2019 19th International Working Conference on Source Code Analysis and Manipulation (SCAM)

Interactive Refactoring Documentation Bot

Soumaya Rebai, Oussama Ben Sghaier, Vahid Alizadeh, Marouane Kessentini and Meriem Chater
CIS Department
University of Michigan
Dearborn, Michigan, USA
E-mail: {srebal, oussama, alizadeh, marouane, chater} @umich.edu

***Abstract*—The documentation of code changes is significantly important but developers ignore it, most of the time, due to the pressure of the deadlines. While developers may document the most important features modification or bugs fixing, recent empirical studies show that the documentation of quality improvements and/or refactoring is often omitted or not accurately described. However, the automated or semi-automated documentation of refactorings has not been yet explored despite the extensive work on the remaining steps of refactoring including the detection, prioritization and recommendation. In this paper, we propose a semi-automated refactoring documentation bot that helps developers to interactively check and validate the documentation of the refactorings and/or quality improvements at the file level for each opened pull-request before being reviewed or merged to the master. The bot starts by checking the pull-request if there are significant quality changes and refactorings at the file level and whether they are documented by the developer. Then, it checks the validity of the developers description of the refactorings, if any. Based on that analysis, the documentation bot will recommend a message to document the refactorings, their locations and the quality improvement for that pull-request when missing information is found. Then, the developer can modify his pull request description by interacting with the bot to accept/modify/reject part of the proposed documentation. Since refactoring do not happen in isolation most of the time, the bot is documenting the impact of a sequence of refactorings, in a pull-request, on quality and not each refactoring in isolation. We conducted a human survey with 14 active developers to manually evaluate the relevance and the correctness of our tool on different pull requests of 5 open source projects and one industrial system. The results show that the participants found that our bot facilitates the documentation of their quality-related changes and refactorings.**

***Index Terms*—Intelligent bot, refactoring, documentation.**

I. Introduction

Documentation is a recommended practice in software development and maintenance to help developers understand the code quickly and improve their productivity [1]. According to a study [2], the lack of up-to-date documentation is one of the biggest challenges in software maintenance. In fact, developers often ignore the documentation of their changes due to the time pressure to meet deadlines. The situation is even worse with the documentation of quality improvements since developers only/mainly focus on documenting functional changes and bugs fixing [3]–[5].

Refactoring [6] is used to improve the quality of code while preserving its behavior. Tom Mens et al. [7] defined the different steps of refactoring including the detection, prioritization, recommendation, testing and documentation. While existing refactoring studies extensively addressed the first four steps [8]–[10], the last documentation step received the least attention from the refactoring community and there are no tools support currently for refactorings documentation.

Github is a well-known collaborative platform used by the development community to manage their software projects as part of a continuous integration process. In this context, programmers need documentation such as commit messages and pull requests descriptions to understand the rationales behind changes without digging into the low-level details [11]–[14]. As part of our preliminary work, we found that an average of only 12% of commit messages described applied refactorings for JHot-Draw, Xerces, and three eBay projects while 46% of these systems commits are mainly about refactorings as detected using RefactoringMiner [8]. Furthermore, developers often do not explain why they do the refactorings. Software engineering researchers often use antipatterns as the causes for the refactorings, but they are not accurately documenting the quality improvements of their code in terms of quality metrics. Another study highlighted that several refactoring opportunities or applied refactorings documented in commit messages could not be captured using traditional quality metrics or antipatterns [15]. One of the reasons is that many developers lack the background of exact (formal) definitions of antipatterns and quality metrics so they may use them in different ways than the academic settings. Thus, a tool support is not only needed for the generation of refactoring documentation but also checking and fixing the documentation specified by developers to describe their quality improvements.

To the best of our knowledge, the automated documentation of refactorings has not been explored yet. Therefore, we need semi-automated tool support for checking/validating and recommending refactorings documentation. This documentation system will enhance the understandability of introduced quality improvements and the rationale behind that, and will motivate developers to conduct refactorings. A recent study of Mcburney et al. [1] shows that documentation needs to be prioritized for refactoring.

In this paper, we propose a semi-automated bot, implemented as a Git app, to generate documentation for two different levels of refactorings. The documentation for code-level refactorings and architectural refactorings will be provided in one message that, if accepted, will be submitted as a description for the pull-request. When the developer submits

2470-6892/19/$31.00 ©2019 IEEE
DOI 10.1109/SCAM.2019.00026

a pull-request, our documentation bot will generate a natural language explanation for each introduced quality changes and refactoring using a rules-based approach, linking the quality improvements to the applied refactorings. Even though we are able to automatically generate explanations for refactorings and quality changes, the developer's intervention is required since they may not find all the generated messages important to integrate into the pull-request description or they may disagree with some of them. In other words, a developer in the loop to evaluate the documentation is necessary to make sure that what we described is actually what he/she intended to change in that specific pull request. In our interactive documentation framework, the users can accept, reject or modify the suggested message. An accepted documentation will be automatically submitted as a description to the pull-request. Since refactoring do not happen in isolation most of the time, the bot is documenting the impact of a sequence of refactorings, in a pull-request, on quality and not each refactoring in isolation. Programmers frequently floss refactor, that is, they interleave refactoring with other types of programming activity. Thus, the documented refactorings and quality changes are actually appended to other descriptions related to functional changes.

We conducted a human survey with 14 active developers to manually evaluate the relevance and the correctness of our tool on different pull requests of 5 open source projects. The results show that the participants found that our bot facilitates the documentation of their quality-related changes and refactorings. A tool demo of our refactoring documentation bot can be found in [16].

The primary contributions of this paper can be summarized as follows:

1) The paper introduces, for the first time, a documentation bot for refactorings implemented as a Git app that can be easily integrated to any GitHub repository. The bot generates in natural language a pull-request description documenting the applied refactorings, their rationale and explanations on their impact on quality. It can also detect inconsistencies in the commit messages or pull-request description already documented by the developer then suggests how to fix them.
2) The developer can interact with the bot to accept/modify/reject the recommended refactorings documentation after checking the explanation provided by the bot in a Web app linked to GitHub.
3) The paper reports the results of an empirical study on the implementation of our approach. The obtained manual evaluation results by practitioners provide evidence to support the claim that our bot generates relevant and consistent documentation for refactorings.

The remainder of this paper is structured as follows. Section 2 presents relevant background details. Section 3 describes our approach while the results obtained from our experiments are presented and discussed in Section 4. Threats to validity are discussed in Section 5. Section 6 provides an account of related work. Finally, in Section 7, we summarize our conclusions and present some ideas for future work.

TABLE I: Quality attributes and their computation equations.

Quality attributes	**Definition**
	Computation
Reusability	A design with low coupling and high cohesion is easily reused by other designs.
	$0.25 * Coupling + 0.25 * Cohesion + 0.5 * Messaging + 0.5 * DesignSize$
Flexibility	The degree of allowance of changes in the design.
	$0.25 * Encapsulation - 0.25 * Coupling + 0.5 * Composition + 0.5 * Polymorphism$
Understandability	The degree of understanding and the easiness of learning the design implementation details.
	$0.33 * Abstraction + 0.33 * Encapsulation - 0.33 * Coupling + 0.33 * Cohesion - 0.33 * Polymorphism - 0.33 * Complexity - 0.33 * DesignSize$
Functionality	Classes with given functions that are publicly stated in interfaces to be used by others.
	$0.12 * Cohesion + 0.22 * Polymorphism + 0.22 * Messaging + 0.22 * DesignSize + 0.22 * Hierarchies$
Extendibility	Measurement of design's allowance to incorporate new functional requirements.
	$0.5 * Abstraction - 0.5 * Coupling + 0.5 * Inheritance + 0.5 * Polymorphism$
Effectiveness	Design efficiency in fulfilling the required functionality.
	$0.2 * Abstraction + 0.2 * Encapsulation + 0.2 * Composition + 0.2 * Inheritance + 0.2 * Polymorphism$

II. Problem Statement

A. Background

Quality attributes. The QMOOD model is one of the most widespread quality models to estimate the effect of code changes on software quality. This model is defined as a set of quality metrics, using the ISO 9126 specification [17]. As described in table I, each of the used quality metrics is defined using a combination of low-level metrics. One advantage of the QMOOD model that makes it widely used in existing studies and also in industry [18]–[20]. QMOOD model is based on six high-level design quality attributes (reusability, flexibility, understandability, functionality, extendibility, and effectiveness) which helps to assess the quality of the software from all its perspectives. These six quality attributes can be easily calculated using the 11 lower level design metrics.

Refactoring documentation: pull-requests description and commit messages. Nowadays, version control systems such as Github are widely used to manage the evolving source code of software projects. Each time a developer commits a change to his branch in a version-control repository, a commit dedicated for this change is created and the developer is able to write a textual message called a commit message to describe the code-level changes that he applied. After performing a set of code-level changes, the programmer submits a pull request

as an architecture-level change in which they write a thorough description of the new changes. If the changes are accepted, the branch is merged into the master branch. Figure 1 shows the overall architecture refactoring process in a version-control repository. The list of refactoring types that can be supported by our bot are described in Table II.

TABLE II: Refactoring types considered in our study

Refactoring Types	Definition
Encapsulate Field	Changes the access modifier of public fields to private and generates it getter and setter.
Increase Field Security	Changes the access modifier of protected fields to private, and of public fields to protected.
Decrease Field Security	Changes the access modifier of protected fields to public, and of private fields to protected.
Pull Up Field	If two subclasses have the same field then this rule moves this field to their superclass.
Push Down Field	If a field is used by only some subclasses then this rule moves this field to those subclasses.
Move Field	Moves a field to another class.
Increase Method Security	Changes the access modifier of protected methods to private, and of public methods to protected.
Decrease Method Security	Changes the access modifier of protected methods to public, and of private methods to protected.
Pull Up Method	If two subclasses have the same method then this rule moves the method to their superclass.
Push Down Method	If a method is used by only some subclasses classes then this rule moves the method to those subclasses.
Move Method	Moves a method to another class.
Extract Class/Method	Creates a new class/method from an existing one.
Extract Superclass	If two subclasses have similar features, this rule creates a superclass and moves these features into it.
Extract Subclass	If two superclasses have similar features, this rule creates a subclass and moves these features into it.
Rename Mehtod/Class/Field	Changes the name of a code element.

B. Motivations

During our extensive interactions with software developers from industry, we observed that a lot of their projects had little to no refactoring documentation. Developers confirmed that they consider documentation very important but the limited time and budget prevented them from adequately document their work especially related to the quality improvements. They confirmed in one of the surveys with industry, as part of an NSF project, that documenting their changes takes time since they have to write what refactorings they applied, their locations and what they intended to improve in their code quality. They also claimed that it is not always straightforward to specify the quality attributes to improve since several programmers in the organization may use different jargon to describe quality improvements.

Developers need documentation to comprehend refactoring, but they may not use traditional academic words to explain the refactorings such as antipatterns, code smells, and even their perception of quality metrics is different from the academic one [1]. As part of our survey and analysis with the industrial partners, we found that an average of only 12% of commit messages described applied refactorings for JHotDraw, Xerces, and three industrial projects while 38% of these systems' commits are mainly about refactorings as detected using REFACTORINGMINER [8]. Software engineering researchers often use antipatterns as the causes for the refactorings, but in our preliminary work, we found that only 0.13% of the commit messages from 1,984 popular projects in GitHub contain any antipattern. For example, *abstraction inversion*, a design antipattern of not exposing a functionality required by users, does not occur once in all the commit messages. This observation indicates that developers do not know about the terms of antipatterns, such as *abstraction inversion*, or they do not make connections between refactorings and antipatterns. Therefore, we need to understand the developers' intention when they are performing refactorings from commit messages without assuming that they have the background knowledge of antipatterns.

We used the 59 software engineering antipattern terms defined in Wikipedia. Then, we searched these antipattern terms in all the commit messages from 1,984 popular projects (including C and Java) in GitHub. Only 0.13% of the 8.4 million commit messages mention any antipattern term. This shows that developers do not use antipattern terms in software documents, which indicates that developers may not understand antipattern terms. Furthermore, we searched these antipattern terms in three large-scale projects' pull requests, Redis, React-native, and Git. In all the 9,172 closed pull requests, we found only 14 "hard code", four "call super", two "magic numbers", one "circular dependency", and one "spaghetti code." Missing antipattern terms in commit messages does not mean that developers do not explain refactoring opportunities.

The two main challenges associated with the current refactoring documentation can be presented as follows:

- **Poorly written pull request documentation:** Figure 3 shows that in the pull request captured in Figure 2, 4 out of 6 QMOOD quality attributes were improved. Despite the different changes in the quality attributes, the developer did not accurately document his changes in a well-written and comprehensive way that shows how importantly his changes impacted the quality.
- **Documenting functional changes rather than quality changes:** Programmers, when working in teams, try to accurately document their pull request to facilitate the collaboration. Despite the effort to write good and comprehensive documentation, developers often document the code changes which are related to the functional requirements of the software. They often forget to describe and explain the changes from quality perspective. Non-functional requirements such as the " quality attributes" improvement are often neglected by developers in their documentation as described in Figure 4 that shows the significant quality improvements before and after the pull request.

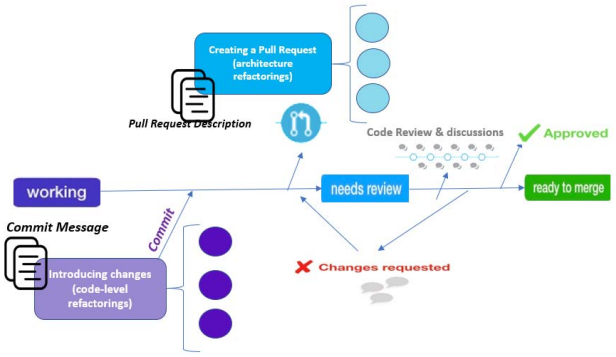

Fig. 1: An architecture refactoring process in a version-control repository

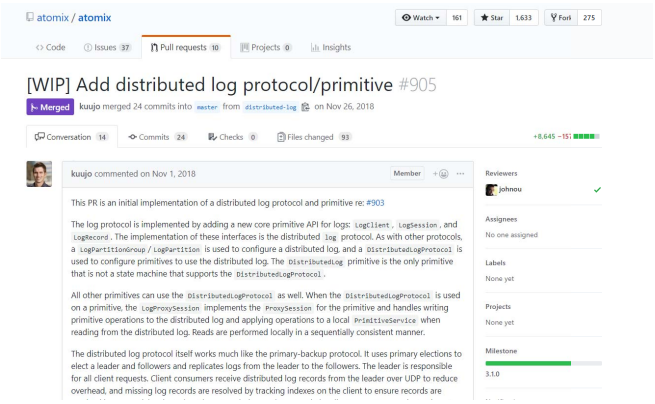

Fig. 4: PR with only functional changes are documented

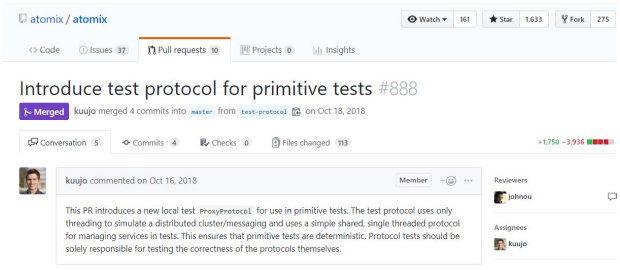

Fig. 2: Pull Request with Poor Documentation

III. Refactoring Documentation Bot

Figure 5 gives an overview of our refactoring documentation bot consisting mainly of three main components: 1) the analysis of the pull-request changes to identify the changed files and evaluate the quality changes; 2) the check of the documentation written by the developer to identify any missing or potential incorrect documentation about the refactorings; and 3) the rules-based generation of the documentation. To generate commit messages, there are three types of approaches: (a) rule-based natural language generation systems; (b) search-based systems that find the most similar commits in the history and use their commit messages; and (c) deep learning models as natural language generation systems. Rule-based approaches, such as DeltaDoc [21], ChangeScribe [22], [23], and others [24], [25], extract the information of a commit's changes and generate commit messages based on rules. Our bot is using the third category of documentation generation approaches, for timely response in terms of execution time, by linking the identified quality changes to specific pre-defined templates to document them as detailed later. Once the refactorings documentation of the Pull-Request is generated, the developer can interact with the bot to accept or reject or modify some of the generated sentences after checking the explanations supporting them in a Web app.

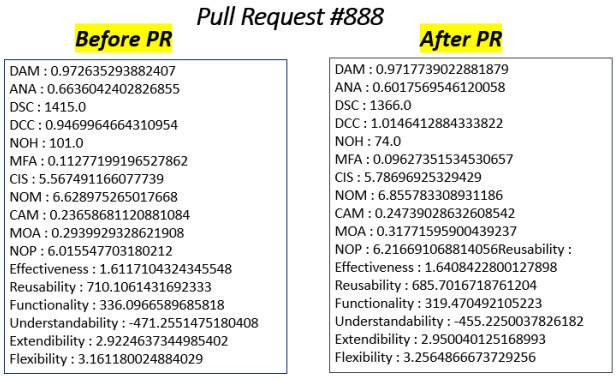

Fig. 3: The quality metrics change in the pull request.

The documentation-Bot is a Spring Boot application that is implemented as a GitHub App [26]. The bot can be used by any public and private GitHub Java repository without restrictions after simply adding it to the repository. Then, the bot will start monitoring the repository and get notified by any new or opened pull-request then it will execute in a sequence the three main components as detailed in Figure 5.

A. Pull-Request Changes Analysis:

When the documentation-bot gets notified of a new pull request, it clones the repository on GitHub for local editing of the source code. The bot extracts automatically all commits messages and modified files of the submitted pull-request. The GitHub API was used to identify these changed files. In order to assess the quality change, it compares the QMOOD quality attributes value at the file level before and after the pull request using our own parser. We have also used RefactoringMiner [8] to find out which refactorings have been applied in that Pull-Request. We selected RefactoringMiner due its high precision and recall score of more than 90% as reported in [8].

B. Checking the Current Documentation of the Developer

After the identification of the changed files, the important QMOOD quality changes and the refactorings from the history of commits in the pull request as described in the previous step,

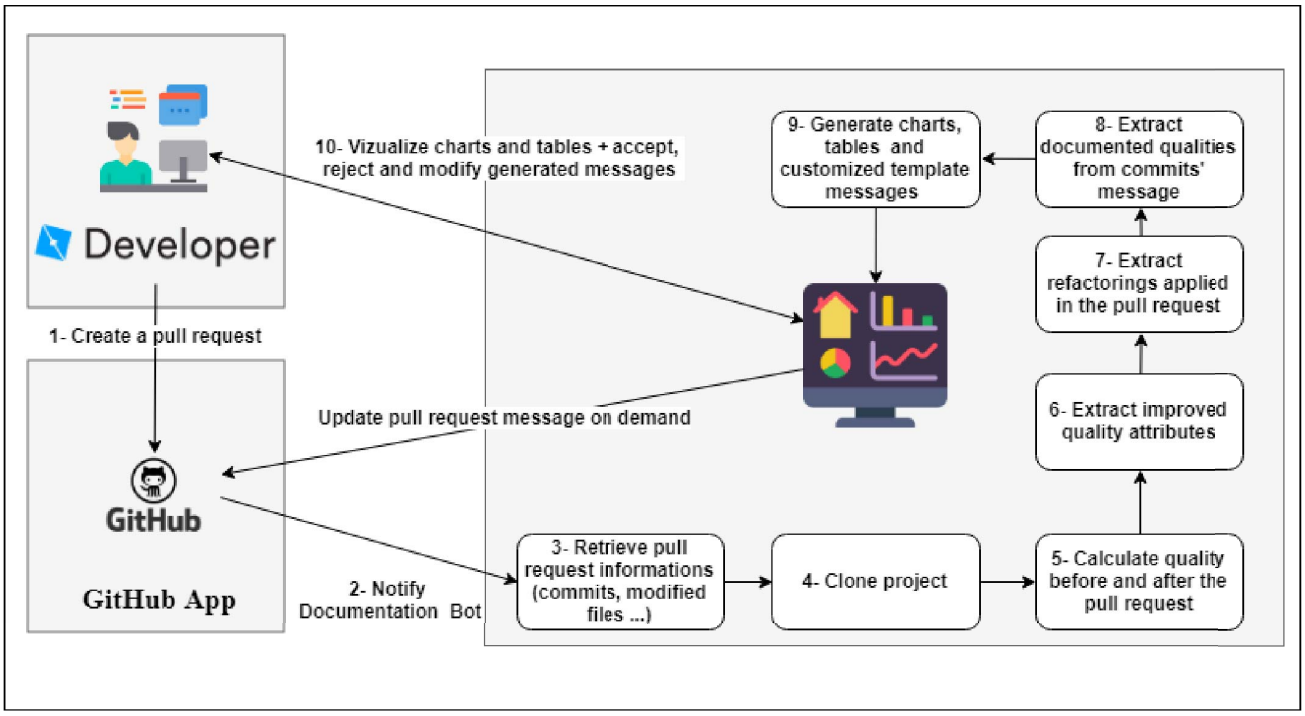

Fig. 5: Approach Overview: Refactoring Documentation Bot

the refactoring documentation-bot checks whether refactorings and their quality change have been documented by developers. In order to perform this verification step, we manually defined a large set of keywords that may cover most of the words used by developers to document quality attributes. Then, we manually classified those keywords into the 6 QMOOD categories (extendibility, reusability, flexibility, understandability, functionality, extendibility, effectiveness). The full list of used keywords can be found in Table III. These keywords have been already defined in the literature based on different surveys including Microsoft developers [27].

The combination of keywords and the detected refactorings along with the name of the modified files represent a sufficient set of features that help us checking whether the specific quality attributes and refactorings detected in the previous step are documented in any of the commit messages and the developer's pull request description. This step serves as both detecting inconsistencies in the refactoring documentation manually provided by the developer and detecting missing refactoring and quality documentation. A recent empirical study shows that developers may introduce inconsistent documentation of refactorings and quality changes [15]. The bot can check, for instance, if reusability was really improved as claimed by the developer in the commit message or the pull-request description.

C. Generation of the Refactoring Documentation and Interaction with Developers

The previous two steps are important towards the generation and correction of the refactoring and quality documentation. The bot will not only be limited to generating or fixing the documentation but also 1) providing a support of the recommended documentation based on the identified refactoring and quality attributes change; and 2) enabling developers interaction to accept or reject or modify the documentation as shown in Figure 11. To make the interaction easy, we are providing low-level interactions at the file level by linking the generated documentation to the changed file(s).

The generated refactoring documentation will follow a specific set of rules template as described in Figure 7: Our message will be composed of the location (file name), the refactoring applied and the quality attributes that have significantly changed and the developers missed them in their documentation. In other words, our bot will document what has been refactored? Why the refactorings were applied? What is the impact of these refactorings on quality. Then, the developers can interact to introduce more details if needed.

After a round of interactions, the developer may decide to update the current description and messages on the GitHub repository as shown in Figure 8.

IV. Validation

To evaluate the ability of our refactoring documentation bot to generate relevant messages for commits and pull-requests, we conducted a set of experiments based on 5 open

TABLE III: List of used keywords related to refactoring

Abstraction	Access	Aggregate	Anti Pattern	Antipattern	Architecture
Change design	Cleanup	Code beauty	Code cleansing	Code cleanup	Code cosmetics
Code improvements	Code optimization	Code reformatting	reordering	Code revision	Code smells
Cohesion	Compatibility	Complexity	Composition	Cosmetic changes	Coupling
Dead code	Decompose	Decoupling	Deprecated code	Design	Design Pattern
Design metric	Designed code	Divide	Duplicate	Easy	Effectiveness
Encapsulation	Enhance	Extend	Extendibility	Extract	Fix a design flaw
Fix code smell	Fix issue	Fix module structure	Fix quality	Fix technical debt	Flexibility
Functionality	Getting code out of	Hierarchies	Improve	Inheritance	Inline
Less code	Long method	Maintenance	Make easier	Messaging	Metrics
Modular	Modularize	Moved code out of	Multi module	Nicer code	Move
Performance	Polishing code	Polymorphism	Poor coding	Pull down	Push
Pull up	Quality	Redesign	Redundant	Refactor	Reformat
Rename	Remove dependency	Reorganize	Replace	Restructure	Reusability
Reuse	Rework	Rewrite	Robustness	Scalability	Separate
Simplify	Split	Stability	Structural changes	Structure	Understandability
Understanding	Unneeded	Unnecessary code	Unused	Useless	Visibility

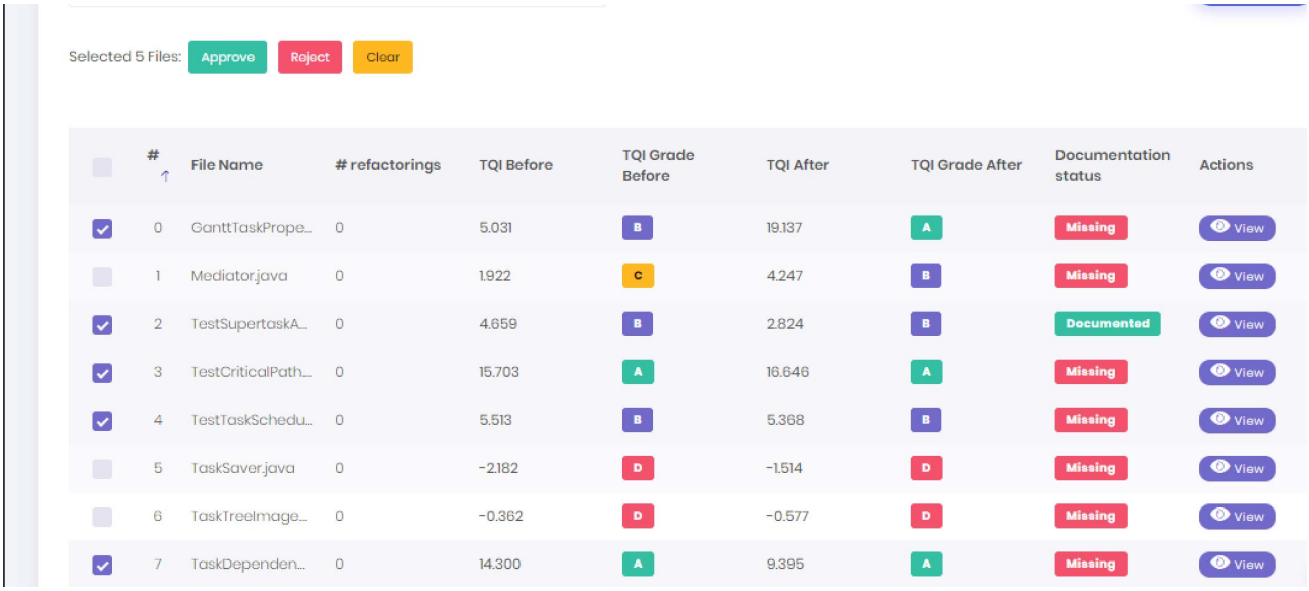

Fig. 6: Developer's interaction with the Refactoring Documentation Bot

source systems. A demo of the refactoring documentation bot can be found in [16]. In this section, we first present our research questions and validation methodology followed by experimental setup. Then we describe and discuss the obtained results.

A. Research Questions

It is important to evaluate, first, the correctness of the generated refactoring documentation. Developers are not interested, in practice, to include *all* the correct refactorings documentation especially at the pull-request level. Thus, we evaluated the relevance of the recommended refactorings documentation to include in commits and pull-request messages and analyzed the interaction data of the users. We defined two main research questions to measure the correctness, relevance and benefits of our refactoring documentation bot. The research questions are as follows:

- **RQ1: Correctness and Relevance of the recommended refactoring documentations.** To what extent our bot can generate correct and meaningful documentations based on the feedback from participants?
- **RQ2: Insights from practitioners.** How do programmers evaluate the **usefulness of our tool** (survey)?

B. Experimental Setting and Studied Projects

To address the different research questions, we used the 5 open source systems in Table IV. We selected these projects because of their size, number of commits, applied refactorings,

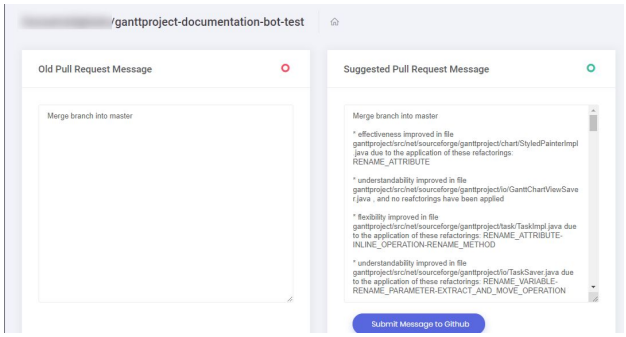

Fig. 7: Developer's Pull Request Description vs. our Bot's Description

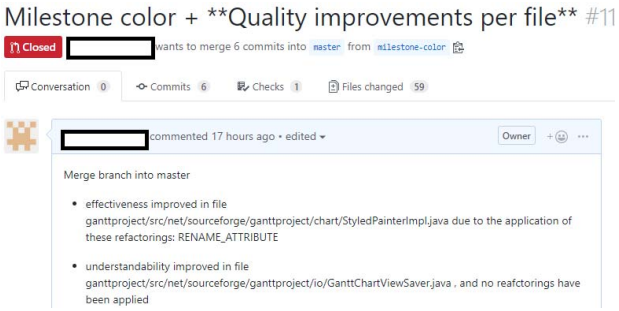

Fig. 8: A generated pull request description submitted on GitHub by our bot

etc. To answer RQ1, we asked a group of 14 active programmers to manually evaluate the correctness and relevance of the messages generated by our bot documenting the quality improvements and related refactoring. The correctness is defined as the number of correctly documented commits and pull-requests over the total number of generated messages. Since not all correct refactoring documentations will be actually applied by developers, we asked them to also report those they found relevant and actually integrated to expand current pull-request/commits messages then we calculated the relevance score which is the number of relevant messages divided by the total number of messages generated by the bot. We have also collected the interaction data between the developers and the bot in terms of the number of accepted, modified and rejected messages.

Since not all pull-requests are mainly related to refactorings, we selected the ones that included at least 5 refactoring operations per pull-request and made significant change in the average QMOOD quality measure of at least 0.1. The number of pull-requests per project are described in Table IV.

To answer RQ2, we used a questionnaire that collected the opinions of the participants about their experience in using our bot. It contains mainly questions on the usability of the documentation bot, the use of QMOOD to document quality changes, the importance of refactoring documentation, and the need for a refactoring documentation bot.

TABLE IV: Summary of the evaluated systems.

System	Release	#Classes	#Pull Requests
Gson	v2.8.5	206	18
JHotDraw	v7.5.1	585	11
GanttProject	v1.10.2	241	14
Apache Ant	v1.8.2	1191	9
JFreeChart	v1.0.9	521	12

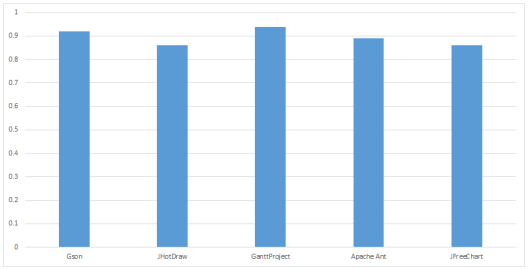

Fig. 9: The average manual correctness score on the different 5 systems as evaluated by the participants.

All the participants are volunteers and familiar with Java development and refactoring. The experience of these participants on Java programming ranged from 4 to 19 years. We carefully selected the participants to make sure that they already applied refactorings during their previous experiences in development.

Participants were first asked to fill out a pre-study questionnaire containing five questions. The questionnaire helped to collect background information such as their role within the company, their programming experience, and their familiarity with software refactoring. In addition, all the participants attended one lecture about refactoring. Each participant was asked to evaluate all the pull-requests selected for our experiments on the different projects during a period of one week.

C. Results

Results for RQ1. Figure 9 summarizes our findings regarding the correctness of the generated pull-request and commit messages on the 5 systems. We found that a considerable number of proposed documentation for refactoring, with an average between 94% and 86% respectively for Gantt and JFreeChart, were already considered correct by the participants. The manual correctness score was consistent on all the five systems which confirm that the results are independent from the size of the systems, number of refactorings and quality changes.

We report as well the results of our empirical evaluation of the relevance (not only correctness) in Figure 10. In fact, developers may not want to document all quality changes and associated refactorings in the commits and pull request message. As reported in this figure, the majority of the refactoring documentation solutions recommended by our interactive approach were relevant and approved by developers. On average, for all of our five studied projects,the manual relevance score is 4.3 based on a Likert scale (from 1 to 5). The highest MC score is 4.6 for both the Gantt and

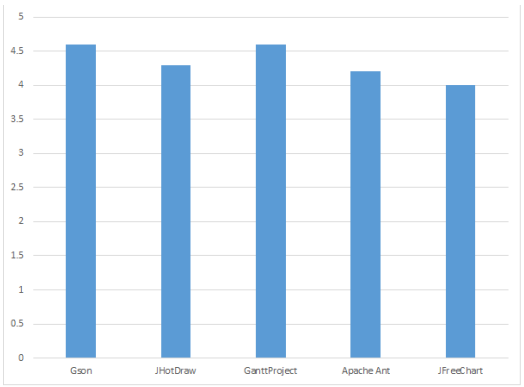

Fig. 10: The average manual relevance score on the different five systems

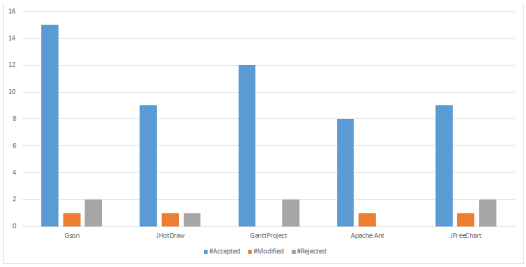

Fig. 11: The average number of AR (percentage of accepted messages), NMR (percentage of modified messages) and NRR (percentage of rejected messages) on the different five systems.

Gson projects and the lowest score is 4 for JFreeChart. Most of the refactorings/quality changes documentation that were not manually approved by the developers were found to be introducing minor improvements or they have to be grouped together to make sense.

Considering three other metrics NAR (percentage of accepted messages), NMR (percentage of modified messages) and NRR (percentage of rejected messages), we seek to evaluate the efficiency of our interactive approach to avoid a high interaction effort. We recorded these metrics using a feature that we implemented in our tool to record all the actions performed by the developers during the evaluation. Figure 11 shows that, on average, more than the majority of the generated messages were applied by the developers an few of them were either modified or rejected. For instance, we found on the large Gson open source system that 15 out of the 18 generated messages were approved by developers and only two were rejected. Thus, it is clear that our recommendation tool successfully suggested a good set of messages to documment refactorings/quality changes.

Results for RQ2. We asked participants to rate their agreement on a Likert scale from 1 (complete disagreement) to 5 (complete agreement) with the following statements:

1) The interactive refactoring documentation bot is desirable feature for continuous integration.

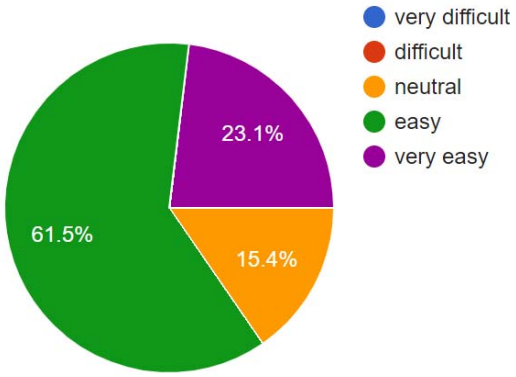

Fig. 12: Distribution of the opinions of the participants about the usability of our refactoring documentation bot

2) The documentation of refactorings based on their impact on the QMOOD changes is effective to explain the rationale.

The post-study questionnaire results show the average agreement of the participants was 4.7 and 4.2 based on a Likert scale for the first and second statements, respectively. This confirms the usefulness of our refactoring documentation approach for the software developers considered in our experiments. Most of the participants mention that our interactive documentation is faster than the tedious manual way to document refactorings since they admitted the lack of refactoring documentation comparing to functional changes. Thus, the developers liked the functionality of our tool that helps them to expand the commits and pull-requests message in an interactive fashion.

The participants also suggested some possible improvements to our refactoring documentation bot. Some participants believe that it will be very helpful to extend the tool by adding a new feature to select up-front the types of refactoring and quality improvements to be documented. Another suggested improvement is to expand the tool to generate documentation for both functional and non-functional changes.

Figure 12 shows that over 60% of the participants agreed that the bot was easy to use especially in the context of continuous integration. The bot did not require any configuration and it is installed as a Git app in any GitHub repository. When the developers can check his pull request to add more documentation from the bot before submitting it for peer review.

Over 75% of the participants found that documenting refactorings is important as described in Figure 13. The majority of them highlighted that it is a missing feature in existing refactoring tools and it can help reviewers in understanding the code changes that are related to refactorings and why they were applied. The managers/executives want to check if their developers care about the quality of their code thus it is easier for them to check the pull-requests/commits description rather than looking to the code.

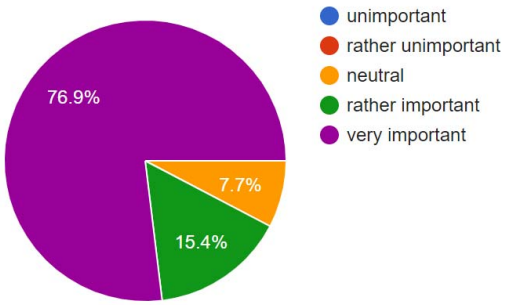

Fig. 13: Approach Overview: Refactoring Documentation Bot

V. Threats to validity

We discuss in this section the different threats related to our experiments.

Internal validity. Threats to internal validity can be related to the list of keywords and their grouping into the QMOOD categories that we used to identify whether the quality attributes changes and the refactorings were documented by the developers. However, the impact of this threat was limited by considering the use of RefactoringMiner to identify the actual refactorings applied by developers. Furthermore, the user interaction may help mitigating this threat since our goal is not fully automating the documentation generation process.

Construct validity is concerned with the relationship between theory and what is observed. We have used the QMOOD quality attributes to capture the quality changes between commits. While the QMOOD model is already empirically validated by existing studies [28], it is possible that some of the quality changes may not be detected using QMOOD. Another threat to construct validity can be related to the diverge opinions of developers involved in our experiments when evaluating the documentation. Actually, we received different opinions about the suggested documentation in terms of importance and relevance which may impact the validity of our results. However, some of the participants are the original programmers of the evaluated systems which may reduce the impact of this threat where they are confident about the relevance of the documented quality changes.

External validity refers to the generalizability of our findings. We performed our experiments on 5 open-source systems belonging to different domains and we conducted our survey with active developers. However, we cannot assert that our results can be generalized to other applications and other developers. Our bot is mainly now limited to object-oriented programming languages. However, Java, for instance, is one of the most popular programming language which is used in a large number of projects. In the future, we will extend our approach to support other programming languages and paradigms. Future replications of this study are necessary to confirm our findings.

VI. Related Work

We summarize, in the following, existing studies in the area of software documentation. We classify them into three categories:commit messages generation, pull request description generation and source code summarization.

Most of the existing studies investigate software documentation and its importance through surveys. For example, Forward et al. [29] conducted a survey with software professionals about existing documentation tools. Their results prove that software profesionals are looking for new technologies to improve the automation of the documentation process and its maintenance. Software documentation was also addressed in the study [2] where de Souza et al. tried to establish what documentation artificats are the most useful to software maintainers through a survey.

We categorize the commit messages generation techniques into three groups: **(1) rule-based** natural language generation systems; **(2) search-based systems** that find the most similar commits in the history and use their commit messages; and **(3) deep learning models** as natural language generation systems. Rule-based approaches, such as Delta- Doc [21], ChangeScribe [22], [23], and others [24], [25], extract the information of a commits changes and generate commit messages based on rules. For instance, Buse et al. have built DeltaDoc [21], which extracts path predicates of code source change, then it generates and follows a set of predefined rules to generate a summary. ChangeScribe [23] starts first by analyzing source code changes and Abstract Syntax Trees. Then it generates a commit message following predefined templates and rules. To reduce the length of the genrated message, Shen et al. proposed an approach similar to ChangeScribe where they used method stereotypes and the type of change to generate commit messages but they removed repeated information in the change [24]. The tool proposed by Le et al. in their study [30] uses dynamic analysis to infer the semantics of changes between two versions of a code.

The commit messages generated by these approaches can be lengthy and full of details. In contrast, search-based approaches,such as the one proposed by Huang et al. [31] and the one proposed by Liu et al. [32], output human-written commit messages. Given a commit, these approaches find an existing commit with the code changes that are most similar to the given commit. Then, the search-based approaches reuse the found commits commit message as the message for the new commit. These approaches work for the code changes that repeat in software repositories, such as updates of API dependencies. However, these approaches do not work for the new code changes.

Although many refactoring tools have been built [9], [10], [33], [34], there is no tool for automated architecture refactoring documentation. One recently proposed tool, RCLinker [35] (designed for linking commit messages to the related issues), may be used for linking pull requests to the corresponding issues (bug reports). Another potentially useful approach is treating pull requests as commit messages, and

using automated commit message generation tools [21]–[25] for generating pull requests. This approach may work for the architecture refactorings that have fewer changes, but it does not work for large-scale refactorings. Similarly, we can treat pull requests as version updates and use the release notes generation tools such as ARENA [36]. ARENA combines changes from the source code, libraries, and licenses with related issues to generate release notes.

Source code summarization techniques are used to generate documentation for code changes in commits [37].Some existing work leverages text retrieval techniques to generate source code summaries. For instance, Haiduc et al. [38] used Latent Semantic Indexing (LSI) [39], to generate source code entities descriptions. Moreover, the study [40] led by Haiduc et al. shows that a Vector Space Model could also be useful in automatically generating summaries. Recent existing studies proposed tools that generate natural language summaries of source code, such as Java methods and classes [41]–[47].For instance, to generate summaries for Java methods, Sridhara et al. proposed a tool that first extracts relevant information from Java methods then expresses the extracted content in natural language based on predefined text templates [44]. This work was extended to automatically describe high-level actions within methods [48]. In addition to the previous mentioned techniques, Iyer et al. used NMT(Neural Machine Translation) to build Code-NN, a framework that generates summaries for C and SQL code [49].

VII. Conclusion

In this paper, we presented a documentation bot to document the developers changes in terms of quality attributes improvement and refactorings. The bot also enable the interaction with the developer to adjust the generated documentation. To evaluate the correctness and the relevance of our bot, we selected developers to evaluate our bot on different pull requests of 5 open-source projects. The results show clear evidence that our bot helped developers documenting the quality improvement of the applied refactorings.

Future work will involve extending our experiments on larger set of systems and participants. We will also evaluate different documentation generation techniques to adopt them for documenting refactorings rather than the use of the rules-based techniques.

References

[1] P. W. McBurney, S. Jiang, M. Kessentini, N. A. Kraft, A. Armaly, M. W. Mkaouer, and C. McMillan, "Towards prioritizing documentation effort," *IEEE Transactions on Software Engineering*, vol. 44, no. 9, pp. 897–913, 2018.

[2] S. C. B. de Souza, N. Anquetil, and K. M. de Oliveira, "A study of the documentation essential to software maintenance," in *Proceedings of the 23rd annual international conference on Design of communication: documenting & designing for pervasive information*. ACM, 2005, pp. 68–75.

[3] T. Roehm, R. Tiarks, R. Koschke, and W. Maalej, "How do professional developers comprehend software?" in *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, 2012, pp. 255–265.

[4] B. Fluri, M. Wursch, and H. C. Gall, "Do code and comments co-evolve? on the relation between source code and comment changes," in *14th Working Conference on Reverse Engineering (WCRE 2007)*. IEEE, 2007, pp. 70–79.

[5] M. Kajko-Mattsson, "A survey of documentation practice within corrective maintenance," *Empirical Software Engineering*, vol. 10, no. 1, pp. 31–55, 2005.

[6] M. Fowler, *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 2018.

[7] T. Mens and T. Tourwé, "A survey of software refactoring," *IEEE Transactions on software engineering*, vol. 30, no. 2, pp. 126–139, 2004.

[8] N. Tsantalis, M. Mansouri, L. M. Eshkevari, D. Mazinanian, and D. Dig, "Accurate and efficient refactoring detection in commit history," in *Proceedings of the 40th International Conference on Software Engineering*. ACM, 2018, pp. 483–494.

[9] G. Bavota, S. Panichella, N. Tsantalis, M. Di Penta, R. Oliveto, and G. Canfora, "Recommending refactorings based on team co-maintenance patterns," in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. ACM, 2014, pp. 337–342.

[10] V. Alizadeh and M. Kessentini, "Reducing interactive refactoring effort via clustering-based multi-objective search," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, 2018, pp. 464–474.

[11] M. Yan, Y. Fu, X. Zhang, D. Yang, L. Xu, and J. D. Kymer, "Automatically classifying software changes via discriminative topic model: Supporting multi-category and cross-project," *Journal of Systems and Software*, vol. 113, pp. 296–308, 2016.

[12] A. Mockus and L. G. Votta, "Identifying reasons for software changes using historic databases." in *icsm*, 2000, pp. 120–130.

[13] Y. Fu, M. Yan, X. Zhang, L. Xu, D. Yang, and J. D. Kymer, "Automated classification of software change messages by semi-supervised latent dirichlet allocation," *Information and Software Technology*, vol. 57, pp. 369–377, 2015.

[14] A. E. Hassan, "Automated classification of change messages in open source projects," in *Proceedings of the 2008 ACM symposium on Applied computing*. ACM, 2008, pp. 837–841.

[15] J. Pantiuchina, M. Lanza, and G. Bavota, "Improving code: The (mis) perception of quality metrics," in *2018 IEEE International Conference on Software Maintenance and Evolution, ICSME 2018, Madrid, Spain, September 23-29, 2018*, 2018, pp. 80–91. [Online]. Available: https://doi.org/10.1109/ICSME.2018.00017

[16] "Interactive Refactoring Documentation Bot Demo." [Online]. Available: https://sites.google.com/view/scam-2019

[17] J. Bansiya and C. G. Davis, "A hierarchical model for object-oriented design quality assessment," *IEEE Transactions on software engineering*, vol. 28, no. 1, pp. 4–17, 2002.

[18] M. OKeeffe and M. O. Cinnéide, "Search-based refactoring for software maintenance," *Journal of Systems and Software*, vol. 81, no. 4, pp. 502–516, 2008.

[19] A. C. Jensen and B. H. Cheng, "On the use of genetic programming for automated refactoring and the introduction of design patterns," in *Proceedings of the 12th annual conference on Genetic and evolutionary computation*. ACM, 2010, pp. 1341–1348.

[20] S. Lee, G. Bae, H. S. Chae, D.-H. Bae, and Y. R. Kwon, "Automated scheduling for clone-based refactoring using a competent ga," *Software: Practice and Experience*, vol. 41, no. 5, pp. 521–550, 2011.

[21] R. P. Buse and W. Weimer, "Automatically documenting program changes." in *ASE*, vol. 10, 2010, pp. 33–42.

[22] L. F. Cortés-Coy, M. Linares-Vásquez, J. Aponte, and D. Poshyvanyk, "On automatically generating commit messages via summarization of source code changes," in *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*. IEEE, 2014, pp. 275–284.

[23] M. Linares-Vásquez, L. F. Cortés-Coy, J. Aponte, and D. Poshyvanyk, "Changescribe: A tool for automatically generating commit messages," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 2. IEEE, 2015, pp. 709–712.

[24] J. Shen, X. Sun, B. Li, H. Yang, and J. Hu, "On automatic summarization of what and why information in source code changes," in *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*, vol. 1. IEEE, 2016, pp. 103–112.

[25] T.-D. B. Le, J. Yi, D. Lo, F. Thung, and A. Roychoudhury, "Dynamic inference of change contracts," in *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE, 2014, pp. 451–455.

[26] G. Gousios, M. Pinzger, and A. v. Deursen, "An exploratory study of the pull-based software development model," in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 345–355.

[27] M. Kim, T. Zimmermann, and N. Nagappan, "An empirical study of refactoringchallenges and benefits at microsoft," *IEEE Transactions on Software Engineering*, vol. 40, no. 7, pp. 633–649, 2014.

[28] O. Baysal and R. Holmes, "A qualitative study of mozillas process management practices," *David R. Cheriton School of Computer Science, University of Waterloo, Waterloo, Canada, Tech. Rep. CS-2012-10*, 2012.

[29] A. Forward and T. C. Lethbridge, "The relevance of software documentation, tools and technologies: a survey," in *Proceedings of the 2002 ACM symposium on Document engineering*. ACM, 2002, pp. 26–33.

[30] T.-D. B. Le, J. Yi, D. Lo, F. Thung, and A. Roychoudhury, "Dynamic inference of change contracts."

[31] Y. Huang, Q. Zheng, X. Chen, Y. Xiong, Z. Liu, and X. Luo, "Mining version control system for automatically generating commit comment," in *Proceedings of the 11th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. IEEE Press, 2017, pp. 414–423.

[32] Z. Liu, X. Xia, A. E. Hassan, D. Lo, Z. Xing, and X. Wang, "Neural-machine-translation-based commit message generation: how far are we?" in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, 2018, pp. 373–384.

[33] J. Von Pilgrim, B. Ulke, A. Thies, and F. Steimann, "Model/code co-refactoring: An mde approach," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2013, pp. 682–687.

[34] H. Li and S. Thompson, "Automated api migration in a user-extensible refactoring tool for erlang programs," in *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 2012, pp. 294–297.

[35] T.-D. B. Le, M. Linares-Vásquez, D. Lo, and D. Poshyvanyk, "Rclinker: automated linking of issue reports and commits leveraging rich contextual information," in *2015 IEEE 23rd International Conference on Program Comprehension*. IEEE, 2015, pp. 36–47.

[36] L. Moreno, G. Bavota, M. Di Penta, R. Oliveto, A. Marcus, and G. Canfora, "Arena: an approach for the automated generation of release notes," *IEEE Transactions on Software Engineering*, vol. 43, no. 2, pp. 106–127, 2017.

[37] N. Nazar, Y. Hu, and H. Jiang, "Summarizing software artifacts: A literature review," *Journal of Computer Science and Technology*, vol. 31, no. 5, pp. 883–909, 2016.

[38] S. Haiduc, J. Aponte, and A. Marcus, "Supporting program comprehension with source code summarization," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2*. ACM, 2010, pp. 223–226.

[39] T. K. Landauer, P. W. Foltz, and D. Laham, "An introduction to latent semantic analysis," *Discourse processes*, vol. 25, no. 2-3, pp. 259–284, 1998.

[40] S. Haiduc, J. Aponte, L. Moreno, and A. Marcus, "On the use of automated text summarization techniques for summarizing source code," in *2010 17th Working Conference on Reverse Engineering*. IEEE, 2010, pp. 35–44.

[41] P. W. McBurney and C. McMillan, "Automatic documentation generation via source code summarization of method context," in *Proceedings of the 22nd International Conference on Program Comprehension*. ACM, 2014, pp. 279–290.

[42] L. Moreno, J. Aponte, G. Sridhara, A. Marcus, L. Pollock, and K. Vijay-Shanker, "Automatic generation of natural language summaries for java classes," in *2013 21st International Conference on Program Comprehension (ICPC)*. IEEE, 2013, pp. 23–32.

[43] L. Moreno, A. Marcus, L. Pollock, and K. Vijay-Shanker, "Jsummarizer: An automatic generator of natural language summaries for java classes," in *2013 21st International Conference on Program Comprehension (ICPC)*. IEEE, 2013, pp. 230–232.

[44] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker, "Towards automatically generating summary comments for java methods," in *Proceedings of the IEEE/ACM international conference on Automated software engineering*. ACM, 2010, pp. 43–52.

[45] M. Kessentini, M. Wimmer, H. Sahraoui, and M. Boukadoum, "Generating transformation rules from examples for behavioral models," in *Proceedings of the Second International Workshop on Behaviour Modelling: Foundation and Applications*. ACM, 2010, p. 2.

[46] U. Mansoor, M. Kessentini, M. Wimmer, and K. Deb, "Multi-view refactoring of class and activity diagrams using a multi-objective evolutionary algorithm," *Software Quality Journal*, vol. 25, no. 2, pp. 473–501, 2017.

[47] U. Mansoor, M. Kessentini, B. R. Maxim, and K. Deb, "Multi-objective code-smells detection using good and bad design examples," *Software Quality Journal*, vol. 25, no. 2, pp. 529–552, 2017.

[48] G. Sridhara, L. Pollock, and K. Vijay-Shanker, "Automatically detecting and describing high level actions within methods," in *Proceedings of the 33rd International Conference on Software Engineering*. ACM, 2011, pp. 101–110.

[49] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, "Summarizing source code using a neural attention model," in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, vol. 1, 2016, pp. 2073–2083.