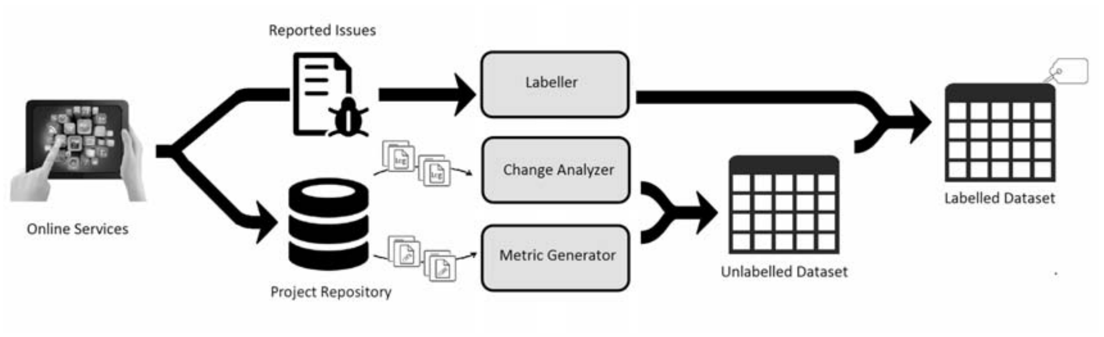


Fig. 3.1: Data Collection Approach Overview.

and linking each change to the previous and next changes. This gives an insight into the evolution of all the changes occurring throughout revisions. Statistical information include name of the developer making the revision, the message associated with the revision, date, type and the location of the change. The Change Analyzer also helps optimize the Metric Generator described in the next section. To compute all code quality metric values of all the files and their components for every revision is a huge overhead especially in large projects with numerous revisions. The Change analyzer helps the Metric Generator to focus on the changes made in every revision and avoids calculating metric data for files that were unchanged from the previous revision.

3.3 Metric Generator

The static code analysis of the project is done by the Metric Generator. The metric generator examines each revision and computes many code quality metrics at different granularities starting from the project as whole, going into file and class level and even drilling down into the method in individual classes. It also identifies known bugs, code smells and vulnerabilities at each revision and generates statistics on it. The Metric Generator inspects the overall health of the project. Thus with the help of the Change Analyzer it generates an unlabeled dataset. The unlabeled dataset on its own is pretty useful; however, this research leverages the advantages of supervised learning to train the Ranking Model and the Prediction Model. Hence, it becomes a necessity to label all the data points in the dataset. The Labeler, described in the next section helps in labeling the generated dataset.

3.4 Labeler

The Change Analyzer and Metric Generator described in the previous section help in generating an unlabeled dataset. However, to train the Prediction Model and Ranking Model we need a labeled dataset. There are many ways to label the dataset using different labeling strategies. The labeler classifies the changes as important or normal based on the labeling strategy described in section 2.2.3 in chapter 2. Thus the Labeler helps generate a labeled dataset.

3.5 Dataset

To summarize, the dataset consists of three major components: a set of change information of all changes in every revision generated by the Change Analyzer, a set of all metric values computed by the Metric Generator for all changes in each revision and finally a label for each of these changes obtained through the Labeler. In total the final dataset had a total of 151 attributes out of which 8 were change information attributes and 143 were code quality metric attributes labeled as important or normal.

CHAPTER 4
IMPLEMENTATION

To implement the approach described in the previous chapter the tools described below were built. Outside of this approach these tools individually or combined is of value as it can be used to explore new approaches which are different from the one used in this research. Different labeling and attribute selection strategies can be explored to generate customized ranking and prediction models. These tools are valuable for future work as well as aid in other studies in the same area. The next few sections describe the tools: GitLogParser, SonarGitAnalyzer and GitIssueParser.

4.1 GitLogParser

GitLogParser is the implementation of the Change Analyzer described in section 3.2 in the previous chapter. It has been implemented using JGit version 4.7.1. Eclipse JGit is a pure Java implementation of the Git version control system. GitLogParser can be used to get commit information, parent information and diff information from a project repository. Figure 4.1 gives us an understanding of the design of the tool. The terms commit and revision will be used interchangeably while describing the following sections. The GitLogParser analyses the project repository to give us the following change information:

- commit_sha: This is a 40 character SHA-1 hash which acts as a revision identifier. They mainly help keep track of all the revisions in the system, information related to the revision and relationship to other revisions.
- author_name: This is the identity of the author that made the revision. This is tied to the commit_sha and tells us which author was responsible for the revision and all the changes made in the respective revision.

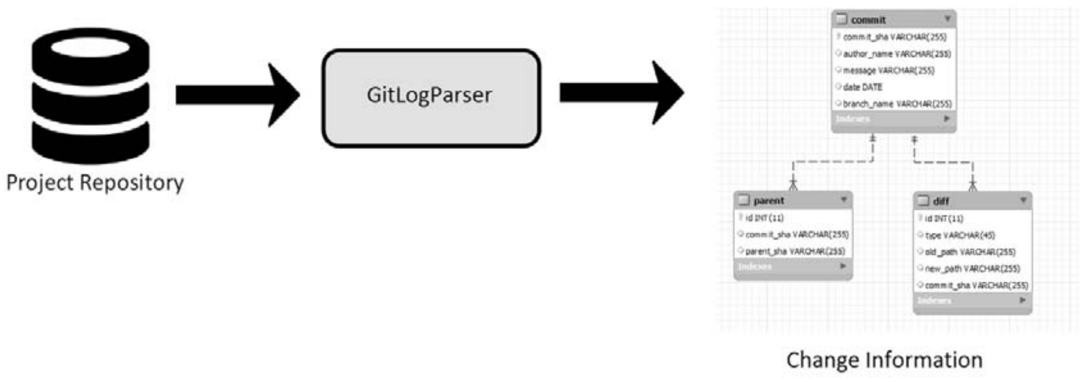


Fig. 4.1: GitLogParser Overview

- message: This is the message associated with a revision made by the author. A well written commit message gives great insight into the code. Previous research has proven that commit message can be used to perform analysis on developers and their coding practices [24].

- date: The date gives us the time stamp information about the revision made. This can help us analyze when the change was made or limit the analysis to a specific date range.

- branch_name: Many branches exists in a project repository, each serve their own purpose. It can be an important piece of information to have when analyzing the changes in a project repository. Thus, strategies can be developed based on specific branches of the repository.

- parent_sha: Like the commit_sha, it is also a 40 character revision identifier. This is helpful in figuring out the parent-child relationship of a revision which could be useful when tracking or analyzing the evolution of changes across revisions.

- type: This identifies the type of change that was made in the revision. This can be of three main types: an add, a delete and a modify. All changes in the revision are tracked at the file level. So a specific change in a revision can be classified as being a new file addition, modification of an existing file or deletion of a file.

- old_path: This gives the old path of the file that was changed in the revision.
- new_path: This gives the new path of the file that was changed in the revision. The old path and new path information in this research was used by the SonarGitAnalyzer described in the next section to generate code quality metric values for all changes.

4.2 SonarGitAnalyzer

SonarGitAnalyzer is the implementation of the Metric Generator described in section 3.3 in the previous chapter. It has been implemented as a wrapper to SonarQube as described in figure 4.2. SonarQube [25] (formerly Sonar) is an open source platform for continuous inspection of code quality. The SonarGitAnalyzer takes an input a list of commit_sha and cycles through each revision to generate code quality metric information for each of them. In essence it takes one commit_sha, automatically resets the project to the state it was in at the specific commit and computes the needed values before moving on to the next commit. This tool can be used to analyze past revisions of a project or can also be called every time a commit occurs to calculate and store code quality metric information in a Continuous Integration Environment. Implementation of the Metric Generator computes many measures but all of them can be grouped into one of the following categories [26]:

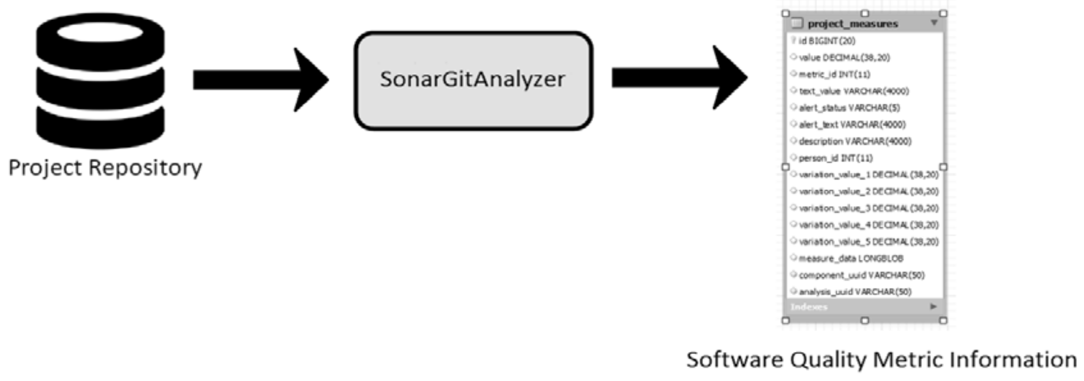

Project Repository
SonarGitAnalyzer
project_measures
id BIGINT(20)
value DECIMAL(38,20)
metric_id INT(11)
text_value VARCHAR(4000)
alert_status VARCHAR(5)
alert_text VARCHAR(4000)
description VARCHAR(4000)
person_id INT(11)
variation_value_1 DECIMAL(38,20)
variation_value_2 DECIMAL(38,20)
variation_value_3 DECIMAL(38,20)
variation_value_4 DECIMAL(38,20)
variation_value_5 DECIMAL(38,20)
measure_data LONGBLOB
component_uuid VARCHAR(50)
analysis_uuid VARCHAR(50)
Indexes
Software Quality Metric Information

Fig. 4.2: SonarGitAnalyzer Overview

- General: Consisting of all the general code quality metrics described below.

  - Classes: Number of classes (including nested classes, interfaces, enums and annotations) in the project.
  - Directories: Number of directories in the project.
  - Files: Number of files in the project.
  - Lines: Number of physical lines in the project.
  - Lines of code: Number of physical lines that contain at least one character which is neither a whitespace or a tabulation or part of a comment in the project.
  - Methods: Number of functions in the project. Depending on the language, a function is either a function or a method or a paragraph.
  - Projects: Number of projects in a view.
  - Public API: A sum of the number of public Classes, the number of public functions and the number of public properties in the project.
  - Statements: Number of statements in the project as a whole.

- Complexity: Consisting of all the code quality metrics that describe the complexity of code.
  - Complexity: Also known as cyclomatic complexity, it is the complexity calculated based on the number of paths through the code. Whenever the control flow of a function splits, the complexity counter gets incremented by one. Each function has a minimum complexity of 1.
  - Cognitive Complexity: It is a measure of how hard it is to understand the code's control flow visually. It complements the cyclomatic complexity and helps us gauge the effort required in maintaining an application.
  - Complexity /class: Average cyclomatic complexity by class.
  - Complexity /file: Average cyclomatic complexity by file.
  - Complexity /method: Average cyclomatic complexity by function.

- Documentation: Consisting of all the code quality metrics pertaining to the documentation of the code.

  - Comment lines: Number of lines containing either comment or commented-out code. These do not include non-significant comment lines such as: empty comment lines, comment lines containing only special characters, etc. which do not increase the number of comment lines.
  - Comments (%): This describes the percentage of comment lines in the project illustrated by the formula:

    $$Density\ of\ comment\ lines = \frac{Comment\ lines}{(Lines\ of\ code + Comment\ lines)} * 100$$

    With such a formula:

    * 50% means that the number of lines of code equals the number of comment lines.
    * 100% means that the file only contains comment lines.
  - Commented-out LOC: Specifically describes the number of commented-out lines of code as opposed to comment messages used purely for descriptive purpose.
  - Public undocumented API: Represents all Public API without comments header.
  - Public documented API (%): This describes the percentage of documented public API illustrated by the formula:

    $$Public\ documented\ API = \frac{(Public\ API - Public\ undocumented\ API)}{(Public\ API)} * 100$$

- Duplications: Consisting of all the code quality metrics pertaining to the duplication of the code.

  - Duplicated blocks: This describes the number of duplicated blocks of lines. There should be at least 10 successive and duplicated statements whatever the number of tokens and lines. Differences in indentation as well as in string literals are ignored while detecting duplications.

- Duplicated files: This describes the number of files involved in duplications.
- Duplicated lines: This describes the number of lines involved in duplications.
- Duplicated lines (%): This describes the percentage of duplicate lines in the project illustrated by the formula:

$$Density\ of\ duplication = \frac{Duplicated\ lines}{Lines} * 100$$

- Issues: Consists of all the issue related code quality metric information described below.
  - New issues: Describes the number of new issues detected in the project.
  - New xxxxx issues: Describes the number of new issues with severity xxxxx, xxxxx being blocker, critical, major, minor or info in the project.
  - Issues: Describes the number of existing issues.
  - xxxxx issues: Describes the number of existing issues with severity xxxxx, xxxxx being blocker, critical, major, minor or info in the project.
  - False positive issues: Describes the number of false positive issues.
  - Open issues: Describes the number of issues whose status is 'Open' in the project.
  - Confirmed issues: Describes the number of 'Confirmed' issues in the project.
  - Reopened issues: Describes the number of issues whose status is 'Reopened' in the project.
  - Statements: Describes the total number of statements in the module.
- Reliability: Consists of all the code quality metrics that give us an understanding of how robust the project is.
  - Bugs: Describes the number of bugs in the project.

  - New Bugs: Describes the number of new bugs in the project.
  - Reliability Rating: Describes a rating that indicates the reliability of the project.

    A = 0 Bug.

    B = at least 1 Minor Bug.

    C = at least 1 Major Bug.

    D = at least 1 Critical Bug.

    E = at least 1 Blocker Bug.
  - Reliability remediation effort: Describes the effort required to fix all bug issues. This code quality metric is measured in minutes.
  - Reliability remediation effort on new code: Same as Reliability remediation effort but is calculated based on the new code entered into the project.
  - Technical Debt: Describes the effort required to fix all bug issues in the project. The value of this metric is calculated in minutes.
  - Technical Debt on new code: Same as Technical Debt but is calculated based on the new code entered into the project.
  - Technical Debt Ratio: This can be described as the ratio between the cost to develop the software and the cost to fix it illustrated by the formula:

$$Technical\ Debt\ Ratio\ = \frac{Remediation\ cost}{Cost\ of\ one\ line\ of\ code\ *\ Total\ lines\ of\ code}$$

  - Technical Debt Ratio on new code: Describes the ratio between the cost to develop the code changed but is calculated based on the new code entered into the project.

- Maintainability: Consists of all the code quality metrics that gives us an understanding of the maintainability of the project.

  - Code Smells: Describes the number of code smells in the project.

  - New Code Smells: Describes the number of new code smells in the project.
  - Maintainability Rating (formerly SQALE Rating): Describes the rating given to the project. It is related to the value of the Technical Debt Ratio. The default Maintainability Rating grid is:

    A = 0-0.05

    B = 0.06-0.1

    C = 0.11-0.20

    D = 0.21-0.5

    E = 0.51-1

    The Maintainability Rating scale can be alternately stated by saying that if the outstanding remediation cost is:

    * less than 5% of the time that has already gone into the application, the rating is A.
    * between 6 to 10% the rating is a B.
    * between 11 to 20% the rating is a C.
    * between 21 to 50% the rating is a D.
    * anything over 50% is an E.

- Security: Consists of all the code quality metrics that gives us an understanding of the security of the project.

  - Vulnerabilities: Number of vulnerabilities.
  - New Vulnerabilities: Number of new vulnerabilities.
  - Security Rating:

    A = 0 Vulnerability.

    B = at least 1 Minor Vulnerability.

    C = at least 1 Major Vulnerability.

    D = at least 1 Critical Vulnerability.

    E = at least 1 Blocker Vulnerability.

- Security remediation effort: Describes the effort to fix all vulnerability issues. The measure is stored in minutes in the database.
- Security remediation effort on new code: Same as Security remediation effort but computed for the new code entered into the project.

4.3 GitIssueParser

GitIssueParser is the implementation of the Labeler described in section 3.4. The GitIssueParser is implemented using the Github REST API v3. This tool is built to leverage the reporting of issues of an open source project on Github. Good open source projects have a community of people testing and reporting issues which are reviewed, verified and fixed. Many of these issues are labeled and classified into predefined categorizes. The figure 4.3 gives us an overall understanding of the methodology of this tool. The GitIssuesParser gives us the following information:

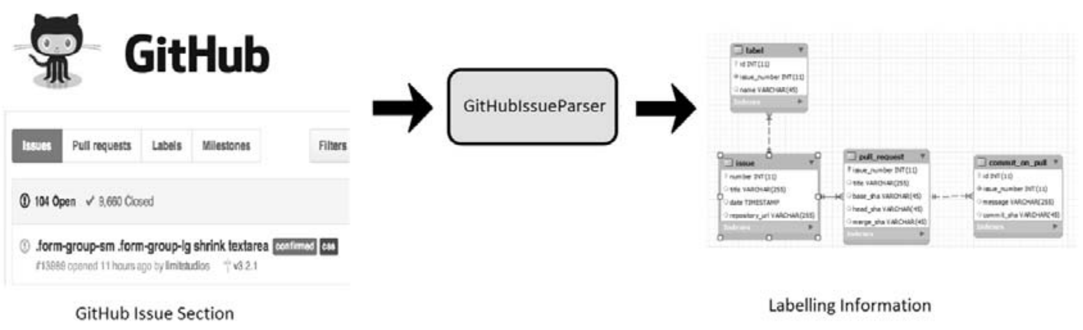


Fig. 4.3: GitHubIssuesParser Overview

- issue_number: This is the identifier of the reported issue. It is used to keep track of the updates and changes to this issue in the repository.
- name : This describes label associated with the issue. Depending on the project the person in charge of reviewing open issues assigns it a predefined label. This kind of classification or labeling information can be leveraged. For example, if an issue is labeled as a bug then all issues and code changes related to other issues with the same

labels can be analyzed. Also, code changes pertaining to two different issues can be compared and analyzed.

- title: The title is a short description of the identified issue.
- date: The date describes when the issue was reported.
- base_sha: This gives the identifying information about the base commit of the pull request that was made as an attempt to fix the issue.
- head_sha: This gives the identifying information about the head commit of the pull request that was made as an attempt to fix the issue.
- merge_sha: This gives the identifying information about the merge commit of the pull request that was made, accepted and merged after the pull request was made.
- commit_sha: This gives the commit number of the change that was made.

CHAPTER 5
EVALUATION

For assessing the effectiveness of this research the Ranking Model, Prediction Model and Refactoring Information Generator was evaluated. The experimental setup and design for each of the following is described in the sections below. All experiments were conducted on a machine with a quad core 2.2 GHz CPU and 16 GB RAM.

5.1 Ranking Model

The Ranking Model was evaluated on the WordPress-Android project. The objective of the experiment was to prove that the Ranking Model can prioritize and rank important changes from a code review perspective. To achieve this, the rank assigned by the Ranking Model was compared to the one assigned through manual inspection. A set of revisions of the project was passed through the Ranking Model to obtain the rank for each revision. The same set was then manually evaluated and ranked by a Sr. Software Engineer with considerable amount of industry and code review experience. Table 5.1 shows the top 10 ranked revisions along with the weight assigned to the revision used by the Ranking Model to calculate the rank. The table also includes the column 'R1' which represents the rank assigned by the Ranking Model and 'R2' which represents the rank assigned by the reviewer. The column 'd' is the difference in rank and 'd^2' is the square of the difference in the rank which is used to compute the similarity between the ranks. The similarity between the ranked list generated by the Ranking Model and the ranked list compiled through manual inspection is calculated using Spearman's Rank-Order Correlation. Spearman's Rank-Order Correlation indicates the degree of association between ranks and can have values between -1 and 1 where -1 indicates no association and 1 indicates perfect association. It can be computed using the formula:

$$\rho = 1 - \frac{6 \sum d_i^2}{n(n^2 - 1)}$$

In the above equation ρ is the Spearman's Rank-Order Correlation, n is the number of items ranked and d_i^2 indicates the square of the difference in the rank of item i. In this case the Spearman's Rank-Order Correlation value was calculated as 0.67 which indicates a strong co-relation between both the ranks.

Table 5.1: Ranking Model Evaluation Result

Commit_Sha	Weight	Label	R1	R2	d	d^2
d4c253fde6fb9023bc15cb1c5d54478da2133fd2	225.73	important	1	1	0	0
b6e0001da958f1631f9b963dea4529f45dOf104a	225.14	normal	2	3	1	1
c4e6Uc674d27a1ddd0978175781c9c30ba4cU7	225.14	normal	3	2	1	1
2ddac7e708d7e9f37822dcec8609b5d8547a61ce	185.72	normal	4	7	3	9
bfe92ebed52f895fa485a57f1f226e5d282d2276	176.30	normal	5	9	4	16
c321149d5daf27e25aOS4cS63e1cf9f1	147.64	normal	6	5	1	1
8c98a329d1b57985fcbbbe5Oec987670a3aed8	144.08	normal	7	6	1	1
93dc21921f35d8861bdcfcd12b675250080c3a	143.23	normal	8	8	0	0
5b02c22cd816266594dfb786dc8120b673abfcc4	140.34	important	9	4	5	25
513ega62eeObe3d59ffdea8ba16e5619f06cUOc	140.76	normal	10	10	0	0

5.2 Prediction Model

The Prediction Model was also evaluated on the WordPress-Android project. The WEKA data mining software [27] was used to conduct the evaluation for the Prediction Model. All the changes in revisions of the last four months of 2016 of the WordPress-Android project were considered for the evaluation. The labeled dataset was generated using the data collection strategy described in chapter 3. All changes that were not Java in nature were excluded since this tool currently only supports analysis of files that are Java in nature. As illustrated in table 5.2, there were a total of 11029 classified instances.

Table 5.2: Prediction Model Evaluation Result

Correctly Classified Instances	7300/11029 (66.1891 %)
Incorrectly Classified Instances	3729/11029 (33.8109 %)
Mean absolute error	0.4356
Root mean squared error	0.462

Out of all the classified instances 7300 were correctly classified as normal or important changes. A total of 3729 instances were incorrectly classified. Hence, the Prediction Model was evaluated to have an accuracy of 66.18%. To give us an understanding of how close the predictions were to the actual ground truth and as another way of measuring the accuracy of the model we calculated the Mean Absolute Error which was 0.4356. Also to understand the deviation between the predicted values and the actual observed values, the Root Mean Squared Error was calculated. The Root Mean Squared Error of the Prediction Model was calculated at 0.462. The model is not highly accurate but there is scope for improvement by fine tuning the labeling strategy and attribute selection process described in chapter 2. However, as a proof of concept the evaluation result of the Prediction Model looks very promising.

5.3 Refactoring Information Generator

This evaluation included in the thesis is part of the research conducted, submitted and accepted to COMPSAC 2017 [28]. For assessing Refactoring Information Generator's effectiveness, two studies were performed. First was the assessment of the summarization of clone refactorings and applicability in real scenarios using six open source projects: AlgoUML, Apache Tomcat, Apache Log4j, Eclipse AspectJ, JEdit and JRuby, by mining clone refactorings from their repositories. These projects were selected for two main reasons. First, all subject applications are written in Java, which is one of the most popular programming languages. Second, these applications are under active development and are based on a collaborative work with at least 48 months of active change history. In the second study, a data set with real refactoring anomalies, incomplete clone refactorings was used to investigate its detection capability. Clone groups and their changes where real developers applied clone refactorings in repositories were manually examined to generate a ground truth data set. First commit logs were parsed to a bag-of-words and stemmed using a standard Natural Language Processing (NPL) tool. Then the keyword matching was used to find these words throughout the revisions of the project. Then all the change files in these revisions were manually inspected to find clone refactorings and clone refactoring anomalies. This dataset was compared to the results.

5.3.1 Clone Refactoring Summarization Evaluation

Table 5.3 illustrates the evaluation results for clone refactoring summarization. Using the ground truth data set G1, precision P1 and recall R1 are calculated as $P1 = \frac{|G1 \cap S|}{|S|}$, $R1 = \frac{|G1 \cap S|}{|G1|}$, where P1 is the percentage of the summarization results that are correct, R1 is the percentage of correct summarization that Refactoring Information Generator reports, and S denotes the clone groups identified by Refactoring Information Generator, all clone instances of which are refactored. Refactoring Information Generator's precision was assessed by examining how many of refactorings of clone groups are indeed true clone refactoring. Refactoring Information Generator summarizes 94 refactorings of clone groups, 92 of which are correct, resulting in 98.9% precision. Regarding recall, Refactoring Information Generator identifies 92% of all ground truth data sets. It identifies 92 out of 98 refactored groups. It summarizes refactorings of clone groups, tracking the clone histories with 94.1% accuracy. Refactoring Information Generator summarizes refactorings of clone groups that are not easy to identify because they require investigating refactorings of individual versions of a program while tracking changes of a clone group. Instead of tracking each version incrementally, simply comparing with the latest version can produce every change to be inspected. However, composite code changes, which inter-mingle multiple development issues together, are commonly difficult to conduct a code review.

5.3.2 Incomplete Refactoring Detection Evaluation Result

Table 5.4 summarizes the evaluation results for Incomplete Refactoring Detection. Using the ground truth data set G2, precision P2 and recall R2 are calculated as $P2 = \frac{|G2 \cap D|}{|D|}$,

Table 5.3: Clone Refactoring Summarization Evaluation Result

ID	RFT	VER	TIM	CLS	GTS	P	R	A
1	EM	3	1.1	2 / 1	2 / 1	100	100	100
2	PU	3	0.1	5 / 2	6 / 3	100	66.7	80.0
3	PU	4	0.4	6 / 1	7 / 2	100	50.0	66.7
4	ES	9	0.9	20 / 4	20 / 4	100	100	100
5	ES	4	0.4	2 / 1	9 / 3	100	33.3	50.0
6	ES	3	0.8	17 / 7	17 / 7	100	100	100
7	ES	3	0.6	48 / 24	48 / 24	100	100	100
8	EM+PU	7	1.0	3 / 1	8 / 2	100	50.0	66.7
9	EM	3	0.8	4 / 2	4 / 2	100	100	100
10	EM	9	0.9	8 / 1	8 / 1	100	100	100
11	PU	4	0.7	9 / 3	9 / 3	100	100	100
12	PU	3	0.5	2 / 1	2 / 1	100	100	100
13	PU	3	1.1	2 / 1	2 / 1	100	100	100
14	PU	3	0.3	2 / 1	2 / 1	100	100	100
15	PU	3	0.4	2 / 1	2 / 1	100	100	100
16	ES	3	2.3	40 / 20	40 / 20	100	100	100
17	PU+EM	3	2.2	6 / 3	6 / 3	100	100	100
18	ES	3	1.4	11 / 5	12 / 6	71.4	83.3	76.9
19	EM+PU	3	1.5	2 / 1	2 / 1	100	100	100
20	EM	3	0.6	2 / 1	2 / 1	100	100	100
21	MN	3	2.4	2 / 1	2 / 1	100	100	100
22	PU	3	0.1	2 / 1	2 / 1	100	100	100
23	PU	3	0.5	2 / 1	2 / 1	100	100	100
24	EM	3	0.6	2 / 1	2 / 1	100	100	100
25	EM	3	0.6	2 / 1	2 / 1	100	100	100
26	ES	4	0.7	6 / 3	6 / 3	100	100	100
27	EM+MM	5	1.3	8 / 3	8 / 3	100	100	100
	Total or Avg	103	0.9	217 / 92	232 / 98	98.9	92.0	94.1

RFT: the refactoring types that developers perform across revisions, VER: the number of revisions where developers apply refactorings, TIM: the time that Refactoring Information Generator completes each task (an average of time (sec.) per group), CLs: the number of clones correctly summarized by Refactoring Information Generator (instance/group), GTs: the number of the ground truth data set for clone refactoring summarization (instance/group), P: precision (%), R: recall (%), and A: accuracy (%).

$R2 = \frac{|G2 \cap D|}{|G2|}$, where P2 is the percentage of the detection results that are correct, R2 is the percentage of correct detection results that Refactoring Information Generator reports, and D indicates the clone groups detected by Refactoring Information Generator, some clone instances of which are not refactored. The precision of Refactoring Information Generator was estimated by evaluating how many of the unrefactored clones are indeed a true omission of refactoring. As Refactoring Information Generator detects clone groups in which some clone instances are omitted from refactorings either intentionally or unintentionally, it is considered that any instance resulting in refactoring deviations of other refactored siblings in the group as a true clone refactoring anomaly. Refactoring Information Generator detects 1,162 unrefactored groups, 1,152 of which are correct, resulting in 99.4% precision. Regarding recall, Refactoring Information Generator detects 97.8% of all ground truth data sets. It detects 1,152 out of 1,164 unrefactored groups with clone refactoring classification with 98.4% accuracy. Refactoring Information Generator helps developers investigate unrefactored clone instances and understand how these instances are diverged from other clone siblings. It automatically classifies whether unrefactored clone instances cannot be easily removed by standard refactoring techniques, which is not easy to determine since understanding clone differences between clones is difficult.

5.4 User Study

Every code reviewer has his own style of reviewing the code since the review is a task involving many human factors [23]. Recommendations from the Recommendation Engine are meant to make the job easier for the code reviewer. Thus, a good way to evaluate the recommendation engine is through a user study. User study is meant to capture the user experience by having a product, in this case the code review tool, tested by users. A user study was conducted with 5 users and a high fidelity prototype of the code review tool. The users participating in the user study have a combined experience of over 25 years in the software development industry and regularly take part in the code review process as part of their work. The objective, hypothesis, methodology and result of the user study are explained in detail below:

Table 5.4: Incomplete Refactoring Detection Evaluation Result

ID	RFT	VER	TIM	CLD	GTD	P	R	A
1	EM	3	1.1	48 / 3	48 / 3	100	100	100
2	PU	3	0.1	123 / 52	123 / 52	98.1	100	99.0
3	PU	4	0.4	102 / 38	105 / 39	100	97.4	98.7
4	ES	9	0.9	449 / 36	449 / 36	100	100	100
5	ES	4	0.4	318 / 44	322 / 45	95.7	97.8	96.7
6	ES	3	0.8	1,430 / 86	1,430 / 86	98.9	100	99.4
7	ES	3	0.6	1,118 / 99	1,118 / 99	100	100	100
8	EM+PU	7	1.0	237 / 50	237 / 50	92.6	100	98.0
9	EM	3	0.8	20 / 7	20 / 7	100	100	100
10	EM	9	0.9	2,643 / 103	2,643 / 103	100	100	100
11	PU	4	0.7	2,228 / 218	2,306 / 221	99.1	98.6	98.9
12	PU	3	0.5	127 / 52	133 / 55	100	94.5	97.2
13	PU	3	1.1	8 / 3	12 / 5	100	60.0	75.0
14	PU	3	0.3	222 / 66	222 / 66	100	100	100
15	PU	3	0.4	171 / 57	171 / 57	100	100	100
16	ES	3	2.3	2,597 / 64	2,597 / 64	100	100	100
17	PU+EM	3	2.2	1,218 / 42	1,218 / 42	100	100	100
18	ES	3	1.4	1,210 / 41	1,212 / 42	95.3	97.6	96.5
19	EM+PU	3	1.5	2 / 1	2 / 1	100	100	100
20	EM	3	0.6	50 / 15	50 / 15	100	100	100
21	MN	3	2.4	2 / 1	2 / 1	100	100	100
22	PU	3	0.1	29 / 12	29 / 12	100	100	100
23	PU	3	0.5	97 / 32	99 / 33	100	100	100
24	EM	3	0.6	23 / 11	23 / 11	100	97.0	98.5
25	EM	3	0.6	15 / 7	15 / 7	100	100	100
26	ES	4	0.7	0 / 0	0 / 0	-	-	-
27	EM+MM	5	1.3	24 / 12	24 / 12	100	100	100
	Total or Avg	103	0.9	14,511 / 1,152	14,610 / 1,164	99.4	97.8	98.4

RFT: the refactoring types that developers perform across revisions, VER: the number of revisions where developers apply refactorings, TIM: the time that Refactoring Information Generator completes each task (an average of time (sec.) per group), CLd: the number of clones correctly detected by Refactoring Information Generator (instance/group), GTd: the number of the ground truth data set for incomplete clone refactoring detection (instance/group), P: precision (%), R: recall (%), and A: accuracy (%).

Objective

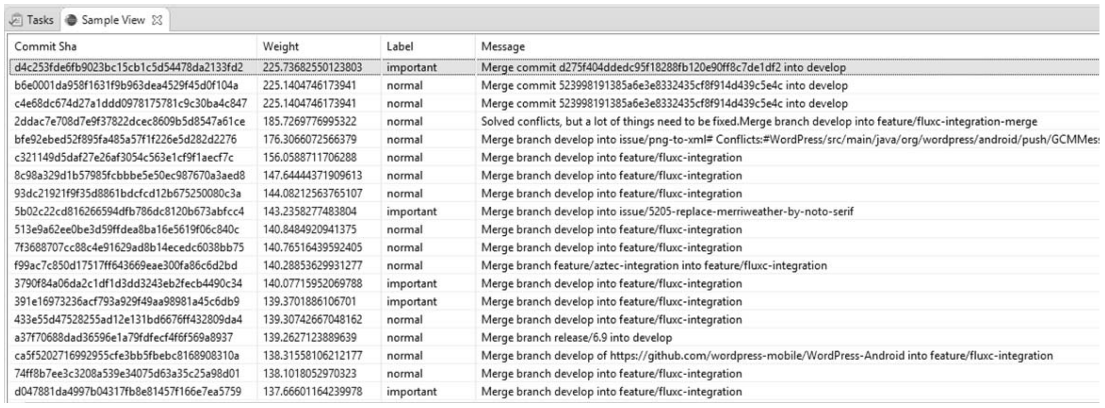
As discussed in chapter 1, code review is a process conducted by the code reviewer. This involves reviewing code changes between revisions. There are many methodologies to perform a code review like: Perspective-Based Reading (PBR) [29] where reviewers read code from several different viewpoints; Checklist-Based Reading (CBR) [30] where reviewers use a predefined checklist to review code; Usage-Based Reading (UBR) [31] where reviewers review code based on the users view point; Defect-Based Reading (DBR) [32] where reviewers focus on defect detection. In this study the participants are asked to review code without any reading criteria making this an Ad-Hoc review. The main objective of this user study is to evaluate the Recommendation Engine and its impact while performing an Ad-Hoc code review. Based on this objective a hypothesis was formed and is explained below.

Hypothesis

The code review tool, with the Recommendation Engine was built to have a positive impact on the code review process. According to the recent research conducted developers would rather postpone a review than rush through them, ignore review of issues they do not know enough about or stay away from issues that they think are uninteresting [6].The code review tool was built primarily to tackle this problem. Hence the hypothesis is that the code review tool optimizes the code review process by:

- Ranking the revisions and the changes in the revision
- Dynamically changing the rank based on the feedback from the reviewer
- Labeling each change in the revision as important or normal
- Providing additional refactoring information

To test this hypothesis a high fidelity prototype of a code review tool was built in eclipse as show in figure 5.1 and a user study was conducted. The methodology used to conduct the user study is explained below.



Tasks | Sample View

Commit Sha	Weight	Label	Message
d4c253fde6fb9023bc15cb1c5d54478da2133fd2	225.73682550123803	important	Merge commit d275f404ddedc95f18288fb120e90ff8c7de1df2 into develop
b6e0001da958f1631f9b963dea4529f45d0f104a	225.1404746173941	normal	Merge commit 523998191385a6e3e8332435cf8f914d439c5e4c into develop
c4e68dc674d27a1ddd0978175781c9c30ba4c847	225.1404746173941	normal	Merge commit 523998191385a6e3e8332435cf8f914d439c5e4c into develop
2ddac7e708d7e9f37822dcec8609b5d8547a61ce	185.7269776995322	normal	Solved conflicts, but a lot of things need to be fixed.Merge branch develop into feature/fluxc-integration-merge
bfe92ebed52f895fa485a57f1f226e5d282d2276	176.3066072566379	normal	Merge branch develop into issue/png-to-xml# Conflicts:#WordPress/src/main/java/org/wordpress/android/push/GCMMes
c321149d5daf27e26af3054c563e1cf9f1aecf7c	156.0588711706288	normal	Merge branch develop into feature/fluxc-integration
8c98a329d1b57985fcbbbe5e50ec987670a3aed8	147.64444371909613	normal	Merge branch develop into feature/fluxc-integration
93dc21921f9f35d8861bdcfcd12b675250080c3a	144.08212563765107	normal	Merge branch develop into feature/fluxc-integration
5b02c22cd816266594dfb786dc8120b673abfcc4	143.2358277483804	important	Merge branch develop into issue/5205-replace-merriweather-by-noto-serif
513e9a62ee0be3d59ffdea8ba16e5619f06c840c	140.8484920941375	normal	Merge branch develop into feature/fluxc-integration
7f3688707cc88c4e91629ad8b14ecedc6038bb75	140.76516439592405	normal	Merge branch develop into feature/fluxc-integration
f99ac7c850d17517ff643669eae300fa86c6d2bd	140.28853629931277	normal	Merge branch feature/aztec-integration into feature/fluxc-integration
3790f84a06da2c1df1d3dd3243eb2fecb4490c34	140.07715952069788	important	Merge branch develop into feature/fluxc-integration
391e16973236acf793a929f49aa98981a45c6db9	139.3701886106701	important	Merge branch develop into feature/fluxc-integration
433e55d47528255ad12e131bd6676ff432809da4	139.30742667048162	normal	Merge branch develop into feature/fluxc-integration
a37f70688dad36596e1a79fdfecf4f6f569a8937	139.2627123889639	normal	Merge branch release/6.9 into develop
ca5f5202716992955cfe3bb5fbebc8168908310a	138.31558106212177	normal	Merge branch develop of https://github.com/wordpress-mobile/WordPress-Android into feature/fluxc-integration
74ff8b7ee3c3208a539e34075d63a35c25a98d01	138.1018052970323	normal	Merge branch develop into feature/fluxc-integration
d047881da4997b04317fb8e81457f166e7ea5759	137.66601164239978	important	Merge branch develop into feature/fluxc-integration

Fig. 5.1: High Fidelity Prototype

Methodology:

There are many methodologies that can be used to test a hypothesis in a user study which includes user interviews, surveys and usability testing. The goal of this study was to see if the code review tool makes a positive impact on the code review process. Usability testing is the best way to evaluate this. Usability testing is a methodology where a user is given a set of instruction to use a product and the researchers observe and analyze the observation to come to a conclusion. A user group of 5 people were chosen to be part of the user study to perform the usability testing in a think out loud process. All of them together have over 25 years of experience in Software Development and regularly participate in the process of code review as a reviewer. This makes them ideal for testing the code review tool. A total of 5 sessions were conducted, one session with each of the user. The total session lasted for about 30 minutes. The following instructions were given to them:

1. Open Eclipse with the installed plug-in.

2. Enter the revision range.

3. Review the first revision on the list.

4. Give feedback to the tool after reviewing the revision.

5. Repeat step 3 and 4 till all changes have been reviewed.

During the process the participants were encouraged to think out loud and their thoughts were recorded and later analyzed.

Result

Each session generated a lot of interest and feedback from the participants. All of which were analyzed and the following can be concluded. Ranking the revisions and the changes in the revision helps the user prioritize the task to work on. The general feedback was that the users had a sense of urgency while tackling the first review in the list and would not delay the review for later. Dynamically changing the rank based on the feedback from the reviewer helped the reviewer work on issues they wanted to work on. This gave them a feeling of working on issues they had a preference for. Labeling each change in the revision as important or normal was helpful but the common consensus was the labeling had to be more fine grained for it to make a greater impact. Providing additional refactoring information was very well received as the common opinion was that these kinds of issues are generally overlooked during a review and thus greater impact could be made to the evolution of the software with a smaller investment of time. All in all the result of the usability test supports our hypothesis that the code review tool has a positive impact on the code review process.

CHAPTER 6
CONCLUSION

The goal of this research was to optimize the code review process and make it more efficient for the code reviewer. As discussed in the previous chapters, code review is a tedious and time consuming process. There are many reasons for that like reviewers not wanting to rush through reviews. Also code review process is difficult because reviewers need to invest time to learn about the code they are about to review. Each reviewer has his/her own preferences and thus they may be more inclined to avoid issues they think are uninteresting [6]. To optimize the code review process, this problem had to be solved. The end result of this research is a code review tool that provides an array of recommendations to the code reviewer. The tool is successfully able to provide a ranked list of revisions for a reviewer to review. The ranking strategy has a high correlation of 0.67 to that of a list of revisions manually ranked by a code reviewer with over 10 years of industrial experience. A ranked list of revisions helps the code reviewer assess the situation and plan accordingly. The tool also labels the generated change by using a Prediction Model that classifies changes into important or normal with an accuracy of 66.18%. This provides more information about revision, making the code reviewer comfortable while entering a code review. The tool also detects refactoring type with a precision of 98.8%, a recall of 92% and an accuracy of 94.1%. Furthermore, the tool is able to detect incomplete refactorings with a precision of 99.4%, a recall of 97.8% and an accuracy of 98.4%. Also, the tool adapts its recommendation based on the feedback from the user. Thus each code reviewer can have a customizable experience while reviewing code. Lastly, a user study was conducted and the result of the user study clearly indicated that the code review tool had a positive impact on the code review process. Hence, in this research a new approach for optimizing code review was conceived and evaluated with promising results.

REFERENCES

[1] H. A. Nguyen, A. T. Nguyen, T. T. Nguyen, T. N. Nguyen, and H. Rajan, "A study of repetitiveness of code changes in software evolution," in *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering.* IEEE Press, 2013, pp. 180–190.

[2] A. Bacchelli and C. Bird, "Expectations, outcomes, and challenges of modern code review," in *Proceedings of the 2013 international conference on software engineering.* IEEE Press, 2013, pp. 712–721.

[3] S. McIntosh, Y. Kamei, B. Adams, and A. E. Hassan, "The impact of code review coverage and code review participation on software quality: A case study of the qt, vtk, and itk projects," in *Proceedings of the 11th Working Conference on Mining Software Repositories.* ACM, 2014, pp. 192–201.

[4] T. Barik, K. Lubick, and E. Murphy-Hill, "Commit bubbles," in *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, vol. 2. IEEE, 2015, pp. 631–634.

[5] P. Thongtanunam, C. Tantithamthavorn, R. G. Kula, N. Yoshida, H. Iida, and K.-i. Matsumoto, "Who should review my code? a file location-based code-reviewer recommendation approach for modern code review," in *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on.* IEEE, 2015, pp. 141–150.

[6] P. C. Rigby and M.-A. Storey, "Understanding broadcast based peer review on open source software projects," in *Proceedings of the 33rd International Conference on Software Engineering.* ACM, 2011, pp. 541–550.

[7] J. Krinke, "Is cloned code more stable than non-cloned code?" in *Source Code Analysis and Manipulation, 2008 Eighth IEEE International Working Conference on.* IEEE, 2008, pp. 57–66.

[8] ——, "Is cloned code older than non-cloned code?" in *Proceedings of the 5th International Workshop on Software Clones.* ACM, 2011, pp. 28–33.

[9] N. Gode and J. Harder, "Clone stability," in *Software Maintenance and Reengineering (CSMR), 2011 15th European Conference on.* IEEE, 2011, pp. 65–74.

[10] L. Jiang, Z. Su, and E. Chiu, "Context-based detection of clone-related bugs," in *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering.* ACM, 2007, pp. 55–64.

[11] L. Barbour, F. Khomh, and Y. Zou, "Late propagation in software clones," in *Software Maintenance (ICSM), 2011 27th IEEE International Conference on.* IEEE, 2011, pp. 273–282.

[12] M. Mondal, C. K. Roy, M. S. Rahman, R. K. Saha, J. Krinke, and K. A. Schneider, "Comparative stability of cloned and non-cloned code: An empirical study," in *Proceedings of the 27th Annual ACM Symposium on Applied Computing.* ACM, 2012, pp. 1227–1234.

[13] L. Barbour, F. Khomh, and Y. Zou, "Late propagation in software clones," in *Software Maintenance (ICSM), 2011 27th IEEE International Conference on.* IEEE, 2011, pp. 273–282.

[14] M. Toomim, A. Begel, and S. L. Graham, "Managing duplicated code with linked editing," in *Visual Languages and Human Centric Computing, 2004 IEEE Symposium on.* IEEE, 2004, pp. 173–180.

[15] E. Duala-Ekoko and M. P. Robillard, "Tracking code clones in evolving software," in *Proceedings of the 29th international conference on Software Engineering.* IEEE Computer Society, 2007, pp. 158–167.

[16] D. Hou, P. Jablonski, and F. Jacob, "Cnp: Towards an environment for the proactive management of copy-and-paste programming," in *Program Comprehension, 2009. ICPC'09. IEEE 17th International Conference on.* IEEE, 2009, pp. 238–242.

[17] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen, "Clone-aware configuration management," in *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering.* IEEE Computer Society, 2009, pp. 123–134.

[18] Y. Lin, Z. Xing, Y. Xue, Y. Liu, X. Peng, J. Sun, and W. Zhao, "Detecting differences across multiple instances of code clones," in *Proceedings of the 36th International Conference on Software Engineering.* ACM, 2014, pp. 164–174.

[19] M. Vakilian, N. Chen, S. Negara, B. A. Rajkumar, B. P. Bailey, and R. E. Johnson, "Use, disuse, and misuse of automated refactorings," in *Proceedings of the 34th International Conference on Software Engineering.* IEEE Press, 2012, pp. 233–243.

[20] M. Kim, D. Cai, and S. Kim, "An empirical investigation into the role of api-level refactorings during software evolution," in *Proceedings of the 33rd International Conference on Software Engineering.* ACM, 2011, pp. 151–160.

[21] J. Park, M. Kim, B. Ray, and D.-H. Bae, "An empirical study of supplementary bug fixes," in *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories.* IEEE Press, 2012, pp. 40–49.

[22] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "Deckard: Scalable and accurate tree-based detection of code clones," in *Proceedings of the 29th international conference on Software Engineering.* IEEE Computer Society, 2007, pp. 96–105.

[23] T. Thelin, C. Andersson, P. Runeson, and N. Dzamashvili-Fogelstrom, “A replicated experiment of usage-based and checklist-based reading,” in *Software Metrics, 2004. Proceedings. 10th International Symposium on.* IEEE, 2004, pp. 246–256.

[24] E. Guzman, D. Azócar, and Y. Li, “Sentiment analysis of commit comments in github: an empirical study,” in *Proceedings of the 11th Working Conference on Mining Software Repositories.* ACM, 2014, pp. 352–355.

[25] G. Campbell and P. P. Papapetrou, *SonarQube in action.* Manning Publications Co., 2013.

[26] “Documentation - SonarQube Documentation.” [Online]. Available: https://docs.sonarqube.org/display/SONAR/Documentation

[27] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, “The weka data mining software: an update,” *ACM SIGKDD explorations newsletter*, vol. 11, no. 1, pp. 10–18, 2009.

[28] Z. Chen, “Tool support for managing clone refactorings to facilitate code review in evolving software,” 2017.

[29] F. Shull, I. Rus, and V. Basili, “How perspective-based reading can improve requirements inspections,” *Computer*, vol. 33, no. 7, pp. 73–79, 2000.

[30] M. Fagan, “Design and code inspections to reduce errors in program development,” in *Software pioneers.* Springer, 2002, pp. 575–607.

[31] T. Thelin, P. Runeson, and B. Regnell, “Usage-based readingan experiment to guide reviewers with use cases,” *Information and Software Technology*, vol. 43, no. 15, pp. 925–938, 2001.

[32] A. A. Porter, L. G. Votta, and V. R. Basili, “Comparing detection methods for software requirements inspections: A replicated experiment,” *IEEE Transactions on software Engineering*, vol. 21, no. 6, pp. 563–575, 1995.