Commits Analysis for Software Refactoring Documentation and Recommendation

by

Soumaya Rebai

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer & Information Science)
in the University of Michigan-Dearborn
2021

DEARBORN

ISELab

Intelligent Software Engineering

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

SOC	Service-Oriented Computing
BLOP	Bi-Level Optimization Problem
BLMPO	Bi-Level Multi-Objective Problems
QoS	Quality of Service
LOC	Lines of Code
WS	Web Service
NSGA-II	Non-dominated Sorting Genetic Algorithm
GMM	Gaussian Mixture Model
IGA	Interactive Genetic Algorithm
GA	Classic Genetic Algorithm
QMOOD	Quality Metrics for Object Oriented Designs
NMT	Neural Machine Translation
NLP	Natural Language Processing

ABSTRACT

Software projects frequently evolve to integrate new features, update existing operations and fix errors/bugs to meet new requirements. While this evolution is critical, it may have a negative impact on the quality of the system. To improve the quality of software systems, the first step is **“detection”** of code antipatterns to be restructured which can be considered as “refactoring opportunities”. Th second step is the **“prioritization”** of code fragments to be refactored/fixed. The third step is **“recommendation”** of refactorings to fix the detected quality issues. The fourth step is **“testing”** the recommended refactorings to evaluate their correctness. The fifth and the last step is the **“documentation”** of the applied refactorings. We addressed all the above steps as part of this dissertation. We mainly focused on analyzing commit messages to improve the process of recommending and documenting refactoring. In this thesis, we offer these following contributions:

1. We designed a bi-level multi-objective optimization approach to enable the generation of antipattern examples that can improve the efficiency of detection rules for bad quality designs. The statistical analysis of our results, based on 662 web services, confirms the efficiency of our approach in detecting web service antipatterns comparing to the current state of the art in terms of precision and recall. The combination of dynamic QoS attributes and structural information of web services improved the efficiency of the generated detection rules.

2. Regarding refactoring recommendations, we proposed an approach that combines machine learning with multi-objective optimization techniques. We first identify refac-

toring opportunities by analyzing developer commit messages and check the quality improvements in the changed files, then we distill this knowledge into usable context-driven refactoring recommendations to complement static and dynamic analysis of code. The evaluation of our approach, based on six open-source projects, shows that we outperform prior studies that apply refactorings based on static and dynamic analysis of code alone. This study provides compelling evidence of the value of using the information contained in existing commit messages to recommend future refactorings.

3. We proposed an interactive refactoring recommendation approach that enables developers to pinpoint their preference simultaneously in the objective (quality metrics) and decision (code location) spaces. Developers may be interested in looking at refactoring strategies that can improve a specific quality attribute, such as extendibility (objective space), but such strategies may be related to different code locations (decision space). A manual validation of selected refactoring solutions by developers confirms that our approach outperforms state of the art refactoring techniques.

4. We proposed 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. 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.

5. We performed interviews with and a survey of practitioners as well as a quantitative analysis of 1,193 commit messages containing refactorings. The interviews with 14 developers aim to establish a refactoring documentation model as a set of components. The survey with an additional 75 developers is to understand the experience of develop-

ers in finding and documenting these different components and their importance. The obtained final model includes the following refactoring documentation components to answer the following questions: *how, why, where, what, and when* the refactorings were introduced. Then, we conducted a manual inspection of commit messages to compare the final refactoring documentation model with manual refactoring documentation extracted from open-source projects. Our quantitative results show that while developers may document in isolation the different components over multiple commits, they rarely document all of them in a single commit. Our findings can enable (1) researchers to automatically improve, assess, and generate refactoring documentation, (2) practitioners to use a standard format in documenting and discussing refactorings, and (3) educators to teach and emphasize the different important components of refactoring.

6. We formulated the recommendation of code reviewers as a multi-objective search problem to balance the conflicting objectives of expertise, availability, and history of collaborations. Our validation confirms the effectiveness of our multi-objective approach on 9 open-source projects by making better recommendations, on average, than the state of the art.

7. We built a dataset composed of 50,000+ composite code changes pertaining to more than 7,000 open-source projects. Then, we proposed and evaluated a new deep learning technique to generate commit messages for composite code changes based on an attentional encoder-decoder with two encoders and BERT embeddings. Our results show that our technique overcomes the existing approaches in terms of BLEU score.

CHAPTER I

Introduction

1.1 Research Context

1.1.1 Web Services Defects

Service-Oriented Computing (SOC) has emerged as an evolutionary paradigm that is changing the way software applications are implemented, deployed, and delivered to help industry meet their ever-more-complex challenges [5]. Nowadays, SOC is becoming widely accepted in industry such as FedEx [1], Dropbox [2], Google Maps [3], eBay[4], etc. The massive adoption of this paradigm and its popularity are mainly due to the offered reusability, modularity, flexibility, and scalability [6]. SOC utilizes services which are independent and portable program units as fundamental elements to support rapid, low cost development of heterogeneous and distributed systems [7].

Any successful deployed web services evolve over time to meet the new changes in the requirements and/or to fix bugs. The continuous changes and evolution of web services may create poor and bad design practices which are generally called "antipatterns" that can impact the performance and usability of the web service [8]. Maintaining a good design quality is critical but it is excessively expensive both in time and resources for the service providers.

[1]http://www.fedex.com/ca_english/businesstools/webservices
[2]https://www.dropbox.com/developers/core
[3]developers.google.com/maps/documentation/webservices
[4]https://developer.ebay.com/docs

1.1.2 Software Refactoring

With the ever-growing size and complexity of software projects, there is a high demand for efficient software refactoring [9, 10, 11, 12] tools to improve software quality, reduce technical debt, and increase developer productivity. However, refactoring software systems can be complex, expensive, and risky [13, 14, 15, 16, 17, 18, 19, 20, 21, 22]. A recent study [23] shows that developers are spending considerable time struggling with existing code (e.g., understanding, restructuring, etc.) rather than creating new code, and this may have a harmful impact on developer creativity.

Various tools for code refactoring have been proposed during the past two decades ranging from manual support [24, 25, 26] to fully automated techniques [27, 28, 29, 30, 31, 32, 33, 34, 35, 36]. While these tools are successful in generating correct code refactorings, developers are still reluctant to adopt these refactorings. This reluctance is due to the tools' poor consideration of context and developer preferences when finding refactorings [37, 38, 29, 39]. software refactoring remains a human activity which is hard to fully automate and requires developer insights. Such insights are important because developers understand their problem domin intuitively and may have a clear target end-state in mind for their system. In fact, the preferences of developers ranging from quality improvements to code locations, are still not well supported by existing tools and a large number of refactorings are recommended, in general, to fix the majority of the quality issues in the system. Some existing approaches rely on the use of quality metrics such as coupling, cohesion, and the QMOOD quality attributes [40] to first identify refactoring opportunities, and then to recommend refactorings to fix them. Many of the quality issues detected using structural metrics are known as code smells or antipatterns [41]. However, recent studies have shown that developers are not primarily interested in fixing antipatterns when they are performing refactoring [42].

1.1.3 Composite Changes Documentation

Documentation is a recommended practice in software development and maintenance to help developers understand the code quickly and improve their productivity [43]. According to a study [44], 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 [45, 46, 47].

Software refactoring, defined as code restructuring while preserving the behavior [48], is guided by decisions from developers for various reasons such as improving quality and preventing bugs [49, 48, 50, 51, 52]. Several empirical studies [53, 54, 55, 56] show that refactoring is complex and time-consuming, and usually involves a sequence of dependent code actions to address challenging quality issues [52]. The effective understanding and documentation of refactorings can play a critical role in reducing and monitoring the *technical debt* [57, 58, 59, 60] by different stakeholders including executives, managers, and developers. In particular, refactoring documentation can help developers, managers, and executives keep track of applied refactorings, their rationale, and their impact on the system.

One of the laws of software evolution is that a successful software project evolves over time to meet the new requirements as well as to fix bugs [61]. However, these continuous changes and evolution may result in a poor quality software product [62]. To help developers maintain these changes made in software repositories, one common practice is to write commit messages, which document these changes in version control systems, e.g., Git [63].

To automatically generate such documents, researchers have been creating search-based approaches [64, 65, 66] that find the most similar commits in the project's history, to reuse or adopt their messages. One of the algorithms that they use for calculating the similarity is the nearest neighbor algorithm [65]. Another line of research uses deep learning models, especially neural machine translation (NMT) models, to generate short commit messages based on diff files [67, 68, 66]. Studies have been proposing different variations of a vanilla NMT model to improve the quality of the generated commit messages. For example, Pointer-Generator Network is added to treat out-of-vocabulary words [66]. Xu et al. [69] modified the encoder to take two inputs: code semantics and code structures for commit message generation

1.1.4 Code Reviews

The source code review process has always been one of the most important software maintenance and evolution activities [70]. Several studies show that a careful code inspection can significantly reduce defects and improve the quality of software systems. Recently this process has become informal, asynchronous, light-weight and facilitated by tools [71] [72]. A survey with practitioners, performed by Bacchelli et al. [73], show that code review nowadays is expanding beyond just looking for defects but to also provide alternatives to improve the code and transfer knowledge among developers.

1.2 Problem Statement & Proposed Contributions

In this thesis, we built a framework to improve and assist the quality assurance process via leveraging different algorithms to enhance the mechanisms of detection, correction and documentation of quality issues and code changes:

- Design and implement scalable approach that combines search algorithms with machine learning and natural language techniques for the detection of refactoring opportunities and refactoring recommendation. We utilized for the first time the knowledge extracted from commit messages analysis to improve the existing tools for detection and correction of quality issues.
- Design empirical foundation and a thorough empirical evaluation to build a model for refactoring and quality documentation.

- Design and implement approaches leveraging template-based and deep-learning techniques to generate documentation for composite code changes.

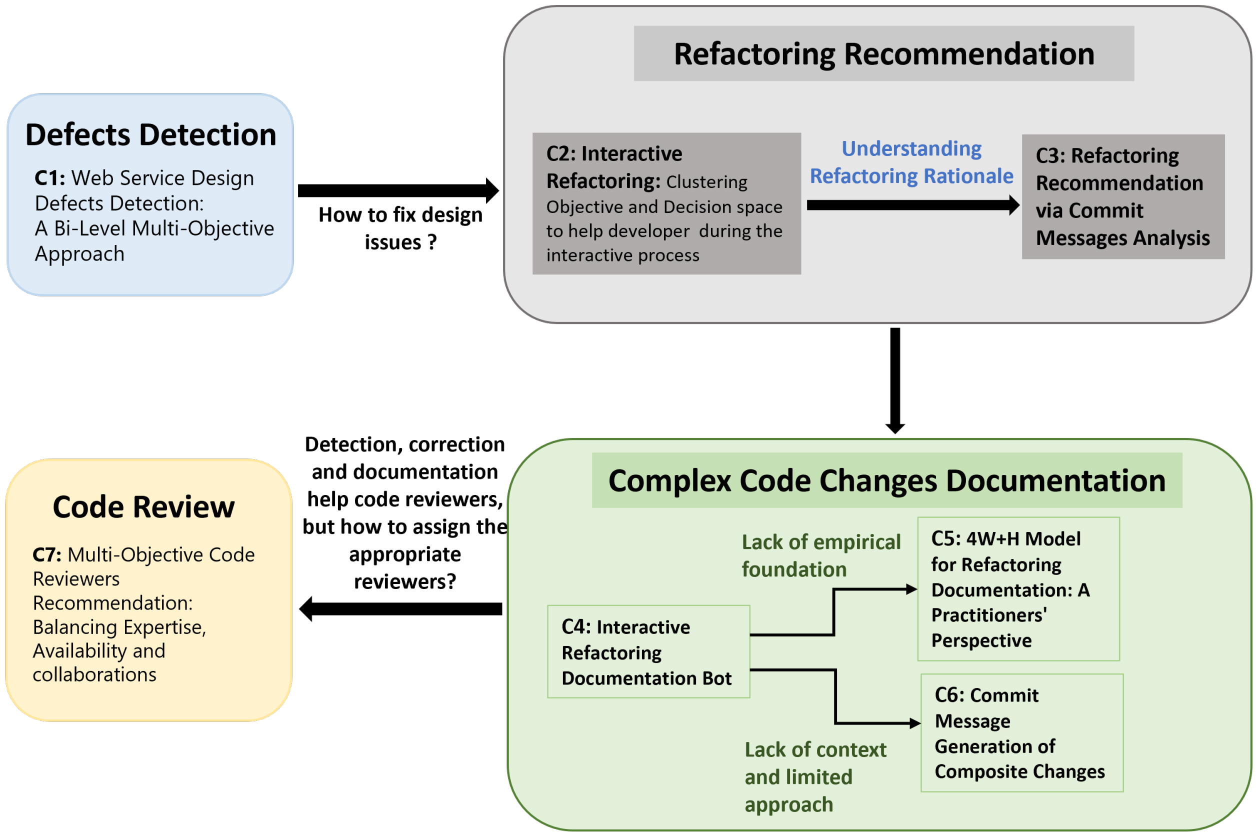

Defects Detection
C1: Web Service Design Defects Detection: A Bi-Level Multi-Objective Approach
How to fix design issues ?
Refactoring Recommendation
C2: Interactive Refactoring: Clustering Objective and Decision space to help developer during the interactive process
Understanding Refactoring Rationale
C3: Refactoring Recommendation via Commit Messages Analysis
Complex Code Changes Documentation
Lack of empirical foundation
C5: 4W+H Model for Refactoring Documentation: A Practitioners' Perspective
C4: Interactive Refactoring Documentation Bot
C6: Commit Message Generation of Composite Changes
Lack of context and limited approach
Detection, correction and documentation help code reviewers, but how to assign the appropriate reviewers?
Code Review
C7: Multi-Objective Code Reviewers Recommendation: Balancing Expertise, Availability and collaborations

Figure 1.1: Overview of the contributions of this thesis.

Figure 1.1 represents the different contributions of this thesis. The presented framework contains four main components. The first component is the **"defects detection"** where we published one research contribution. Second, we published two research contributions within the **"refactoring recommendation component"**. For the **"complex code changes documentation"** component, we published one research contributions and successfully submitted two contributions to a top journal. Finally, we published one contribution for **"code review component"**.

In the following, we will summarize the objectives of each contribution.

1.2.1 Contribution 1: Web Service Design Defects Detection

There are various challenges in the existing studies to address the detection of web service design defects. In fact, there was no use of quality of service metrics such as the response tine and availability in assisting the performance of a web services. Additionally, most of the existing tools were either based on a manual detection which is error-prone and requires high calibration effort to find a threshold for each metric; or automated tools that suffers from lack of data which impacts their efficiency because to provide efficient detection rules, we need a high number of interface design defect examples to feed to the model. To address the previously mentioned challenges, we designed a bi-level multi-objective optimization approach to enable the generation of antipattern examples that can improve the efficiency of detection rules for bad quality designs. The statistical analysis of our results, based on a data-set of 662 web services, confirms the efficiency of our approach in detecting web service antipatterns comparing to the current state of the art in terms of precision and recall. The multi-objective search formulation at both levels helped to diversify the generated artificial web service defects which produced better quality of detection rules. Furthermore, the combination of dynamic QoS attributes and structural information of web services improved the efficiency of the generated detection rules. A paper is published in the Information and Software Technology Journal [5]

Note that the adopted detection techniques for Object-Oriented (OO) designs and/or web services defects are almost similar. In this contribution, we were interested in generalizing the usefulness of a bi-level optimization and the importance of considering quality attributes of a software projects beyond the Object-Oriented context.

[5]Soumaya Rebai, Marouane Kessentini, Hanzhang Wang, Bruce R. Maxim, "*Web service design defects detection: A bi-level multi-objective approach*"

1.2.2 Contribution 2: Interactive Refactoring Recommendation

Antipatterns represent a source for refactoring opportunities. To fix these poor quality systems, various refactoring recommendation approaches were previously proposed (the approaches are described in details in section 2.3). The most significant challenge in the existing interactive refactoring recommendation approaches is related to the huge effort required from the user in order to evaluate and explore the refactoring solutions. To the best of our knowledge, there are no existing refactoring tools that enable the interactive exploration of both quality metrics and code locations during the refactoring process. Thus, we propose an interactive approach where we combine machine learning technique with multi-objective optimization to recommend more relevant refactoring solutions. Our proposed solution consists of not only clustering the solution at the objective space (quality metrics) but also at the decision space (code locations) and therefore we succeed to reduce the developer effort and the optimization process towards the developer's region of interest. A paper is accepted at the IEEE Transactions in Software Engineering journal [6]

1.2.3 Contribution 3: Refactoring Recommendation via Commit Messages Analysis

Although the results of our interactive clustering approach are promising and outperform the state-of-the-art approaches, we noticed that there are other resources such as commit messages that help improve the refactoring recommendation and reduce even more the effort of the developer in exploring the generated solutions. As a result we propose a refactoring recommendation solution via commit messages analysis where we combine NLP techniques with multi-objective optimization to detect better refactoring opportunities and then recommend more personalized refactoring recommendations. The results shows an outperformance over the state-of-the-art techniques. A paper is published in the Information and Software

[6]Soumaya Rebai, Vahid Alizadeh, Marouane Kessentini, Houcem Fehri, and Rick Kazman, "*Enabling Decision and Objective Space: Exploration for Interactive Multi-Objective Refactoring.*"

Technology Journal [7]

1.2.4 Contribution 4: Interactive Refactoring Documentation Bot

Our previous analysis of the commit messages for refactoring documentation and the promising results of our previous contributions put emphasis on the importance of having a good documentation in guiding the refactoring process and thus assessing and improving software quality. Additionally, several automated techniques for the generation and recommendation of documentation of diffs and atomic changes (e.g., [66, 74, 75, 67, 76, 77, 78]) have been recently proposed. However, most of the current development workflows/pipelines in industry are lacking tools/steps to document refactorings and quality changes/technical debt. As a result, we propose as a first attempt, a template-based refactoring documentation bot which helps documenting refactoring and non-functional changes while submitting new code changes. The feedback of practitioners show the usefulness and the importance of the bot in assisting developer while committing their changes. A paper is published in 19th International Working Conference on Source Code Analysis and Manipulation (SCAM) [8]

1.2.5 Contribution 5: 4W+H Model for Refactoring Documentation: A Practitioners' Perspective

To the best of our knowledge, there is no solid empirical foundation on what information is (or is not) useful to developers when documenting composite changes(e.g. refactorings) based on surveys with practitioners. Therefore, to further improve the documentation tools and bots, we were interested in exploring in more details the practitioners' perspective about the importance of automating refactoring documentation and what are the most important components to be documented. The results of the conducted survey report 5 important

[7]Soumaya Rebai, Marouane Kessentini, Vahid Alizadeh, Oussama Ben Sghaier, and Rick Kazman, "*Recommending Refactorings via Commit Message Analysis.*"

[8]Soumaya Rebai, Oussama Ben Sghaier, Vahid Alizadeh, Marouane Kessentini and Meriam Chater, "*Interactive Refactoring Documentation Bot.*"

components to be documented (Why, Where, What, When and How). A paper is submitted to the IEEE Transactions in Software Engineering journal [9]

1.2.6 Contribution 6: Commit Message Generation of Complex Code Changes

Most of the currently available techniques for code changes documentation are particularly suitable only when dealing with commits composed of a few *atomic* changes, i.e., operations where developers apply a set of disjointed changes, like additions/deletion of lines of code, to multiple files. However, they might not be as accurate when facing what we define as *composite* changes, namely, a set of conceptually related modifications that are intended to implement a unique high-level code change. Therefore, in another attempt towards improving the documentation of complex code changes, we first build a dataset composed of 50,000+ composite code changes pertaining to more than 7,000 open-source projects."Composite change" is a set of conceptually related atomic modifications forming a , e.g., a bug-fixing activity touching more files. Afterward, we propose and evaluate a new deep learning technique to generate commit messages for composite code changes based on an attentional encoder-decoder with two encoders and BERT embeddings. The model takes diff files and composite changes as two different input sources. Our results show that our technique overcomes the existing approaches in terms of BLEU score. A paper is submitted to the IEEE Transactions in Software Engineering journal [10]

1.2.7 Contribution 7: Code Reviewers Assignment

Code reviews is considered as the last step in every software project where the code needs to be reviewed before it goes into production. Our previous contributions focus mainly on detecting, fixing and documenting changes which significantly help and support the reviewers during the code review process. However, we were also interested in addressing the challenge

[9] Soumaya Rebai, Marouane Kessentini, Tushar Sharma, and Thiago Ferreira, "*4W+H Model for Refactoring Documentation: A Practitioners' Perspective.*"

[10] Soumaya Rebai, Siyuan Jiang, Marouane Kessentini, Weji Huang, and Fabio Palomba, "*Commit Message Generation of Composite Changes.*"

of assigning the appropriate reviewers to review certain code fragments. In fact, despite recent progress [79, 80] code reviews are still time-consuming, expensive, and complex involving a large amount of effort by managers, developers and reviewers. Thongtanunam et al. [81] found on four open source projects with 12 days as the average to approve a code change. The automated recommendation of peer code reviewers may help to reduce delays by finding the best reviewers who will then spend less time in reviewing the assigned files. Therefore, we propose a multi-objective reviewers assignment solution where we balance collaborations, expertise and availability. A paper is published in Automated Software Engineering journal [11]

1.3 Organization of the Dissertation

This thesis is organized as follows: Chapter II introduces the current state of the art and related works to this thesis. Chapter III presents multi-objective bi-level approach for web service design defects detection. Chapter IV describes our proposed approach to interactively recommend refactoring. Chapter V, discusses our proposed method to recommend refactoring via analyzing commit messages. We present our refactoring documentation bot to assist the developers documenting their code changes in chapter VI. Chapter VII describes our proposed model for refactoring documentation, constructed from practitioners' perspective. An approach for commit message generation of composite changes is presented in chapter VIII. We present our multi-objective approach for assigning code reviewers in chapter IX. Finally, a summary and future research directions are presented in X.

[11]Soumaya Rebai, Abderrahmen Amich, Somayeh Molaei, Marouane Kessentini and Rick Kazman, "*Multi-Objective Code Reviewer Recommendations: Balancing Expertise, Availability and Collaborations.*"

CHAPTER II

State of the Art

2.1 Introduction

In this chapter, we cover the necessary background information related to our work followed by an overview of existing studies.

2.2 Background

Quality attributes. QMOOD is a widely used quality model, based on the ISO 9126 product quality model [82]. We selected this model because it is a widely accepted quality model in industry and it has been validated based on hundreds of industrial projects[82, 83, 84, 33, 1]. Each quality attribute in QMOOD is defined using a combination of low-level metrics as detailed in Tables 2.1 and 2.2. The QMOOD model has been used in many studies [40, 85, 86] to estimate the effects of proposed refactoring solutions on software quality. QMOOD defines six high-level design quality attributes (reusability, flexibility, understandability, functionality, extendibility, and effectiveness) that can be calculated using 11 lower-level design metrics.

Commits and refactoring. Refactoring documentation has two major parts: pull requests for "high-level" refactorings [87] and commit messages for code-level refactorings. The individual commit messages describe refactorings applied by a developer. A refactoring process typically starts with a new branch. In this branch, each commit should correspond to

Table 2.1: QMOOD metrics description.

Design Metric	Design Property	Description
Design Size in Classes (*DSC*)	Design Size	Total number of classes in the design.
Number Of Hierarchies (*NOH*)	Hierarchies	Total number of "root" classes in the design (*count(MaxInheritenceTree (class)=0)*)
Average Number of Ancestors (*ANA*)	Abstraction	Average number of classes in the inheritance tree for each class.
Direct Access Metric (*DAM*)	Encapsulation	Ratio of the number of private and protected attributes to the total number of attributes in a class.
Direct Class Coupling (*DCC*)	Coupling	Number of other classes a class relates to, either through a shared attribute or a parameter in a method.
Cohesion Among Methods of class (*CAMC*)	Cohesion	Measure of how related methods are in a class in terms of used parameters. It can also be computed by: *1 − LackOfCohesionOfMethods()*
Measure Of Aggregation (*MOA*)	Composition	Count of number of attributes whose type is user defined class(es).
Measure of Functional Abstraction (*MFA*)	Inheritance	Ratio of the number of inherited methods per the total number of methods within a class.
Number of Polymorphic Methods (*NOP*)	Polymorphism	Any method that can be used by a class and its descendants. Counts of the number of methods in a class excluding private, static and final ones.
Class Interface Size (*CIS*)	Messaging	Number of public methods in class.
Number of Methods (*NOM*)	Complexity	Number of methods declared in a class.

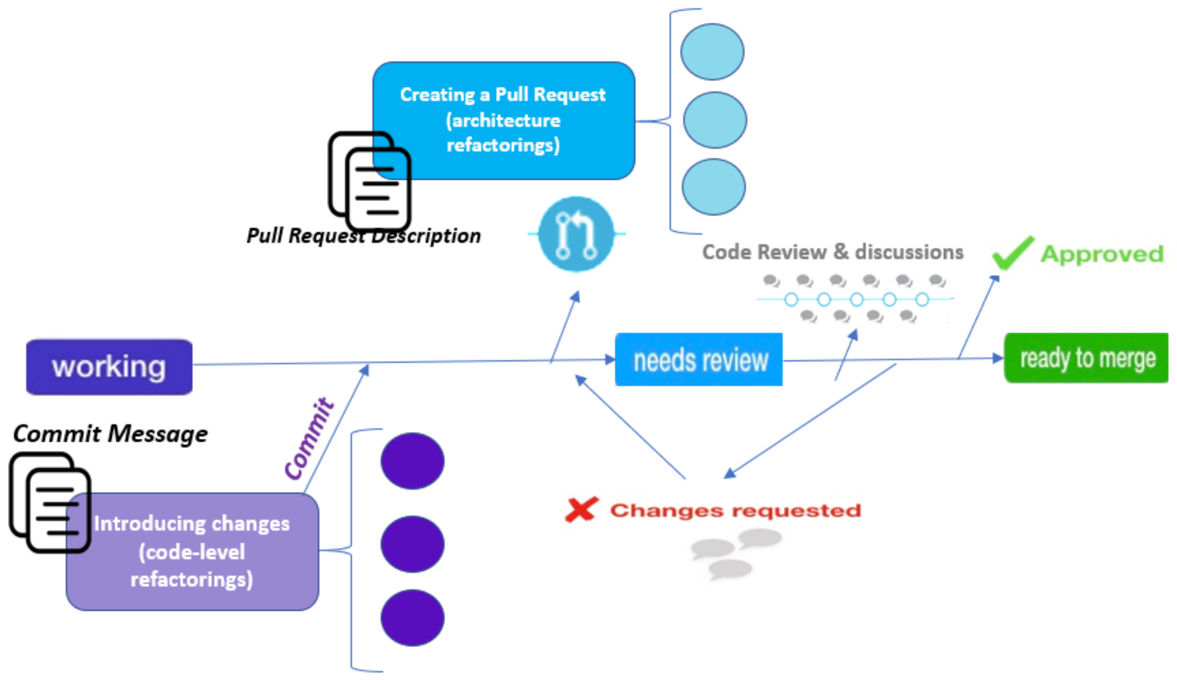
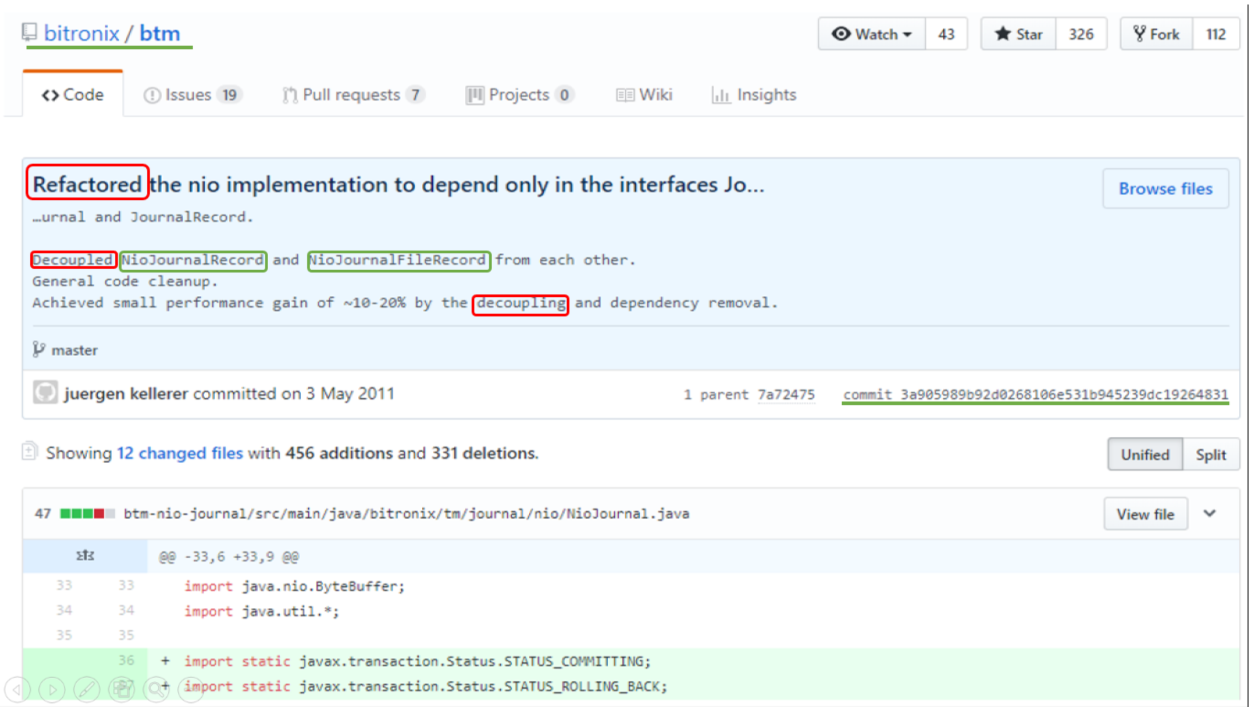

Creating a Pull Request (architecture refactorings)
Pull Request Description
Code Review & discussions
Approved
working
needs review
ready to merge
Commit Message
Commit
Introducing changes (code-level refactorings)
Changes requested

Figure 2.1: An architecture refactoring process in a version-control repository

Table 2.2: Quality attributes and their 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 a design's ability 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.
	$0.2 * Abstraction + 0.2 * Encapsulation + 0.2 * Composition + 0.2 * Inheritance + 0.2 * Polymorphism$



bitronix / btm

Watch 43 Star 326 Fork 112

Code Issues 19 Pull requests 7 Projects 0 Wiki Insights

Refactored the nio implementation to depend only in the interfaces Jo…

```
…urnal and JournalRecord.

Decoupled NioJournalRecord and NioJournalFileRecord from each other.
General code cleanup.
Achieved small performance gain of ~10-20% by the decoupling and dependency removal.
```

Browse files

master

juergen kellerer committed on 3 May 2011

1 parent 7a72475 commit 3a905989b92d0268106e531b945239dc19264831

Showing 12 changed files with 456 additions and 331 deletions.

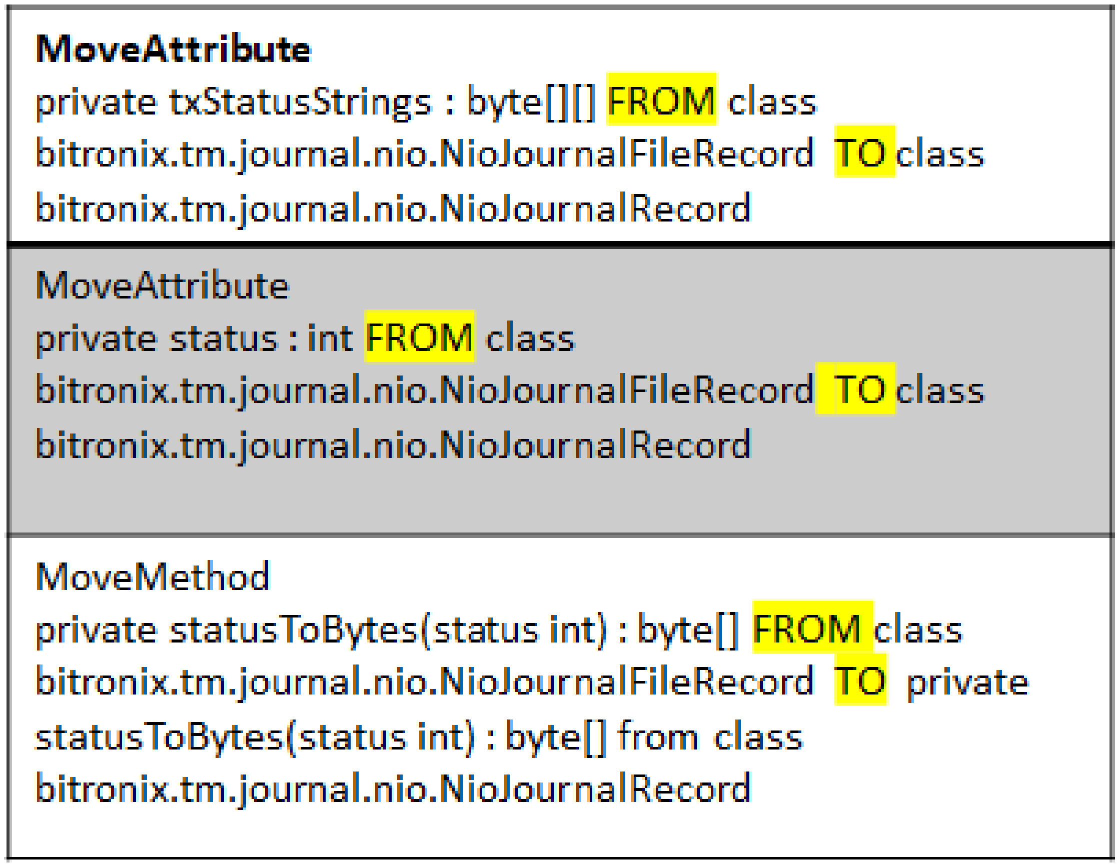
Unified Split

47 btm-nio-journal/src/main/java/bitronix/tm/journal/nio/NioJournal.java View file

```
@@ -33,6 +33,9 @@
   import java.nio.ByteBuffer;
   import java.util.*;

+  import static javax.transaction.Status.STATUS_COMMITTING;
+  import static javax.transaction.Status.STATUS_ROLLING_BACK;
```

Figure 2.2: An example commit from the "btm" project.

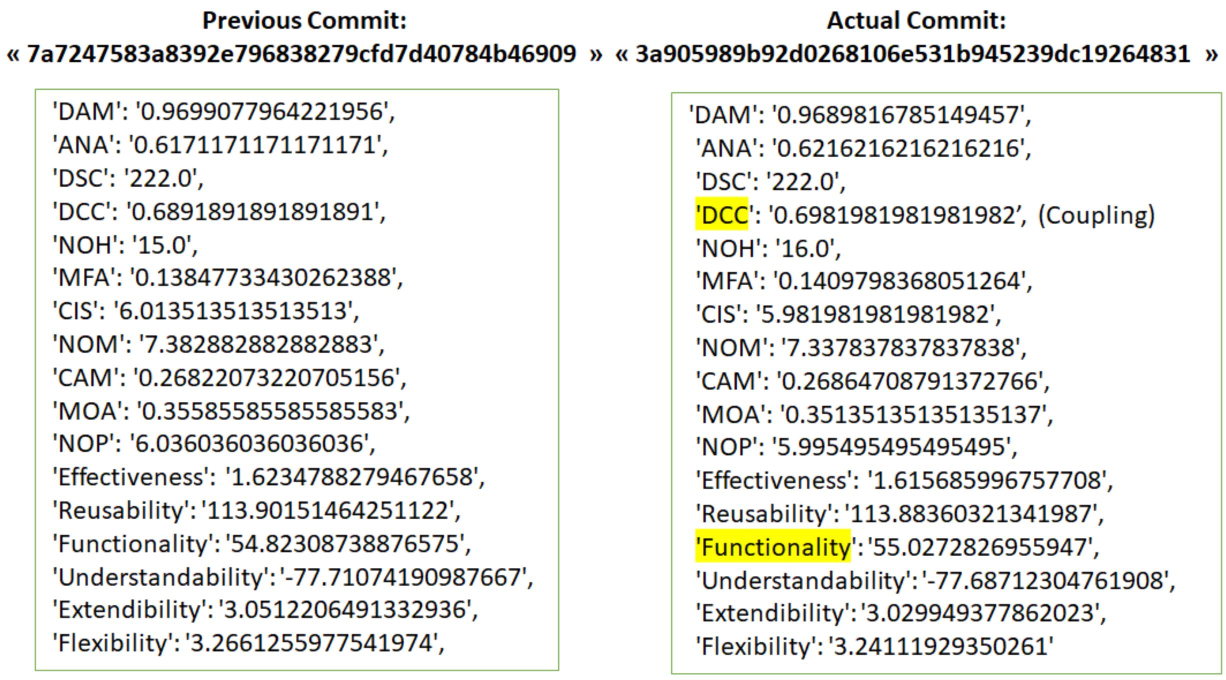


MoveAttribute
private txStatusStrings : byte[][] FROM class bitronix.tm.journal.nio.NioJournalFileRecord TO class bitronix.tm.journal.nio.NioJournalRecord

MoveAttribute
private status : int FROM class bitronix.tm.journal.nio.NioJournalFileRecord TO class bitronix.tm.journal.nio.NioJournalRecord

MoveMethod
private statusToBytes(status int) : byte[] FROM class bitronix.tm.journal.nio.NioJournalFileRecord TO private statusToBytes(status int) : byte[] from class bitronix.tm.journal.nio.NioJournalRecord

Figure 2.3: The list of refactorings applied in the commit.


Previous Commit:
« 7a7247583a8392e796838279cfd7d40784b46909 »

```
'DAM': '0.9699077964221956',
'ANA': '0.6171171171171171',
'DSC': '222.0',
'DCC': '0.6891891891891891',
'NOH': '15.0',
'MFA': '0.13847733430262388',
'CIS': '6.013513513513513',
'NOM': '7.382882882882883',
'CAM': '0.26822073220705156',
'MOA': '0.35585585585585583',
'NOP': '6.036036036036036',
'Effectiveness': '1.6234788279467658',
'Reusability':'113.90151464251122',
'Functionality':'54.82308738876575',
'Understandability':'-77.71074190987667',
'Extendibility':'3.0512206491332936',
'Flexibility':'3.2661255977541974',
```

Actual Commit:
« 3a905989b92d0268106e531b945239dc19264831 »

```
'DAM': '0.9689816785149457',
'ANA': '0.6216216216216216',
'DSC': '222.0',
'DCC': '0.6981981981981982', (Coupling)
'NOH': '16.0',
'MFA': '0.1409798368051264',
'CIS': '5.981981981981982',
'NOM': '7.337837837837838',
'CAM': '0.26864708791372766',
'MOA': '0.35135135135135137',
'NOP': '5.995495495495495',
'Effectiveness': '1.615685996757708',
'Reusability':'113.88360321341987',
'Functionality':'55.0272826955947',
'Understandability':'-77.68712304761908',
'Extendibility':'3.029949377862023',
'Flexibility':'3.24111929350261'
```

Figure 2.4: The quality metric changes in the commit.

a code-level refactoring. After developers commit all the code-level refactorings (i.e., finish the refactoring process), developers make a pull request in which they write a description of the overall refactoring. If the refactorings are accepted, the branch is merged into the master branch. Figure 2.1 shows the overall architecture refactoring process in a version-control repository. The list of refactoring types that can be supported by our research tools are described in Table 2.3.

Figure 2.2 shows an example of a commit extracted from an open source project. The refactorings applied by the developers are summarized in Figure 2.3, and the changes in the coupling (DCC) metric can be seen in Figure 2.4. Of course, refactoring rarely happen in isolation and most of commits and pull-requests contain a sequence of refactorings as described in the example Figure 2.3 that shows a sequence of three refactorings applied in one commit.

Review Process A code review includes all the interactions between the submitter of a pull-request and one or more reviewers of that change including comments on the code and discussions with reviewers. The owner is the programmer making the changes to the

Table 2.3: 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 Method/Class/Field	Changes the name of a code element.

code and then submitting the review request. A peer reviewer is a developer assigned to contribute in reviewing the set of code changes. These reviewers write review comments as feedback to the owner about the introduced changes.

Figure 2.5 shows the *code review* process in a version-control repository. A code review process starts with a new branch (①). In this new branch, each commit should correspond to a code-level change (②). After developers commit all the code-level changes, developers make a pull request, in which they write a description of the code changes (③). After a pull request has been sent out, it appears in the list of pull requests for the project in question,

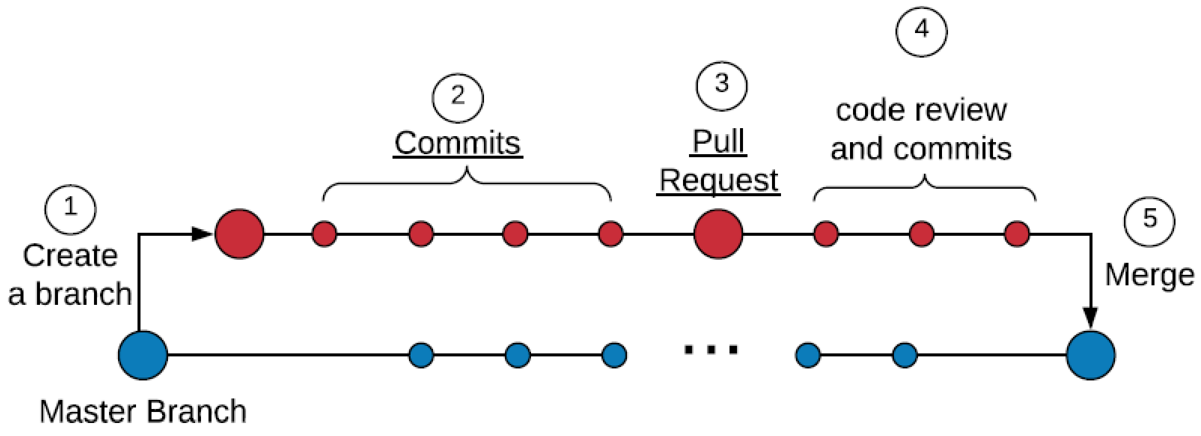


Figure 2.5: A summary of the code review process.

visible to anyone who can see the project. Then, other collaborators can check the changes made in the branch and discuss the changes (code reviews ④). During the code review, developers may make more changes to the branch. Finally, if the collaborators accept these code changes, this branch is merged into the master branch (⑤).

Figure 9.1 shows one example of code reviews where many possible reviewers can be assigned to review the changes. Thus, dealing with a large number of possible reviewers for multiple pull requests is a management problem which is under-studied in the research literature. This management process requires handling multiple competing criteria including expertise, availability and previous collaborations with the owners and reviewers.

2.3 Related Work

2.3.1 Software Changes Documentation

Empirical Studies. Several existing studies employ techniques such as interviews or surveys to investigate software documentation from different perspectives (e.g., quality, benefits, and usage.), as described in Table 2.4. The table shows that Forward et al. [88] concluded from surveying 48 software professionals that documentation tools should be improved in order to reduce the effort required for documentation maintenance, and that we

Table 2.4: Summary of previous empirical studies on Software Documentation

Paper	Main Focus	Summary
Forward and lethrbidge [88]	Documentation tools	The authors ran a survey with 48 participants to conclude that new technologies should be used to enhance the automation of documentations tools and their maintenance.
Kajko-Mattsson [47]	Documentation within corrective maintenance	The authors conducted an exploratory study within 18 software organizations in Sweden. They concluded that documentation within corrective maintenance is very neglected
De Souza et al. [44]	Software maintenance and documentation	The authors ran a survey with 76 software maintainers where they established what documentation artifacts are the most useful for software maintenance.
Chen and Huang [89]	Software maintainability and documentation quality	After surveying 137 practitioners, the authors conclude that the high severity documentation quality problems for software maintenance are: untrustworthiness, incompleteness or inexistence, lack of traceability.
Robillard [90]	Documentation as an obstacle for learning APIss	The author interviewed 12 developers and surveyed other 83 at Microsoft to find out that the main API learning obstacles are: resources for learning (documentation, examples, etc.), API structure, Background, Technical environment, Process.
Dagenais and Robillard [91]	Documentation evolution	The authors surveyed 22 practitioners to find that documentation could improve the code quality and constant interaction with the projects' community positively impacted the documentation.
Robillard and Deline [92]	Documentation as an obstacle for learning APIs	The authors conducted a research study with 440 professional developers at Microsoft to conclude that the main obstacles faced by developers while learning new APIs are closely related to API documentation.
Garousi et al. [93, 94]	Documentation usage and quality	The authors proposed a hybrid approach that consists of both quantitative and qualitative methods to evaluate software documentation usage and quality.
Plösch et al. [95]	Documentation quality	The authors conducted a survey with 88 experts to prove the need for an automated tool to support software documentation quality. The most important documentation quality attributes are accuracy, clarity, undertandability, consistency, readability and structuredness.
Uddin and Robillard [96]	Common documentation problems	The authors surveyed 323 IBM software professionals to investigate the manifestation of API documentation problems in practice. Their main finding is that developers prioritize content-related problems over presentation problems.
Sohan et al. [97]	API documentation	The authors conducted a study with 26 software engineers to conclude that REST API client developers face productivity with using correct data types, data formats, required HTTP headers and request body when documentation lacks usage examples.
Eman et al. [97]	Refactoring in commit messages	The authors proposed a preliminary work to quantitatively check if developers self-admit refactorings in commit messages and the keywords used for this purpose.
Safwan et al. [98]	Decomposing the rationale for code changes to help in documentation	The authors interviewed 20 software developers and surveyed 26 additional developers to decompose the rationale of code commits and understand practitioners' experience with the identified components.
Emad Aghajani et al. [99]	Documentation classifications	The authors surveyed 124 participants to classify documentation artifacts based on their importance given a specific task(refactoring, fixing bugs,etc...)

are still lacking a clear understanding of the decomposition of documentation based on the stockholders' needs. Safwan et al. [98] conducted interviews and surveys to understand the documentation of the rationale of code changes by practitioners. However, the focus on the empirical is just related to the documentation of why developers introduced the code changes to decompose these reasons into components. Eman et al. [100] proposed a preliminary work to quantitatively check if developers self-admit refactorings in commit messages and the used keywords for this purpose. However, they did not conduct surveys to provide an empirical foundation for refactoring documentation. Zhi et al. [101] described a mapping study about the quality and importance of software documentation and concluded that there is a need for large-scale empirical studies and collaborations with industry to better understand the way that practitioners document code changes. Our study is within this direction with a focus on complex code changes such as refactoring.

While many studies focused on the importance, usage, and quality of documentation spanning the software life-cycle, none of them addressed refactoring/composite changes documentation.

Tools. In addition to the empirical studies, existing research related to our work focused on building tools to automatically generate documentation for functional and atomic changes based on the diffs. We classify them into three categories: *commit messages generation*, *pull request descriptions generation* and *source code summarization*. Another related research line proposed an empirical foundation for software documentation and its importance through surveys [88, 44].

Table 2.5 summarizes some of the previously mentioned studies related to commit message generation and shows the missing features that are not yet addressed in the existing research. The last column clearly state that the existing work lacks the focus on documentation for refactoring and non-functional changes. The *input sources* column shows that most of the study related to commit generations used the same input source which mainly consists of the diff files.

2.3.1.1 Commit Messages Generation

Researchers have adopted three main techniques in existing studies to automatically generate commit messages: template-based, search-based, and deep learning-based methods. Template-based approaches [102, 103] including Delta-Doc [78] and ChangeScribe [76, 77] extract information of a commit and generate messages for the changes following a specific predefined template. The commit messages generated by these approaches can be lengthy and visibly machine-generated messages. In contrast, search-based approaches [64, 65, 66] generate commit messages that perceive human-written messages. The search-based approaches find the most similar commits in the history and reuse their commit messages as a generated message for the new commit. Search-based techniques work for the code changes that repeat in software repositories, such as updates of API dependencies. However, they may fail for new types of code changes. NNGen [66] analyzed the performance of neural machine translation (NMT) [67] for commit message generation, and showed that a simpler and faster approach based on nearest neighbor outperforms relatively complex deep learning methods.

Recently, *deep learning techniques* attracted researchers' attention to generate commit messages [67, 68, 66]. Siyan et al. [67] proposed a NMT-based approach to "translate" diffs into commit messages. Their model consists of a sequence-to-sequence (Seq2Seq) recurrent neural network. The model fails to perform well in some cases because they trained their model using a noisy and not well-cleaned data.

2.3.1.2 Pull Request Description Generation and Source Code Summarization

Liu et al. [74] proposed a deep-learning Seq2Seq approach to generate pull-requests descriptions. They consider pull-request description as a summary of commits messages generated since the last pull-request. Thus, they address the problem of pull-request descriptions generation as a text summarization problem for mainly functional changes. Recently, Rebai et al. [75] built a bot that enables the documentation of applied refactorings and quality attributes changes as a pull-request description in a continuous integration environment using the template-based technique. However, the documentation bot suffers from some limitations such as the generated pull request description can be lengthy containing limited information based on the pre-defined template.

Source code summarization techniques [104] are used to generate documentation for source code fragments. These techniques can also be used to summarize code changes in commits [104]. Recent studies proposed tools that generate natural language summaries of source code (e.g. Java methods and classes) [105, 106, 107, 108, 75, 17, 109, 110]. For instance, Sridhara et al. [111] proposed a tool to generate summaries for Java methods following two complementary steps. First, they extracted important information from Java methods. Second, they expressed the extracted content in natural language following pre-defined templates. The authors extended their work to automatically generate description for high-level actions within methods. Other existing work [112, 113] leverage text retrieval techniques to generate source code summaries. For instance, Haiduc et al. [113] addressed the problem of generating source code entities descriptions using Latent Semantic Indexing (LSI) [114]. Deep learning models are also used by researchers in addressing source code summarization. For example, Iyer et al. [115] have proposed a framework, named Code-NN, to generate summaries for C# and SQL code using a NMT method.

2.3.2 Refactoring Recommendation

2.3.2.1 Manual Refactoring

We start, this section, by summarizing existing manual approaches for software refactoring. In Fowler's book [9] a non-exhaustive list of low-level design problems in source code has been defined. For each type of code smell, a list of possible refactorings is suggested that can be applied by the developers. Du Bois *et al.* [24] start from the hypothesis that refactoring opportunities correspond to those that improve cohesion and coupling metrics, and use this to perform an optimal distribution of features over classes. They analyze how refactorings manipulate coupling and cohesion metrics, and how to identify refactoring opportunities that improve these metrics. However, this approach is limited to only certain refactoring types and a small number of quality metrics. Murphy-Hill *et al.* [116, 117] proposed several techniques and empirical studies to support refactoring activities. In [117, 118], the authors proposed new tools to assist software developers in applying refactoring such as selection as-

Table 2.5: Related work in automated documentation generation

Paper	**Technique used**	**Category**	**Granularity**	**Input sources**	Refactoring documentation
DeltaDoc; Automatically documenting program changes [78]	Template-based approach	Source code summarization	Commit level	Two versions of a program	No
Changescribe; A tool for automatically generating commit messages [76, 77]	Template-based approach	Source code summarization	Commit level	Source code and ASTs (Abstract Syntax Trees)	No
Automatically generating commit messages from diffs using neural machine translation [67]	Deep learning—NMT	Commit Message Generation	Commit level	Diff files	No
Neural machine-translation-based commit message generation: How far are we? [66]	Search-based method—Nearest Neighbors	Commit message generation	Commit level	Diff files	No
Automatic Generation of Pull Request [74] Decriptions	Deep Learning: Encoder-Decoder Model	Pull requests description generation	Pull request level	Commit messages and source code comments	No
Interactive Refactoring Documentation Bot [75]	Template-based approach	Pull requests description generation	Pull request level	Commit source code refactoring applied and quality attributes changes	Yes

sistant, box view, and refactoring annotation based on structural information and program analysis techniques.

Recently, Ge and Murphy-Hill [119] have proposed a new refactoring tool called GhostFactor that allows the developer to transform code manually, but checks the correctness of the transformation automatically. BeneFactor [26] and WitchDoctor [120] can detect manual refactorings and then complete them automatically. Tahvildari *et al.* [121] also propose a framework of object-oriented metrics used to suggest to the software developer refactoring opportunities to improve the quality of an object-oriented legacy system. Dig [122] proposes an interactive refactoring technique to improve the parallelism of software systems. However, the proposed approach did not consider learning from the developers' feedback and focused on making programs more parallel. Other contributions are based on rules that can be expressed as assertions (invariants, pre- and post-conditions). All these techniques are more concerned around the correctness of manually applied refactorings rather than interactive recommendations.

The use of invariants has been proposed to detect parts of the program that require refactoring [123]. In addition, Opdyke [124] has proposed the definition and use of pre- and post-conditions with invariants to preserve the behavior of the software when applying refactorings. Hence, behavior preservation is based on the verification/satisfaction of a set of pre- and post-condition. All these conditions are expressed as first-order logic constraints expressed over the elements of the program.

To summarize, manual refactoring is a tedious task for developers that involves exploring the software system to find the best refactoring solution that improves the quality of the software and fix design defects.

2.3.2.2 Automated Refactoring

To automate refactoring activities, new approaches have been proposed. JDeodorant [125] is an automated refactoring tool implemented as an Eclipse plug-in that identifies certain types of design defect using quality metrics and then proposes a list of refactoring strategies to fix them. Search-based techniques [126] are widely studied to automate software refactoring and consider it as an optimization problem, where the goal is to improve the design quality of a system based mainly on a set of software metrics. The majority of existing work combines several metrics in a single fitness function to find the best sequence of refactorings. Seng *et al.* [127] have proposed a single-objective optimization approach using a genetic algorithm to suggest a list of refactorings to improve software quality. The work of O'Keeffe *et al.* [128] uses various local search-based techniques such as hill climbing and simulated annealing to provide an automated refactoring support. They use the QMOOD metrics suite [129] to evaluate the improvement in quality.

Kessentini *et al.* [130] have proposed single-objective combinatorial optimization using a genetic algorithm to find the best sequence of refactoring operations that improve the quality of the code by minimizing as much as possible the number of design defects detected in the source code. Kilic *et al.* [131] explore the use of a variety of population-based approaches to search-based parallel refactoring, finding that local beam search could find the best solutions. Harman *et al.* [132] have proposed a search-based approach using Pareto optimality that combines two quality metrics, CBO (coupling between objects) and SDMPC (standard deviation of methods per class), in two separate fitness functions. Ouni *et al.* [133] proposed also a multi-objective refactoring formulation that generates solutions to fix code smells. Ó Cinnéide *et al.* [134] have proposed a multi-objective search-based refactoring to conduct an empirical investigation to assess some structural metrics and to explore relationships between them. They have used a variety of search techniques (Pareto-optimal search, semi-random search) guided by a set of cohesion metrics.

The majority of existing multi-objective refactoring techniques propose as output a set of non-dominated refactoring solutions (the Pareto front) that find a good trade-off between the considered maintainability objectives. This leaves it to the software developers to select the best solution from a set of possible refactoring solutions, which can be a challenging task as it is not natural for developers to express their preferences in terms of a fitness functions value. Thus, the exploration of the Pareto front is still performed manually, which limits the use of multi-objective search techniques to address software engineering problems. An intelligent exploration of the Pareto front is required to expand the applicability of multi-objective techniques for search-based software engineering problems.

In summary, developers should accept the entire refactoring solution and existing tools do not provide the flexibility to adapt the suggested solution in existing fully-automated refactoring techniques. Furthermore, existing automated refactoring tools execute the whole algorithm again to suggest new refactorings after a number of code changes are introduced by developers, rather than simply trying to update the proposed solutions based on the new code changes. While automation is important, it is essential to understand the points at which human oversight, intervention, and decision-making should impact on automation. Human developers might reject changes made by any automated programming technique. Especially if they feel that they have little control, there will be a natural reluctance to trust and use the automated refactoring tool [135].

2.3.2.3 Interactive Refactoring

Interactive techniques have been generally introduced in the literature of Search-Based Software Engineering and especially in the area of software modularization. Hall *et al.* [136] treated software modularization as a constraint satisfaction problem. The idea of this work is to provide a baseline distribution of software elements using good design principles (e.g. minimal coupling and maximal cohesion) that will be refined by a set of corrections introduced interactively by the designer.

The approach, called SUMO (Supervised Re-modularization), consists of iteratively feeding domain knowledge into the remodularization process. The process is performed by the designer in terms of constraints that can be introduced to refine the current modularizations. Initially, the system begins with generating a module dependency graph from an input system. This dependency is based on the correlation between software elements (coupling between methods, shared attributes etc.). Possible modularizations are then generated from the graph using multiple simulated authoritative decompositions. Then, using a clustering technique called Bunch, an initial set of clusters is generated that serves as an input to SUMO.

The SUMO algorithm provides a hypothesized modularization to the user, who will agree with some relations, and disagree with others. The user's corrections are then integrated into the modularization process, to generate a more satisfactory modularization. The SUMO algorithm does not necessarily rely on clustering techniques, but it can benefit from their output as a starting point for its refinement process.

Bavota *et al.* [137] presented the adoption of single objective interactive genetic algorithms in software re-modularization process. The main idea is to incorporate the user in the evaluation of the generated remodularizations. Interactive Genetic Algorithms (IGAs) extend the Classic Genetic Algorithms (GAs) by partially or entirely involving the user in the determination of the solution's fitness function. The basic idea of the Interactive GA (IGA) is to periodically add a constraint to the GA such that some specific components shall be put in a given cluster among those created so far. Initially, the IGA evolves similarly to the non-interactive GAs.

After a user-defined set of iterations, the individual with the highest fitness value is selected from the population set (in the case of single-objective GA) or from the first front (in the case of multi-objective GA) and presented to the user. After analyzing the current modularization, the user provides feedback in terms of constraints dictating for example, that a specific element needs to be in the same cluster as another one. Although user feedback is important in guaranteeing convergence, it is essential not to overload the user by asking for a decision about all the current relationships between elements, especially for a large system.

Overall, the above existing studies of interactive remodularization are limited to few types of refactoring such as moving classes between packages and splitting packages. Furthermore, the interaction mechanism is based on the manual evaluation of proposed remodularization solutions which could be a time-consuming process. The proposed interactive remdoularization techniques are also based on a mono-objective algorithm and did not consider multiple objectives when evaluating the solutions. A recent study [138] extended our previous work [139] to propose an interactive search based approach for refactoring recommendations. The developers have to specify a desired design at the architecture level then the proposed approach try to find the relevant refactorings that can generate a similar design to the expected one. In our work, we do not consider the use of a desired design, thus developers are not required to manually modify the current architecture of the system to get refactoring recommendations. Furthermore, developers maybe interested to change the architecture mainly when they want to introduce an extensive number of refactorings that radically change the architecture to support new features.

Several possible levels of interaction are not considered by existing refactoring techniques. It is easy for developers to identify large classes or long methods that should be refactored, but they find it is difficult, in general, to locate a target class when applying a move method refactoring [140]. In addition, existing refactoring tools do not update their recommended refactoring solutions based on the software developer's feedback such as accepting, modifying or rejecting certain refactoring operations.

Furthurmore, None of the above interactive studies considered reducing the interaction effort with developers which is an important step to improve the applicability of refactoring tools as highlighted in the survey with developers.

To address the above-mentioned limitations, we proposed in this thesis, a new way for software developers to refactor their software systems as a sequence of transformations based on different levels of interaction, implicit exploration of non-dominated refactoring solutions and dynamic adaptive ranking of the suggested refactorings.

2.3.2.4 Search Based Software Refactoring

Search-based techniques [126] are widely studied to automate software refactoring where the goal is to improve the design quality of a system based mainly on a set of software metrics. The majority of existing work combines several metrics in a single fitness function to find the best sequence of refactorings. Seng et al. [141] have proposed a single-objective optimization approach using a genetic algorithm to suggest a list of refactorings to improve software quality. The work of O'Keeffe et al. [128] uses various local search-based techniques such as hill climbing and simulated annealing to provide an automated refactoring support. They use the QMOOD metrics suite [129] to evaluate the improvement in quality. Kessentini et al. [130] have proposed single-objective combinatorial optimization using a genetic algorithm to find the best sequence of refactoring operations that improve the quality of the code by minimizing as much as possible the number of design defects detected in the source code. Kilic et al. [131] explore the use of a variety of population-based approaches to search-based parallel refactoring, finding that local beam search could find the best solutions. Harman et al. [132] have proposed a search-based approach using Pareto optimality that combines two quality metrics, CBO (coupling between objects) and SDMPC (standard deviation of methods per class), in two separate fitness functions. Ouni et al. [133] proposed also a multi-objective refactoring formulation that generates solutions to fix code smells. Ó Cinnéide et al. [134] have proposed a multi-objective search-based refactoring to conduct an empirical investigation to assess some structural metrics and to explore relationships between them. They have used a variety of search techniques (Pareto-optimal search, semi-random search) guided by a set of cohesion metrics.

The majority of existing multi-objective refactoring techniques propose as output a set of non-dominated refactoring solutions (the Pareto front) that find a good trade-off between the considered maintainability objectives. This leaves it to the software developers to select the best solution from a set of possible refactoring solutions, which can be a challenging task as it is not natural for developers to express their preferences in terms of a fitness functions value. Thus, the exploration of the Pareto front is still performed manually, which limits the use of multi-objective search techniques to address software engineering problems. An intelligent exploration of the Pareto front is required to expand the applicability of multi-objective techniques for search-based software engineering problems as addressed in this thesis.

2.3.3 Code Reviews

Expertise has been the most important factor in the studies proposing peer reviewer recommendation. Zanjani et al. found that expertise changes over time and thus both frequency and recency of reviews must be accounted for to find the most appropriate reviewers.

Therefore their approach builds a reviewer expertise model, generated from past reviews, that combines a quantification of review comments and their recency [80].

Balachandran et al. first suggested to use the Review Bot tool, as a recommendation system to reduce human effort and improve review quality by finding source code issues, which need to be addressed, but could be missed during reviewer inspection. The bot can review the code by integrating the static analysis of the source code [71]. The bot, as part of a review process, is able to recommend the most appropriate human reviewer. In cases when the project has been modified frequently and there is a history of the changes for the source code, the bot is a suitable solution. However, Patanamon et al. [142] showed that the Review Bot's algorithm had poor performance on other projects with no or little change in their files due to the lack of history in line-by-line source code. In the same work, they introduced the idea of using file location (but not content) as an indicator for similarity of reviews. Their reviewer recommender approach, called File Path Similarity (FPS), implementing this idea, assumes that files that are located in similar file paths would be managed and reviewed by similarly experienced expert code reviewers. To improve their previous idea, Patanamon et al. [81] introduced REVFINDER, a file location-based code-reviewer recommendation approach. REVFINDER uses the similarity of previously reviewed file paths to recommend an appropriate code-reviewer. However, they did not consider the reviewer's work load and availability.

Xia et al. [143] used bug reports and developer information to recommend developers to resolve bugs. However, the most notable limitation of these works is that the socio-technical aspect of the code review process is not considered.

Several other studies focused on human factors and socio-technical aspects of code review. Cohen et. al., in [144] discuss that code review is a complex process involving both social and personal aspects. Fagan [145], to ensure the quality of software, introduced software inspection as a systematic peer review activity. Other studies [146, 147, 148, 148, 149, 150, 151, 152] motivate the need for a peer review recommendation system, considering the volunteer nature of open-source software (OSS) developers and the peer review structure, suggest that different human factors influence the OSS peer review. Baysal et al. conducted several studies [153, 154, 151] to explore the relationships between a set of personal and social factors and code review.

Bosu et al. [149] conducted a survey on four aspects of peer impression formation: trust, reliability, perception of expertise, and friendship. They concluded that there is a high level of trust, reliability, perception of expertise, and friendship between OSS peers who have participated in code review for a period of time. In another survey on how social interaction networks influence peer impressions formation [155], they found that code review interactions have the most favorable characteristics to support impression formation among OSS participants.

Based on search based software engineering [156, 157, 158, 159, 18], Ouni et. al [79] combined both aspects in their proposed approach, called RevRec, to provide decision-making support for code change submitters and reviewer assigners to identify the most appropriate peer reviewers for code changes. RevRec uses a genetic algorithm to assign reviewers to review a code change based on expertise and history of collaboration. Their single objective optimization approach aims to find appropriate reviewers for a given patch based on the reviewer's expertise with the submitted patch files, and the reviewer's prior collaborations with the review request submitter. Although this is the closest work in the literature to our proposed approach, our work differs from their work in a few ways: their solution representation determines if any of the reviewers are recommended to review a single file, therefore in cases when there are more files to review, let say k files, then the single objective optimization must run k times independently from each other which may not necessarily match the reality of the task. Our solution representation recommends reviewers for all the files that need to be reviewed at the same time. Furthermore, they do not consider the current workload of the reviewers and when they might be available to review the current files that match their expertise. In our method, we account for a reviewer's availability and we provide a ranking for the recommended reviewers so that if one reviewer is the best match, but busy with other work, we do not recommend the reviewer as the first choice for reviewing that file. This will decrease the overall delay in the system for files to get reviewed. Additionally, to capture the complexity of peer code review task, we formulate the problem as interaction among the competing objectives of expertise, availability and history of collaborations.

CHAPTER III

Web Service Design Defects Detection

3.1 Introduction

Service-Oriented Computing (Service-Oriented Computings (SOCs)) has emerged as an evolutionary paradigm that is changing the way software applications are implemented, deployed, and delivered to help industry meet their ever-more-complex challenges [5]. Nowadays, SOC is becoming widely accepted in industry such as FedEx [1], Dropbox [2], Google Maps [3], eBay[4], etc. The massive adoption of this paradigm and its popularity are mainly due to the offered reusability, modularity, flexibility, and scalability [6]. SOC utilizes services which are independent and portable program units as fundamental elements to support rapid, low cost development of heterogeneous and distributed systems [7].

Any successful deployed web services evolve over time to meet the new changes in the requirements and/or to fix bugs. The continuous changes and evolution of web services may create poor and bad design practices which are generally called ”antipatterns” that can impact the performance and usability of the web service [8]. Maintaining a good design quality is critical but it is excessively expensive both in time and resources for the service providers.

To detect web service antipatterns, most of the existing studies consider only the interface

[1]http://www.fedex.com/ca_english/businesstools/webservices
[2]https://www.dropbox.com/developers/core
[3]developers.google.com/maps/documentation/webservices
[4]https://developer.ebay.com/docs

or code-level metrics of bad-designed web services [160, 161, 162, 157]. Therefore, they enable developers to evaluate the quality of their service using mainly static information extracted from the implementation details of the interface and the services, such as coupling, cohesion, and number of operations. However, it is widely known that the quality of service metrics such as the response time and availability play a significant role in evaluating the overall performance of a service-based system. Furthermore, most of these studies [160, 161, 162, 157] are based on declarative rule specification. The detection rules are manually defined to identify the key symptoms that characterize an interface design defect using combinations of mainly quantitative metrics. For each possible interface design defect, rules that are expressed in terms of metric combinations need high calibration efforts to find the right threshold value for each metric. In addition, the translation of the symptoms into rules is not obvious because several symptoms can described using multiple metrics and thresholds.

To address these challenges, few heuristic-based approaches are proposed to generate design defects detection rules from defect examples [163, 164]. However, such studies require a high number of interface design defect examples (data) to provide efficient detection rules solutions. In fact, design defects are rarely documented by developers which explains the need for an approach that is able to generate artificial defects examples in order to improve the efficiency of detection rules. In addition, it is challenging to ensure the diversity of the examples to cover most of the possible bad-practices. In addition, these heuristic-based studies are still also limited to the use of structural metrics and did not consider the impact of antipatterns on the performance of the services. In this work, we start from the hypothesis that the generation of efficient web service defect detection rules heavily depends on the coverage and the diversity of the used defect examples. In fact, both mechanisms for the generation of detection rules and the generation of defect examples are dependent. Thus, the intuition behind this work is to generate examples of defects that cannot be detected by some possible detection solutions and then adapting these rules-based solutions to be able to detect the generated defect examples. These two steps are repeated until reaching a termination criterion (e.g. number of iterations). To this end, we propose, for the first time, to consider the web services defects detection problem as a bi-level one [165]. Bi-Level Optimization Problems (BLOPs) are a class of challenging optimization problems, which contain two levels of optimization tasks. The optimal solutions to the lower level problem become possible feasible candidates to the upper level problem. In addition, we assume that an effective web service antipatterns detection process should be based on a combination of *dynamic Quality of Services (QoSs) attributes* and the *structural information* of web service (static interface/code metrics). Several of Web services antipatterns can negatively impact the QoS such as availability and response time. For instance, a GOWS antipattern typically can include a large number of operations which can reduce the response time dramatically. A GOWS web service suffers, in general, from a low cohesion which may lead to a high response time and a low availability due to the large number of calls between operations at multiple web services.The use of response time quality attribute may help to find the right threshold in terms of number of operations and cohesion level that can truly impact the web service performance. Thus, the generated detection rules can be more accurate.

In our approach, the upper level generates a set of detection rules, a combination of static and dynamic metrics and QoS attributes, which maximizes the coverage of the base of defect examples and the coverage of artificial defects which are generated by the lower level and minimizes the size of a generated rule. The lower level maximizes the number of generated artificial defects that cannot be detected by the rules produced by the upper level and minimizes the distance between the artificial defects and the base of bad-designed web services examples. The advantage of our bi-level approach is that the generation of detection rules is not limited to some defect examples that are hard to collect. However, this approach allows the prediction of new defects that are different from those in the base of examples. Furthermore, our problem requires a search in a large space for a solution which balances different conflicting objectives to generate rules suitable for different scenarios. Therefore, it would be appropriate to consider a multi-objective search-based approach that finds a trade-off between conflicting objectives in each level.

We applied and validated these rules on a benchmark of 662 real-world web services from different application domains and five common web service antipatterns. However, our proposed approach can be used in a generic way for any other type of defect as long as a number of examples are available. Statistical analysis of our experiments shows the efficiency of our bi-level multi-objective approach in detecting web service antipatterns, with a precision of 84% and recall of 91%. The results confirm the outperformance of our bi-level approach compared to state-of-art web service design defects detection techniques [163, 164, 166] and our previous work limited to mono-objective bi-level approach using only structural metrics [157]. Thus, the validation confirmed our hypothesis that the detection of antipatterns require a combination of structural and performance metrics.

3.2 Motivating Example and Challenges

Web Services Design Defects. Web service interface defects are defined as bad design choices that can have a negative impact on the interface quality such as maintainability, changeability, comprehensibility and discoverability [167] which may impact the usability and popularity of services [168]. They can be also considered as structural characteristics of the interface that may indicate a design problem that makes the service hard to evolve and maintain, and trigger refactoring [169]. In fact, most of these defects can emerge during the evolution of a service and represent patterns or aspects of interface design that may cause problems in the further development of the service. In general, they make a service difficult to change, which may in turn introduce bugs. It is easier to interpret and evaluate the quality of the interface design by identifying different defects definition than the use of traditional quality metrics. To this end, some studies defined different types of web services design defects [170, 169]. In our experiments, we focus on the eight following web service defect types since they are the most frequent and severest ones [171], and also to be able to compare our detection approach to the state of the art:

- *God object Web service (GOWS):* implements a high number of operations related to different business and technical abstractions in a single service.
- *Fine-grained Web service (FGWS):* is a too fine-grained service whose overhead (communications, maintenance, and so on) outweighs its utility.
- *Chatty Web service (CWS):* represents an antipattern where a high number of operations are required to complete one abstraction.
- *Data Web service (DWS):* contains typically accessor operations, i.e., getters and setters. In a distributed environment, some web services may only perform some simple information retrieval or data access operations.
- *Redundant PortTypes (RPT):* is an antipattern where multiple portTypes are duplicated with the similar set of operations. In fact, one of the potential sources of RPT defects is the use of defective WSDL generation tools as pointed out in [167]. Another source of RPT is when developers are adding new features in a rush without considering the reusability of their implementation and architecture design (similar to code clones).

The web service antipatterns detection mechanism involves finding the fragments of the design which violate some quality indicators. Table 3.1 describes all the metrics that are used in this thesis (contribution III) to cover bad quality symptoms. These metrics are a combination of static, dynamic and performance metrics related to the following abstraction levels of web services applications :

- **Web service interface-level (WSDL) metrics:** are mainly related to the interface, message, operation and Port type. The list of WSDL metrics are described in Table3.1 from ALPS until RPT. In our approach, we are considering the two WSDL versions 1.0 and 2.0 since they are both supported by our parser in extracting the metrics.

- **Web service code-level metrics:** are the static information that we can extract from the services code skeletons. The most widely-used code-level metrics are those defined by Chidamber and Kemerer [172] as described in Table 3.1 (from Ca until CC). For all code-level metrics, we calculate the average value for all the classes that implement the specific web service. For instance , the depth of inheritance (DIT) represents the depth of inheritance of a class and it is defined as the depth of the class in the inheritance tree and the depth of a node of a tree refers to the length of the maximal path from the node to the root of the tree. Thus, we parsed the code to extract the calls by static analysis and also used relevant keywords such as "extends" to confirm the nature of these calls. The QoS metrics are more related to the execution of web services to calculate response time, availability, etc.

- **Quality of Service (QoS) metrics:** we selected 9 popular metrics (From Response until Documentation in Table 3.1), namely response, availability, throughput, successability, reliability, and latency are dynamic metrics which measure the web service overall performance. Documentation and compliance are static metrics to measure the usability of the web service interface. In our work, We extracted all these metrics from the QWS dataset [173].

We selected these defect types in our experiments because they are the most frequent, the hardest to detect [174, 163], and cover different maintainability factors. We have also several examples of these defects and we are able to compare the performance of our detection technique to existing studies [157, 163, 166]. However, our proposed approach (contribution III) is generic and can be extended to any type of defect.

Challenges: In the following, we introduce some issues and challenges related to the detection of the web service defects. Overall, there is no general consensus on how to decide if a particular design violates a quality heuristic. In fact, there is a difference between detecting symptoms and asserting that the detected situation is an actual design defect. Another issue is related to the definition of thresholds when dealing with quantitative information.

Table 3.1: List of Web services metrics used

Category	Metric Name	Definitions
QoS Metrics	Response	Time to send a request and receive a response (ms)
	Availability	Number of successful invocation/total invocation (%)
	Throughput	Number of invocations for a given time (invokes/sec)
	Successability	Number of response/number of request messages (%)
	Reliability	Ratio of number of error messages to total messages (%)
	Compliance	The extent to which a WSDL follows specification (%)
	Best practices	The extent to which a service follows WS-I BAsic (%)
	Latency	Time taken for the server to process a given request (ms)
	Documentation	Measure of documentation (i.e. description tags) in WSDL
Interface Metrics	ALPS	Average length of port types signature
	COH	Cohesion
	COUP	Coupling
	NAOD	Number of accessor operations declared
	NCO	Number of CRUD operations
	NOD	Number of operations declared
	NOPT	Average number of operations in port types
	NPT	Number of port types
	RAOD	Ratio of accessor operations declared
	ALOS	Average length of operations signature
	AMTO	Average number of meaningful terms in operations names
	ANIPO	Average number of input parameters in operations
	ANOPO	Average number of output parameters in operations
	NPO	Average number of parameters in operations
	ALMS	Average number of message signature
	AMTM	Average number of meaningful terms in message names
	NOM	Number of messages
	NPM	Average number of parts per message
	AMTP	Average number of meaningful terms in port type names
	NCT	Number of complex types
	NCTP	Number of complex types parameters
	NST	Number of primitive types
	RPT	Ratio of primitive types over all defined types
Code Metrics	Ca	Afferent couplings
	CAM	Cohesion Among Methods of Class
	CBO	Coupling Between Object Classes
	Ce	Efferent couplings
	DAM	Data Access Metric
	DIT	Depth of Inheritance Tree
	LCOM	Lack of cohesion in methods
	LCOM3	Lack of cohesion in methods
	LOC	Lines of Code
	MFA	Measure of Functional Abstraction
	MOA	Measure of Aggregation
	NOC	Number of Children
	NPM	Number of Public Methods
	RFC	Response for a Class
	WMC	Weighted methods per class
	AMC	Average Method Complexity
	CC	The McCabe's cyclomatic complexity

For example, the GOWS defect detection involves information such as the interface size as illustrated in Figure 3.1. Although we can measure the size of an interface, an appropriate threshold value is not trivial to define. An interface considered large in a given service/-community of users could be considered average in another. Thus, it may not be accurate to identify a GOWS defect based on structural information such as the number of operations. Both structural and non-structural (QoS attributes) factors are complementary when detecting a GOWS antipattern. The impact of the appearance of GOWS can be seen on the performance of services such as response time and availability. Thus, these attributes can confirm a GOWS antipattern rather than just relying on number of operations. In fact, it is always challenging to define a threshold for the number of operations but a combination of both low QoS attributes and high number of operations will definitely improve the accuracy of the GOWS detection rules. Programmers are mainly interested to fix design defects impacting the quality of services and not those who just violate some metrics such as coupling, cohesion and number of operations. However, existing studies are limited to the use of structural information when detecting design defects.

Our GOWS motivating example was not only related to the size of the interface but also other metrics such as low cohesion. The used Amazon service's interface suffers from low-cohesion and it is already classified in our dataset as a GOWS antipattern, its high response time and low availability can be explained by the low cohesion of operations and not only the size of the interface. If the number of operations becomes high (like in most GOWS antipatterns) then the response time and availability will be dramatically impacted. In practice, one of the main reasons of services low availability is the high number of calls that make some servers inaccessible/down.

The generation of detection rules requires a large defect example set to cover most of the possible bad-practice behaviors. Defects are not usually documented by developers which results in lack of defects examples. Thus, it is time-consuming and difficult to collect defects and inspect manually large web services. In fact, unlike the bugs localization problem where

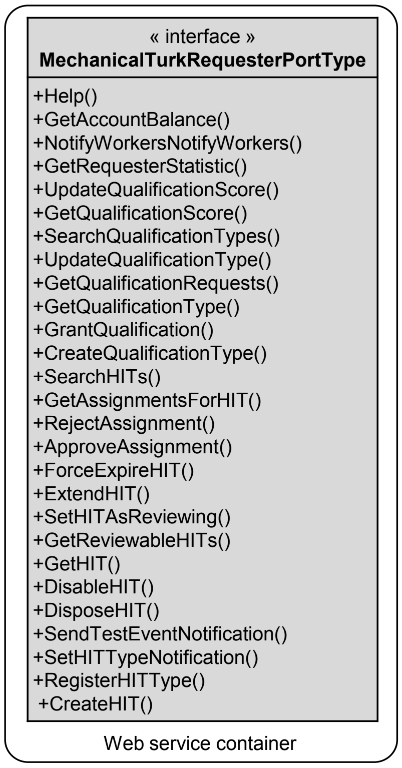


Figure 3.1: God object Web service (GOWS) example.

bug reports data are available to train the model, detecting web services antipatterns suffers from the lack of documented defect examples which affects the efficiency of the generated detection rules. In addition, it is challenging to ensure the diversity of the defect examples to cover most of the possible bad-practices then using these examples to generate good quality of detection rules.

To address the above-mentioned challenges, we propose to consider the web service defects detection problem as a bi-level multi-objective optimization problem.

3.3 Approach:Bi-level Multi-objective Optimization Technique

In this study, we considered the web services defect detection problem as a bi-level multi-objective optimization problem where the optimal solution of the lower level problem determines the feasible space of the upper level optimization problem [165, 175]. In our adaptation, the upper level problem is the generation of detection rules and the lower level problem is the generation of design defects that may not be detected using the rules of the upper level solutions.

We start by describing the basic concepts of bi-level optimization, then we introduce the multi-objective optimization technique.

3.3.1 Bi-level Optimization

Most studied real-world and academic optimization problems involve a single level of optimization. However, in practice, several problems are naturally described in two levels. Bi-level optimization problem (BLOP) also called two-level optimization, is a specific type of optimization where one problem is nested within another [165, 175]. In such problems, we find a nested optimization problem within the constraints of the outer optimization one. The outer optimization task is usually referred as the upper level problem or the leader problem. The nested inner optimization task is referred as the lower level problem or the follower problem, thereby referring the bi-level problem as a leader-follower problem. The follower problem appears as a constraint to the upper level, such that only an optimal solution to the follower optimization problem is a possible feasible candidate to the leader one as described in Figure 3.3.

The problem contains two types of variables: (1) the upper-level variables x_u and (2) the lower-level variables x_l. Formally, BLOP is defined as follows:

Definition1. For the upper-level objective function F: $\mathbb{R}^n \times \mathbb{R}^n \rightarrow \mathbb{R}$ and lower-level objective function $f : \mathbb{R}^n \times \mathbb{R}^n \rightarrow \mathbb{R}$, the bi-level problem is given by:

$$\begin{aligned} \min_{x_u \in X_U, x_l \in X_L} \quad & F(x_u, x_l) \\ \text{subject to} \quad & x_l \in argmin\{f(x_u, x_l), g_j(x_u, x_l) \leq 0, j = 1, ..., J\} \\ & G_k(x_u, x_l) \leq 0, k = 1, ..., K \end{aligned}$$

where $G_k : X_U \times X_L \rightarrow \mathbb{R}$ and $g_j : X_U \times X_L \rightarrow \mathbb{R}$ denote respectively the upper and the lower level constraints. J is the population size at the upper level, K is the population size at the lower level and n is the number of fitness functions,

The study involved multiple objectives at the upper lever, and multiple objectives at the lower level. Thus, the next section presents the Multi-objective optimization technique.

Existing methods to solve BLOPs could be classified into two main families: (1) classical methods and (2) evolutionary methods. The first family includes extreme point-based approaches [176], penalty function methods [177] and trust region methods [178]. The main shortcoming of these methods is that they heavily depend on the mathematical characteristics of the BLOP at hand. The second family includes meta-heuristic algorithms that are mainly Evolutionary Algorithms (EAs). Recently, several EAs have demonstrated their effectiveness in tackling such type of problems thanks to their insensibility to the mathematical features of the problem in addition to their ability to tackle large-size problem instances by delivering satisfactory solutions in a reasonable time [179, 180, 181].

In our adaptation, each level is formulated as a multi-objective problem. The next subsection will give details about multi-objective optimization.

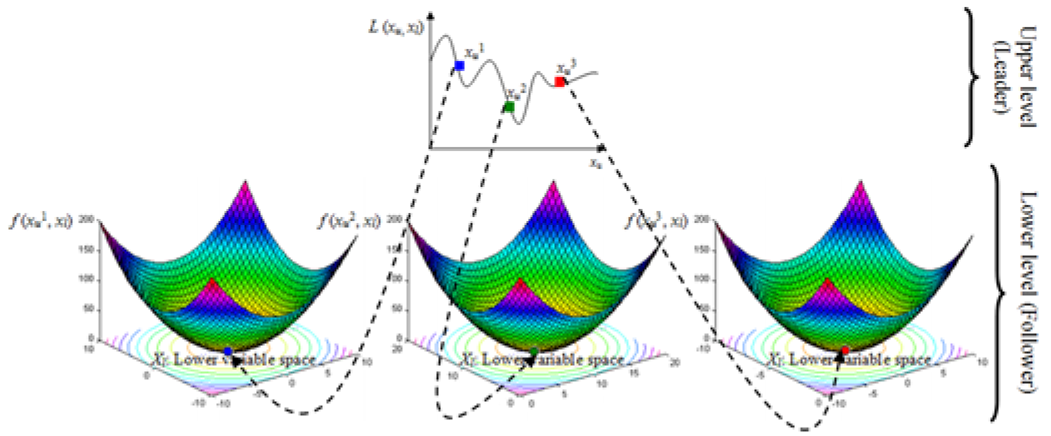


Figure 3.2: The upper and lower levels of the Bi-Level process

3.3.2 Multi-Objective Optimization

Multi-Objective search considers more than one objective function to be optimized simultaneously. Objective functions are used to evaluate the generated solutions. It is hard to find an optimal solution that solves such problem because the objectives to be optimized are conflicting. For this reason, a multi-objective search-based algorithm could be suitable to solve this problem because it finds a set of alternative solutions, rather than a single solution as result. One of the widely used multi-objective search techniques is Non-dominated Sorting Genetic Algorithms (NSGA-IIs) [182] that has shown good performance in solving several software engineering problems [183].

A high-level view of NSGA-II is depicted in Algorithm 1. The algorithm starts by randomly creating an initial population P_0 of individuals encoded using a specific representation (line 1). Then, a child population Q_0 is generated from the population of parents P_0 (line 2) using genetic operators (crossover and mutation). Both populations are merged into an initial population R_0 of size N (line 5). *Fast-non-dominated-sort* [182] is the technique used by NSGA-II to classify individual solutions into different dominance levels (line 6). Indeed, the concept of non-dominance consists of comparing each solution x with every other solution in the population until it is dominated (or not) by one of them. According to Pareto optimality: "A solution x_1 is said to dominate another solution x_2, if x_1 is no worse than x_2 in all objectives and x_1 is strictly better than x_2 in at least one objective". Formally, if we consider a set of objectives f_i , $i \in 1..n$, to maximize, a solution x_1 dominates x_2 :

$$\text{iff } \forall i, f_i(x_2) \leqslant f_i(x_1) \text{ and } \exists j \mid f_j(x_2) < f_j(x_1) \tag{3.1}$$

The whole population that contains N individuals (solutions) is sorted using the dominance principle into several fronts (line 6). Solutions on the first Pareto-front PF_0 get assigned dominance level of 0 Then, after taking these solutions out, *fast-non-dominated-sort* calculates the Pareto-front PF_1 of the remaining population; solutions on this second

Algorithm 1 High level pseudo code for NSGA-II

```
1: Create an initial population $P_0$
2: Create an offspring population $Q_0$
3: $t = 0$
4: while stopping criteria not reached do
5:     $R_t = P_t \cup Q_t$
6:     F = fast-non-dominated-sort($R_t$)
7:     $P_{t+1} = \emptyset \; and \; i = 1$
8:     while $\mid P_{t+1} \mid + \mid F_i \mid \leqslant N$ do
9:         Apply crowding-distance-assignment($F_i$)
10:        $P_{t+1} = P_{t+1} \cup F_i$
11:        $i = i + 1$
12:    end while
13:    $Sort(F_i, \prec n)$
14:    $P_{t+1} = P_{t+1} \cup F_i[N- \mid P_{t+1} \mid]$
15:    $Q_{t+1}$ = create-new-pop($P_{t+1}$)
16:    t = t+1
17: end while
```

front get assigned dominance level of 1, and so on. The dominance level becomes the basis of selection of individual solutions for the next generation. Fronts are added successively until the parent population P_{t+1} is filled with N solutions (line 8). When NSGA-II has to cut off a front PF_i and select a subset of individual solutions with the same dominance level, it relies on the crowding distance [182] to make the selection (line 9). This parameter is used to promote diversity within the population. This front PF_i to be split, is sorted in descending order (line 13), and the first (N- $|P_{t+1}|$) elements of PF_i are chosen (line 14). Then a new population Q_{t+1} is created using selection, crossover and mutation (line 15). This process will be repeated until reaching the last iteration according to stop criteria (line 4).

Therefore, a bi-level multi-objective optimization problem involves two levels of multi-objective optimization problems, each one implements an NSGA-II algorithm with different set of objectives as described in the next subsection.

3.3.3 Approach Overview

As Figure 3.3 shows, our Bi-Level Multi-Objective (BLMO) approach includes two levels where both the leader and the follower have two objectives.

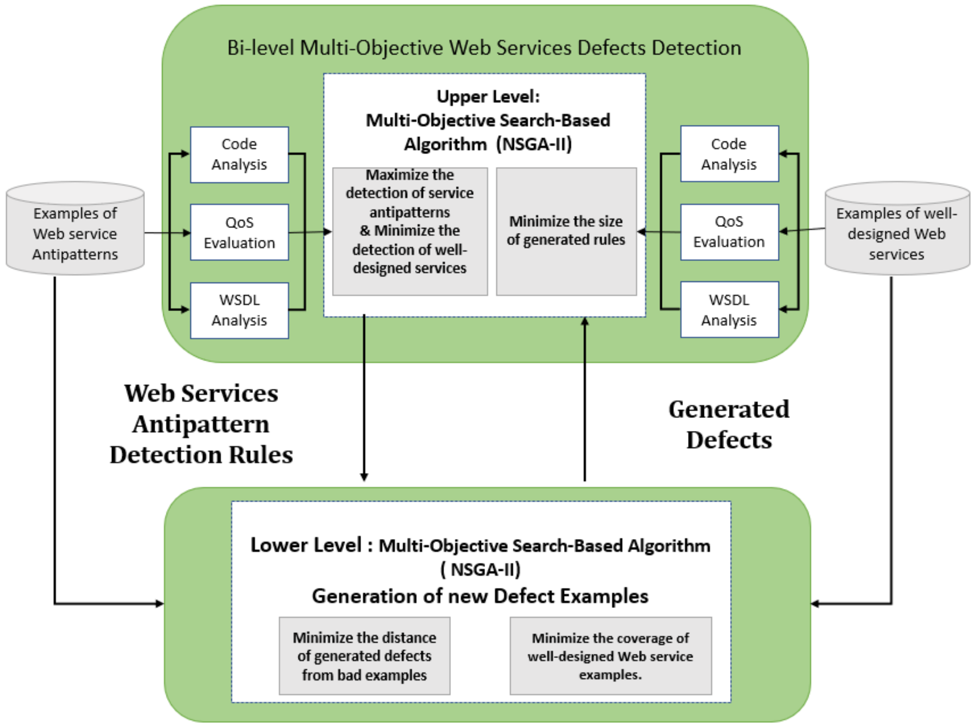

Bi-level Multi-Objective Web Services Defects Detection
Upper Level: Multi-Objective Search-Based Algorithm (NSGA-II)
Maximize the detection of service antipatterns & Minimize the detection of well-designed services
Minimize the size of generated rules
Code Analysis
QoS Evaluation
WSDL Analysis
Code Analysis
QoS Evaluation
WSDL Analysis
Examples of Web service Antipatterns
Examples of well-designed Web services
Web Services Antipattern Detection Rules
Generated Defects
Lower Level : Multi-Objective Search-Based Algorithm (NSGA-II)
Generation of new Defect Examples
Minimize the distance of generated defects from bad examples
Minimize the coverage of well-designed Web service examples.

Figure 3.3: Bi-level Multi-Objective Web service defects detection overview

As described in Figure 3.3, the proposed approach takes as inputs two sets of web service examples: (1) one set contains service antipattern examples and (2) another has well-designed service examples. It extracts the metrics, previously described, of each web service in the sets. Then, the upper level generates a set of detection rules per solution. The detection rule generation process selects randomly, from the list of possible metrics, a combination of quality metrics and their threshold values to detect a specific antipattern type. Therefore, the optimal solution is a set of detection rules that best detect the antipatterns of the base of examples and those generated by the lower level while minimizing the number of generated rules.

The follower (lower level) uses well-designed web service examples to generate “artificial” design defects based on the notion of deviation from a reference (well-designed) set of web services. The generation process of web services defect examples is performed using a multi-objective heuristic search that maximizes on one hand, the distance between generated web service defect examples and reference examples and, on the other hand, maximizes the number of generated examples that are not detected by the leader (detection rules).

In our bi-level multi-objective approach, the two levels are dependent and therefore there is no parallelism. The upper level is executed for a number of iterations then the lower level for another number of iterations. After that, the best solution found in the lower level will be used by the upper level to evaluate the detection rules, and then this process is repeated several times until reaching a termination criterion such as the number of iterations. For each level, we selected the ideal point from the Pareto front of solutions which corresponds to the closest solution to the best possible values of the fitness functions.

Algorithm 2 Upper level algorithm

```
1: **Inputs:** Quality of web service metrics M, web services defect examples base B, Well-designed
   web services base D, Number of best upper solutions that are considered for lower level optimiza-
   tion nbs, Upper population size N1, Lower population size $N_2$, Upper number of generations
   G1, Lower number of generations $G_2$
2: **Output:** Best detection rule BDR
3: **Begin**
4: $P_0 \leftarrow$ Initialization($N$,$M$)
5: **for each** $DB_e$ in $P_0$ **do**
6:     $BCS_0 \leftarrow$ NSGA-IIWSDefectsGeneration($DR_0$,$D$,$N_2$,$G_2$);
7:     $BR_0 \leftarrow$ Evaluations($DR_0$,$B$,$BCS_0$);
8: **end for**
9: $t \leftarrow 1$
10: **while** t<G1 **do**
11:     $Q_t \leftarrow$ Variation($P_{t-1}$)
12:     **for each** $DR_t$ in $Q_t$ **do**
13:         $DR_t$ = UpperEvaluations($DR_t$,$B$);
14:     **end for**
15:     **for each of the best** nbs **rules** $DR_t$ in $Q_t$ **do**
16:         $BCS_t \leftarrow$ NSGA-IIWSDefectsGeneration($DR_t$,$D$,$N_2$,$G_2$);
17:         $DR_t \leftarrow$ EvaluationsUpdate($DR_t$,$BCS_t$);
18:     **end for**
19:     $U_t \leftarrow P_t \cup Q_t$;
20:     $P_{t+1} \leftarrow$ EnvironmentalSelection($N_1$,$U_t$);
21:     $t \leftarrow$t+1;
22: **end while**
23: BDR $\leftarrow$ IdealPointSelection($P_t$);
24: **END**
```

Algorithm 3 Lower level algorithm: NSGA-IIWSDefectsGeneration

```
1: Inputs: Upper level detection rule UDR, Well-desiged web service examples base D, Population
   size N, number of generations G
2: Output: Best artificial web service defects BCS
3: Begin
4: $P_0 \leftarrow$Initialization($N$,$D$);
5: $P_0 \leftarrow$Evaluation($P_0$,D,UDR);
6: $t \leftarrow 1$;
7: while t<G do
8:     $Q_t \leftarrow$ Variation($P_{t-1}$)
9:     $Q_t$ = Evaluation($Q_t$,$D$,$UDR$);
10:    $U_t \leftarrow P_t \cup Q_t$;
11:    $P_{t+1} \leftarrow$ EnvironmentalSelection($N$,$U_t$);
12:    $t \leftarrow$t+1;
13: end while
14: BCS$\leftarrow$IdealPointSelection($P_t$);
15: END
```

3.3.4 Problem Formulation

Solution Representation. Each candidate solution in the upper level is a sequence of detection rules where each rule is represented by a binary tree such that:

- The Root and each internal node represent a logic operator either “AND” or “OR” to connect other nodes.
- Each leaf node represents a quality metric and its corresponding threshold.

For example, the following rule of Fig. 3.4 states that a web service s satisfying the following combination of metrics is considered as a GOWS defect:

As described in Figure 3.5, the generated structure of defects, in the lower level, is represented as a vector where each element is a (metric, threshold) element that characterizes the generated artificial web service defect.

Fitness Functions. At the upper level, we aim to optimize two fitness functions. The first one is formulated to maximize the coverage of web services defect examples (input) and the coverage of the generated artificial web service defects by the lower level. The second fitness function is formulated to minimize the size of the generated rule. Thus, the fitness

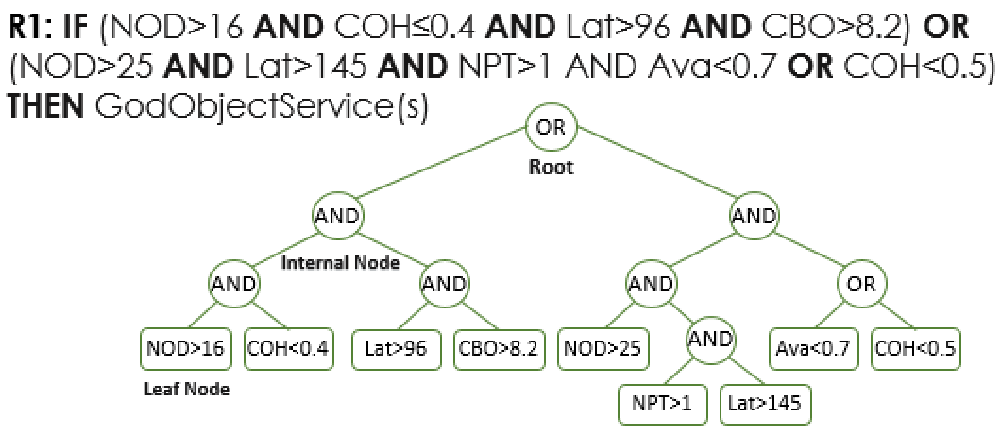
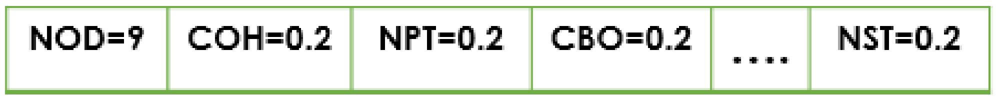


Figure 3.4: Solution Representation at the Upper Level.



NOD=9	COH=0.2	NPT=0.2	CBO=0.2	....	NST=0.2

Figure 3.5: Solution Representation at the Lower Level.

functions at the upper level are defined as follows:

$$\begin{aligned} &Maximize f_{upper,1} = \frac{Precision\ (SR, WSDE{+}AWSD) +\ Recall(SR, WSDE{+}AWSD)}{2} \\ &Minimize f_{upper,2} = size(DetectionRules) \end{aligned} \tag{3.2}$$

where $WSDE$ is the abbreviation for Web Services (WSs) Defect Examples, $AWSD$ is the abbreviation for Artificial WS Defects and SR is the set of generated detection rules (solution).

At the lower level, for each solution of the upper level, an NSGA-II is executed to generate the best set of artificial defects that cannot be detected by the detection rules of the upper level. Two objective functions are formulated at the lower level to maximize the number of un-detected artificial defects that are generated and minimize the distance with web services antipatterns. More formally, the two objectives are expressed as follows:

$$
\begin{aligned}
&Minimize f_{lower,1} = \sum_{i=1}^{M} (ArtificialDefects(i) - Average(RAE(i))) \\
&Minimize f_{lower,2} = countdefects(DR, AD)
\end{aligned}
\tag{3.3}
$$

where RAE is the abbreviation for References Antipatterns Examples, DR is the detection rules defined at the upper level, AD is the generated artificial defects and M is the number of metrics used to compare between artificial defects and the poor Web services examples. The first fitness function calculates the distance between the artificial defects and the ones in our base of examples to make sure that they are different.Thus, M is not restricted based on the type of antipatterns because we do not want our artifical examples to be restricted to limited behavior of antipatterns.

In our proposed approach, we are not generating detection rules only based on the QoS properties but we are including code-level metrics and interface metrics as well. Our benchmark/dataset contains web services along with their code-level , interface and QoS metrics and anti-patterns. The training data/examples guided the bi-level algorithm, via the fitness functions. to identify/generate the patterns and relationships between the different metrics and the anti pattern type. Since the fitness functions are mainly based on coverage criteria thus we can confirm that the best detection rules can be generalized on a large number of web services.

Change Operators. For the upper level, the mutation operator can be applied to a leaf node (metric), or to an internal node (logical operator) in our tree representation. It starts by randomly selecting a node in the tree. Then, if the selected node is a leaf (metric), it is replaced by another metric or another threshold value. Each metric has a maximum and minimum values which represent the range from where the change operator selects a threshold value. However, if it is an internal node (AND-OR), it is replaced by a new function. For the lower-level, the mutation operator consists of randomly changing a metric in one of the vector dimension. Regarding the crossover at the upper level, two parents are selected, and a sub-tree is picked on each one. Then, the crossover operator swaps the nodes and their relative sub-trees from one parent to the other. The crossover operator can be applied to only parents having the anti-patterns type to detect. Each child thus combines information from both parents.

The crossover operator allows creating two offspring Child1 and Child2 from the two selected parents Parent1 and Parent2, where the first x elements of Parent1 become the first x elements of Child2. Similarly, the first x elements of p2 become the first x elements of Child1.

3.4 Evaluation

In order to evaluate our approach for detecting antipatterns using the proposed bi-level multi-objective (Bi-Level Multi-Objective Problemss (BLMPOs)) approach, we conducted a set of experiments based on an existing benchmark[173]. Each experiment is repeated 30 times, and the obtained results are subsequently statistically analyzed with the aim to compare our multi-objective bi-level approach with a variety of existing web service antipatterns detection approaches. In this section, we first present our research questions and then describe and discuss the obtained results.

3.4.1 Research Questions

We defined the following research questions for our empirical study:

- **RQ1.** To what extent does the proposed approach detect various types of web service antipatterns based on a combination of structural and dynamic (QoS) metrics? It is important to quantitatively assess the completeness and correctness of our BLMO detection approach based on QoS and multi-objective search.

- **RQ2.** How does BLOP perform compared to existing web service antipatterns detection algorithms not using QoS metrics and multi-objective search? This research question is helpful to evaluate the benefits of the use of a multi-objective algorithm at both levels since we will compare our approach to our previous work based on a bi-level mono-objective algorithm (Bi-Level Optimization Problems (BLOPs))[157]. Furthermore, we compared our approach with another mono-level search based approach [163] and an existing deterministic approach, SODA-W [166] which is not based on heuristic search. SODA-W is based on manually defined rules(including threshold values) to detect web service antipatterns. Both approaches are limited to the use of structural metrics thus they are useful to evaluate the benefits of considering dynamic quality of services attributes.

- **RQ3.** To what extent the detection of Web service antipatterns based on a combination of QoS and structural metrics can be useful and relevant for practitioners? We collected the opinions of developers about our tool and their perception of the importance of several of detected web service antipattern types.

3.4.2 Experimental Settings

To evaluate the performance of the proposed approach, we used existing benchmarcks of referencen number [173] to build our final dataset which consists of 662 good and bad web services desing. These web services (1) have different sizes, (2) originate from various application categories such as financial, science, travel, weather, etc, (3) have available source code, and (4) belonging to different development teams. These web services are retrieved from the QWS dataset then filtered to eliminate the ones which are not running anymore to be used by subscribers. We extract their interface file and code skeleton. Then, we manually inspected and validated the antipatterns of these services.

We considered the different antipattern types described in Section 2. Table 3.2 shows the distribution of these antipatterns in the 662 web services. We used a 10-fold cross validation procedure. In fact, the 10-fold cross validation means that we split our data into 10 training data sets and 1 evaluation data set. For each fold, one category of services (evaluation data) is evaluated using the remaining nine categories (training data) as training examples.

Then, we repeated the process ten times. We use the two measures of precision and recall to evaluate the accuracy of our approach and to compare it with existing techniques. Precision denotes the ratio of true antipatterns detected to the total number of detected antipatterns, while recall indicates the ratio of true antipatterns detected to the total number of existing antipatterns.

Table 3.2: Web services used in the empirical study.

Antipatterns types	**# services**	**Distribution**
GOWS	237	36%
FGWS	179	27%
CWS	39	5%
DWS	119	18%
RPT	113	17%

Table 3.3: Antipattern occurrences within the 662 Web Services.

Category	**# services**	**# antipatterns**	**average NOD**	**average NOM**	**average NCT**
Financial	121	52	31.73	57.31	22.14
Science	58	19	12.49	17.14	98.72
Search	49	21	9.66	18.94	28.43
Shipping	72	17	17.28	27.76	23.42
Travel	81	22	21.07	33.13	131.12
Weather	73	18	11.63	17.16	8.24
Media	33	19	11.8	16.4	32.29
Education	52	15	12.73	16.23	32.46
Messaging	63	20	9.18	13.36	18.25
Location	83	22	6.89	29.73	11.15
All	662	139	14.18	27.3	48.6

To answer RQ1, we use both recall and precision to evaluate the efficiency of our approach in identifying antipatterns. We investigated the web service defect types that were detected to find out whether there is a bias towards the detection of specific web service defect types.

To answer RQ2, we evaluated on the effectiveness of BLMO compared to existing approaches using the precision (PR) and recall (RC) measures. All three approaches are tested on the same benchmark described in Table 3.2 to ensure a fair comparison.The distribution of antipatterns type means the number of services containing at least one instance of that type of antipattern divided by the total number of analyzed services. We have also evaluated the execution time (T) required by the different approaches.

To answer RQ3, we conducted a post-study survey with developers to understand what types of web services antipatterns are important for them in practice and how useful our detection tool. To this end, we asked 48 software developers, including 29 professional developers working on the development of services-based application and 19 graduate students from the University of Michigan. The experience of these subjects on web development and web services ranged from 2 to 16 years. All the graduate students have an industrial experience of at least 2 years with large-scale systems especially in automotive industry.

An often-omitted aspect in metaheuristic search is the tuning of algorithm parameters. In fact, parameter setting influences significantly the performance of a search algorithm on a particular problem. The stopping criterion was set to 100,000 fitness evaluations for all search algorithms in order to ensure fairness of comparison. We used a high number of evaluations as a stopping criterion since our bi-level approach requires involves two levels of optimization. Each algorithm was executed 30 times with each configuration and then comparison between the configurations was performed based on precision and recall using the Wilcoxon test. The Wilcoxon signed-rank test is a non-parametric statistical hypothesis test used to compare two related samples, matched samples, or repeated measurements on a single sample to assess whether their population mean ranks differ (i.e. it is a paired difference test). In our case, it was used due to the randomness of the meta-heuristic algorithms and to ensure that their out-performance is not random but consistent on 30 independent runs. Additionally, the other parameters value were fixed by trial and error and are as follows: (1) crossover probability = 0.4; mutation probability = 0.7 where the probability of gene modification is 0.1. For our bi-level approach, both lower-level and upper-level are run each with a population of 20 individuals and 30 generations. It should be noted that the lower-level routine is not called for all upper-level population members. To control, the high computational cost of our bi-level approach, only ns% of the best upper-level population members are allowed to call the lower-level optimization algorithm. Based on a parametric study, the value of 5% for ns is found to be adequate empirically in our experiments. For our experiment, we generated up to 100 artificial web service antipatterns from deviation with the best of examples.

Since metaheuristic algorithms are stochastic optimizers, they can provide different results for the same problem instance from one run to another. For this reason, our experimental study is performed based on 30 independent simulation runs for each problem instance, and the obtained results are statistically analyzed by using the Wilcoxon rank sum test [184] with a 95% confidence level ($\alpha = 5\%$). The Wilcoxon signed-rank test is a non-parametric statistical hypothesis test used when comparing two related samples to verify whether their population mean-ranks differ or not. The latter verifies the null hypothesis H0 that the obtained results of two algorithms are samples from continuous distributions with equal medians, as against the alternative that they are not, H1. In this way, we could decide whether the outperformance of BLMO over one of each of the other detection algorithms (or the opposite) is statistically significant or just a random result.

3.4.3 Results and Discussions

3.4.3.1 Results for RQ1

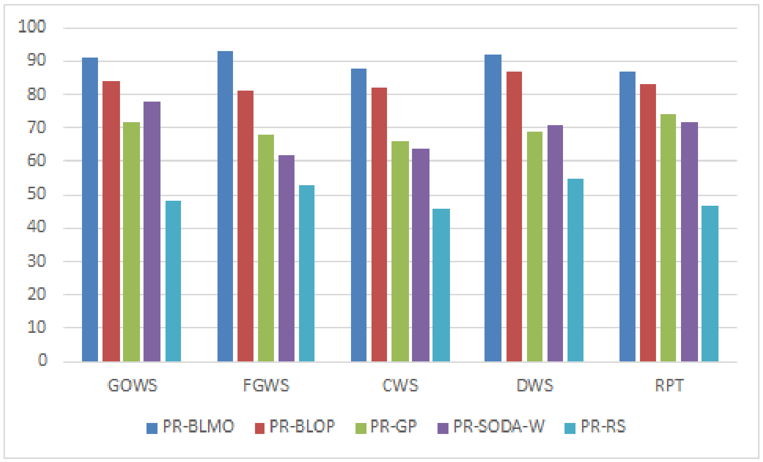


Figure 3.6: Median precision on 30 runs for the 10-folds of the 662 web services using the different detection techniques with a 95% confidence level.

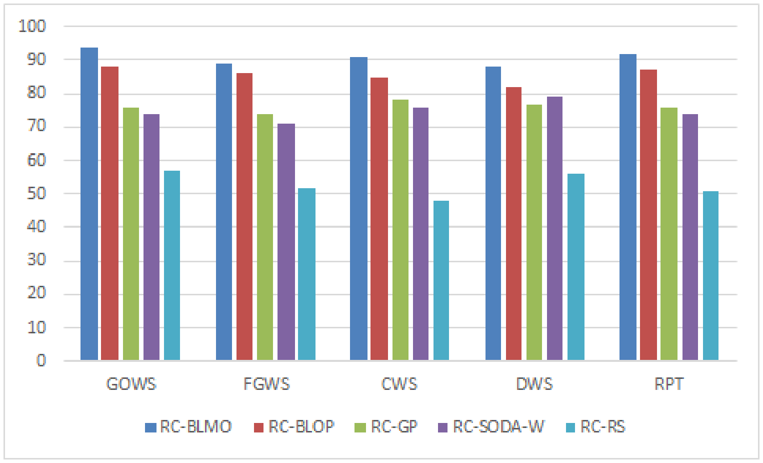


Figure 3.7: Median recall on 30 runs for the 10-folds of the 662 web services using the different detection techniques with a 95% confidence level.

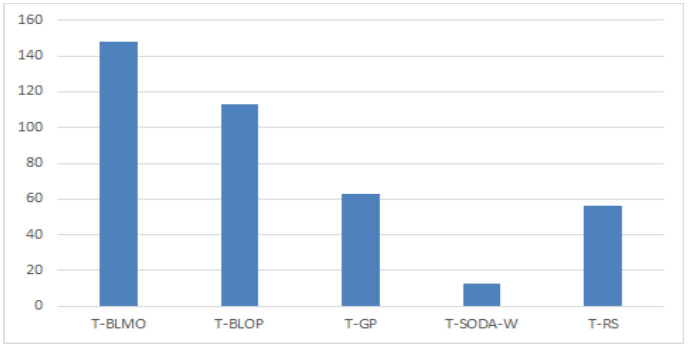


Figure 3.8: Median execution time on 30 runs for the 10-folds of the 662 web services using the different detection techniques.

The results for the first research question RQ1 are presented in Figures 3.6 and 3.7. The obtained results show that our BLMO approach is able to detect most of the expected antipatterns in the different web services with a median precision higher than 90%. Thus, our technique does not have a bias towards the detection of specific web service antipattern types. As described Figures 3.6 and 3.7, we had an almost equal accuracy distribution of each Wev service antipattern types. Having a relatively good distribution of antipattern is useful for developers to make the right decisions about the quality of services. Overall, all the five web service antipatterns types are detected with good precision and recall scores in the different systems (an average of 91%). This ability to identify different types of antipatterns underlines a key strength to our approach. Most other existing tools and techniques rely heavily on the notion of size and static information to detect antipatterns. This is reasonable considering that some antipatterns like the GOWS are associated with the notion of size (number of operations). For web service antipatterns like RPT, however, the notion of size is less important, and this makes this type of anomaly hard to detect using structural information. This also confirms that the use of the dynamic quality of service attributes helped to achieve good results in detecting antipatterns.

The highest precision value for *GOWS* (93%) can be explained by the fact that these web service antipatterns are the easiest to detect due to their structure. For the web services antipattern type *DWS*, the precision is the lowest one (88%), but is still an acceptable score. These antipatterns are likely to be difficult to detect using metrics alone and may require interactions with the user. Sometimes developers have a reason why a Web service is too small such as they wanted to make sure that specific operations are loosely coupled to other services for security reasons. Thus, it is difficult to consider the context of specific requirements in static and dynamic rules. Similar observations are valid for the recall. The obtained results indicate that our approach is able to achieve an average recall of 89%. Thus, the quality of the detection rules are good for almost all the web service defect types considered in our experiments. Thus, we can conclude that our BLMO multi-objective approach detects well all the types of considered antipatterns based on a combination QoS and structural metrics(RQ1).

3.4.3.2 Results for RQ2

The goal of the second research question is to investigate how well BLMO performs against random search (RS), our previous mono-objective bi-level work [157], an existing mono-level and single-objective approach (GP) [163] where only defect examples are used (without the consideration of the lower-level algorithm), and an existing detection tool (SODA-W) [185] not based on computational search. All these existing work did not consider the use of dynamic quality of service metrics and they are limited mainly to the interface level static metrics. The Random Search is implemented as a sanity check to justify the need for intelligent search. It has the same structure of our BLOP approach but without the selection and change operators and it is mainly based on random generation of solutions at both levels.

Figures 3.6 and 3.7 report the average comparative results on 30 runs with 95% as confidence level using the Wilcoxon rank sum test.The confidence level is the threshold to determine if the results are statistically significant or not. RS (random search at both levels using the same fitness functions) did not perform well when compared to BLMO both in terms of precision and recall achieving average less than 50% on the majority of different web service antipattern types. The main reason could be related to the large search-space of possible combinations of metrics and threshold values to explore, and the diverse set of web service defects to detect. Furthermore, the results achieved by BLMO are also better than the mono-objective bi-level and mono-level approaches [163, 157] in terms of both precision and recall. In fact, the mono-objective genetic programming technique have an average between 74% and 79% of precision and recall however BLOP (mono-objective bi-level) has better scores with an average of more than 84% of precision and recall on most of the different web services. Thus, both techniques have lower precision and recall than BLMO. These results confirm that an intelligent search is required to explore the search space and that the use of the mutli-objective search at two levels along with the QoS attributes improved the obtained detection results.

While SODA-W shows promising results with an average precision of 76% and recall of 73%, it is still less than BLMO in all the five considered defect types. We conjecture that a key challenge with SODA-W is that it simplifies the different notions/symptoms that are useful for the detection of certain antipatterns. Indeed, SODA-W is limited to a smaller set of WSDL interface metrics comparing to our approach. In an exhaustive scenario, the number of possible antipatterns to manually characterize with rules can be large and hard to generalize, and rules that are expressed in terms of metric combinations need substantial calibration efforts to find the suitable threshold value for each metric. However, our approach needs only some examples of defects to generate detection rules.

Since our proposed solution is based on bi-level optimization, it is important to evaluate the execution time (T). It is evident that BLMO requires higher execution time than RS, BLOP, GP,and SODA-W since BLMO has an optimization algorithm to be executed at the lower level. To reduce the computational complexity of our BLOP adaptation, we selected only best solutions (10%) at the upper level to update their fitness evaluations based on the coverage of artificial web service antipatterns that are generated by the optimization algorithms executed at the lower level for every selected solution. All the search-based algorithms under comparison were executed on machines with Intel Xeon 3 GHz processors and 8 GB RAM. As described in Figure 3.8, all the existing studies were faster than BLMO. However, the execution for BLMO is reasonable because the algorithm is executed only once then the generated rules will be used to detect antipatterns. There is no need to execute BLMO again except in the case that the base of examples (training set) will be updated with a high number of new web service antipattern examples.

One of the advantages of using our BLMO adaptation is that the developers do not need to provide a large set of examples to generate the detection rules. In fact, the lower-level optimization can generate examples of web service defects that are used to evaluate the detection rules at the upper level. The existing mono-level work of Ouni et al. [163] (GP) require a higher number of defect examples than BLMO to generate good quality of detection rules. We can conclude, based on the obtained results that our BLMO approach outperforms, in average, the state of the art web service antipatterns detection techniques that are not using multi-objective search and QoS metrics (response to RQ2).

3.4.3.3 Results for RQ3

Subjects were first asked to fill out a questionnaire containing five questions. The questionnaire helped to collect background information such as their role within the company, their programming experience, their familiarity with web services. The first part of the questionnaire includes questions to evaluate the relevance of some detected web service antipatterns by BLMO using the following scale: 1. Not at all relevant; 2. Slightly relevant; 3. Moderately relevant; and 4.Extremely relevant. If a detected antipattern is considered relevant then this means that the developer considers that it is important to fix it. The second part of the questionnaire includes questions for those antipatterns that are considered at least "moderately relevant". We asked the subjects to specify their usefulness based on the following options: 1. web services selection; 2. Quality assurance; 3. Bug prediction; 4. Effort prediction; and 5. Refactoring opportunities. The questionnaire is completed anonymously thus ensuring confidentiality and this study were approved by the IRB at the University of Michigan: "Research involving the collection or study of existing data, documents, records, pathological specimens, or diagnostic specimens, if these sources are publicly available or if the information is recorded by the investigator in such a manner that participants cannot be identified, directly or through identifiers linked to the participants".

During the entire process, subjects were encouraged to think aloud and to share their opinions, issues, detailed explanations and ideas with the organizers of the study (one graduate student and one faculty) and not only answering the questions. A brief tutorial session was organized for every participant around web services antipatterns and quality of services to make sure that all of them have a minimum background to participate in the study. The instructions indicate also that the developers need to inspect the source code and the interfaces to evaluate the detected web service antipatterns and their relevance and not by evaluating the quality metric values. In addition, all the developers performed the experiments in a similar environment: similar configuration of the computers, tools and facilitators of the study. These sessions were also recorded as audio and the average time required to finish all the questions was 2 hours.

We evaluated, first, the relevance of a set of 30 detected web service antipatterns (6 instances from each type of antipattern) by the participants. Figure 3.9 illustrates that only less than 17% of detected antipatterns are considered not at all relevant by the developers. Around 68% of the antipatterns are considered as moderately or extremely relevant by the different participants, and this confirms the importance of the detected web service antipatterns for developers.

To better evaluate the relevance of the detected web service antipatterns, we investigated the types of antipatterns that developers perhaps consider them more or less important than others (e.g. RPT, GOWS, etc.). Figure 3.10 summarizes our findings. It is clear that the detected GOWS are considered very relevant for developers. One of the reasons can be the impact of large number of operations on the performance of the services (response time, availability, etc.). In addition, it is very difficult for users to select a relevant operation when the interface contains a very large number of operations. Another interesting observation is that RPT antipatterns are not considered very relevant by developers. It is hard for developers to decide about the relevance of some types of antipattern without checking manually some of the detect ones and understand their possible impact. Thus, we did this post-study questionnaire to ask the developers after using the tool and checking some of the results. It may inform future research about which antipattern types to prioritize.

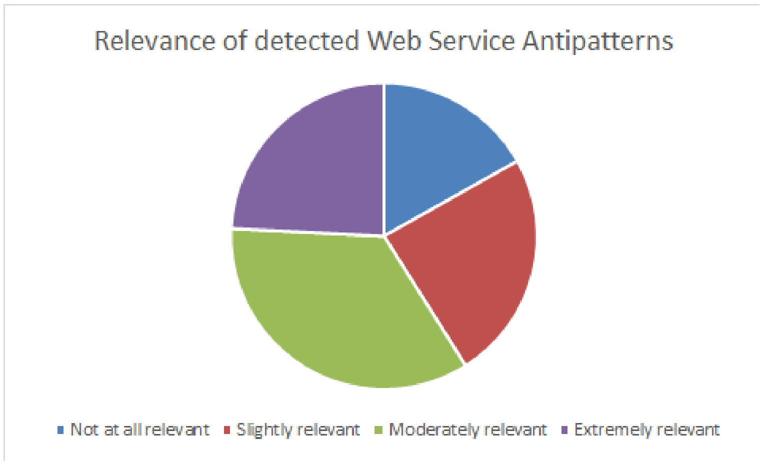


Figure 3.9: The relevance of detected web service antipatterns.

It is also important to evaluate the usefulness of the detected web service antipatterns from the developers perspective. Thus, we asked the participants to justify the usefulness

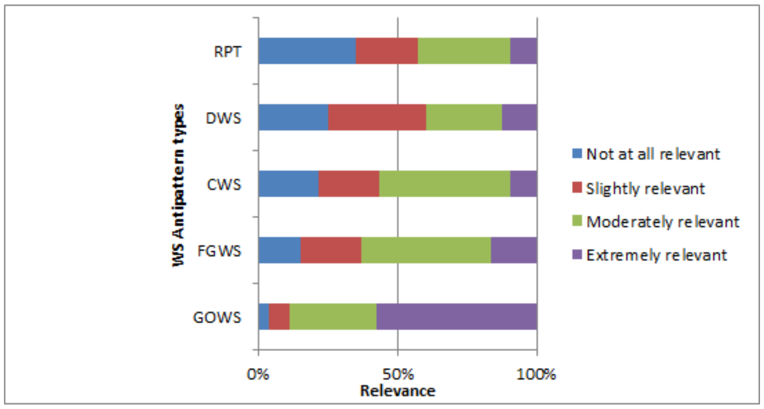


Figure 3.10: Relevance of different types of web service antipattern.

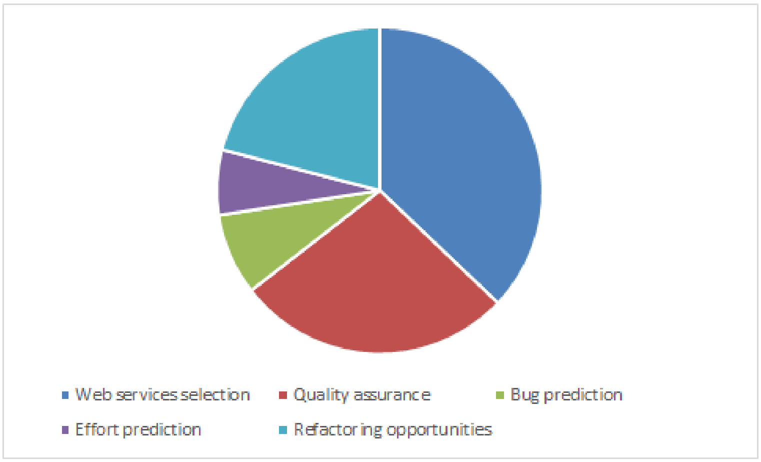


Figure 3.11: The usefulness of web service antipatterns for developers

of the code-smells ranked as moderately or extremely relevant. Figure 3.11 describes the obtained results. The main usefulness is related to web services selection, refactoring guidance and quality assurance. In fact, most of the participants we interviewed found that the detected antipatterns give relevant advices about where refactorings should be applied to fix operations and portTypes. In addition, they found that the web service antipatterns detection process is much more helpful than the traditional analysis of quality metrics to find refactoring opportunities. They consider the use of traditional quality metrics for Quality Assurance as a time consuming process, and it is easier to interpret the results of detected antipatterns and apply the appropriate refactorings to improve the overall quality of the web services.

We summarize briefly in the following the feed-back of the participants during the think aloud sessions. Most of the participants mention that the detection rules generated by our bi-level multi-objective approach represents a faster solution than manual assessment of the quality of web services. The manual techniques represent a time consuming process to calibrate the metrics threshold or the combination of metrics to identify a maintainability issue manually. The participants found the detection rules useful to maintain a good quality of the design of web services. In addition, the developers liked the flexibility to modify the rules (metrics or thresholds) if required. Some possible improvements for our detection techniques were also suggested by the participants. Some participants believe that it will be very helpful to extend the tool by adding a new feature to rank the detected web service antipatterns based on several criteria such as risk, cost and benefits. They believe that current web service quality assessment tools do not provide any support to estimate the risk, cost and benefits of fixing some maintainability issues.

To conclude, the developers found the use of QoS and multi-objective search efficient to detect web service antipatterns and found most of the detected types relevant (answer to RQ3).

3.5 Threats to Validity

One possible construct validity threat arises because although we considered several well-known web services design defect types, we must further evaluate the performance and ability of our bi-level technique to detect other defect types. A construct threat can also be related to the corpus of manually detected web service design defects since developers do not all agree if a candidate is a defect or not. We will ask some new experts to extend the existing corpus and provide additional feedback regarding the detected antipatterns. In addition, the recall score is challenging to calculate by the developers of our experiments and requires additional participants to check its accuracy. A limitation related to our experiments is the difficulty to set the thresholds for some of existing state of the art techniques. In fact, we used the default thresholds used by these techniques that can have an impact on the quality of the results. The evaluation of detected web service defects for some participants is mainly based on the definitions of the antipatterns and the examples that we provided during the pilot study. However, the definition of antipatterns is subjective and depends on the programming behavior of the participants thus this can affect the accuracy of the detection results.

A construct threat is related to the fact that our detection results depend on the examples of antipatterns and well-designed web services. In addition, the generation of artificial web services can lead to several non-useful examples (generated by the lower-level). Additional constraints should be defined to better guide the search at a lower level to refine the generation of artificial web service examples. The same observation is valid for the used change operators at both the upper and lower levels that can generate invalid rules and antipattern examples (e.g. redundancy) may be avoided by the definition of additional constraints.

We take into consideration the internal threats to validity in the use of stochastic algorithms since our experimental study is performed based on 30 independent simulation runs for each problem instance, and the obtained results are statistically analyzed by using the Wilcoxon rank sum test with a 95% confidence level. The parameter tuning of the different optimization algorithms used in our experiments creates another internal threat that we need to evaluate in our future work by additional experiments to evaluate the impact of upper and lower levels' parameters on the quality of the results.

For the selection threat, the subject diversity in terms of profile and experience could affect our study. We mitigated the selection threat by giving written guidelines and examples of antipatterns already evaluated with arguments and justification. Additionally, each group of subjects evaluated different antipatterns from different systems using different techniques/algorithms. Randomization also helps to prevent the learning and fatigue threats. Only few antipatterns per system were randomly picked for the evaluation. Diffusion threat is related to the fact that most of the subjects are from the same university, and the majority know each other. However, they were instructed not to share information about the experience before a certain date.

External validity refers to the generalization of our findings. In our study, external threat to validity concerns mainly the employed defect types and the studied web services. We considered five types of web service antipatterns which constitute a wide representative set of standard and most frequent defects. Likewise, we have selected 662 real world web services belong to different application domains, offer diverse functionalities, have different sizes and were developed by different companies.

3.6 Conclusions and Future Work

We proposed a bi-level multi-objective approach for the web service antipatterns detection problem. In our approach adaptation, the upper level generates a set of detection rules which are a combination of QoS, Interface, and code level metrics, using two conflicting fitness functions. The first objective is to maximize the coverage of both the base of defect examples and artificial defects generated by the lower level and to minimize the coverage of well-designed web service examples. The second objective is to minimize the size of a detection rule. The lower level generates artificial defects that cannot be generated by the upper-level detection rules which will help to generate fitter rules.

We implemented our proposed approach and evaluated it on a benchmark of 662 web services and several common web service antipattern types. The empirical study shows that proposed bi-level multi-objective optimization approach outperforms our previous multi-objective approach, bi-level approach and other state-of-the-art approaches. As part of our future work, we are planning to explore the use of bi-level for the automated repair of detected antipatterns. Additionally, we may consider other techniques such as the concept of generative adversarial networks [186] in generating the artificial defects. We will also work on the prioritization of detected defects due to the large number of potential issues that need to be fixed when improving the quality of Web services. For example, we can evaluate the impact of detected defects on the overall quality of service as a way to rank the identified antipatterns. Furthermore, an empirical study about the impact of different antipatterns on QoS (maintainability, changeability, comprehensibility, discoverability) is considered as part of our future work.

CHAPTER IV

Enabling Decision and Objective Space Exploration for Interactive Multi-Objective Refactoring

4.1 Introduction

With the ever-growing size and complexity of software projects, there is a high demand for efficient refactoring [9] tools to improve software quality, reduce technical debt, and increase developer productivity. However, refactoring software systems can be complex, expensive, and risky [13, 14, 15]. A recent study [23] shows that developers are spending considerable time struggling with existing code (e.g., understanding, restructuring, etc.) rather than creating new code, and this may have a harmful impact on developer creativity.

Various tools for code refactoring have been proposed during the past two decades ranging from manual support [24, 25, 26] to fully automated techniques [27, 28, 29, 30, 31, 32, 33, 34, 35, 36]. While these tools are successful in generating correct code refactorings, developers are still reluctant to adopt these refactorings. This reluctance is due to the tools' poor consideration of context and developer preferences when finding refactorings[37, 38, 29, 39]. In fact, the preferences of developers ranging from quality improvements to code locations, are still not well supported by existing tools and a large number of refactorings are recommended, in general, to fix the majority of the quality issues in the system.

In our recent survey, supported by an NSF I-Corps project, with 127 experienced developers in software maintenance at 38 medium and large companies (Google, eBay, IBM, Amazon, etc.) [1, 2], 84% of face-to-face interviewees confirmed that most of the existing automated refactoring tools detect and recommend hundreds of code-level issues (e.g., antipatterns and low quality metrics/attributes) and refactorings. However, these tools do not specify where to start or how they relate to a developer's context (e.g., the recently changed files) and preferences in terms of quality targets. This observation is consistent with another recent study [187]. Furthermore, refactoring is a human activity that cannot be fully automated and requires a developer's insight to accept, modify, or reject recommendations because developers understand their problem domain and may have a clear target design in mind. Several studies reveal that automated refactoring does not always lead to the desired architecture even when quality issues are properly detected, due to the subjective nature of software design choices [188, 36, 189, 33, 38, 35, 190]. However, manual refactoring is often error-prone and time-consuming [42, 156].

Several studies have been proposed recently to have developers interactively evaluate refactoring recommendations [3, 187, 191, 1, 2]. The developers provide feedback about the refactored code and may introduce manual changes to some of the recommendations. However, this interactive process can be expensive since developers must evaluate a large number of possible refactorings and eliminate irrelevant ones. Both interactive and automated refactoring approaches have to deal with the challenge of considering many quality attributes for the generation of refactoring solutions. One of the most commonly used quality attributes are the ones of the Quality Metrics for Object Oriented Designss (QMOODs) model including reusablitiy, extensibility, effectiveness, etc [192]. QMOOD was empirically validated by many studies, based on hundreds of open source and industry projects, to ensure that they are associated with the qualities they are supposed to measure and that they are also conflicting [193, 188, 83].

Refactoring studies have either aggregated these quality metrics to evaluate possible code changes or treated them separately to find trade-offs [188, 187, 191, 36, 189, 33, 190, 84]. However, it is challenging to define weights upfront for the quality objectives since developers are often unable to express them. Furthermore, the number of possible trade-offs between quality objectives is large, which makes developers reluctant to look at many refactoring solutions—a time-consuming and confusing process. The closest work to this study of Alizadeh et al. [2, 1] shows that even the clustering of non-dominated refactoring solutions based on quality metrics will still generate a considerable number of refactorings to explore. Developers, in practice, combine the use of quality metrics and code locations/files to target when deciding which refactoring to apply. However, existing refactoring tools are not enabling the interactive exploration of both quality metrics and code locations during the refactoring process. The search is beyond just filtering the refactorings but how can the algorithm find better recommendations after understanding the preferences of the users and giving them a good understanding on how the refactorings are distributed if they are interested in improving specific quality objectives.

In this research work, we propose an interactive approach that combines multi-objective search, interactive optimization, and unsupervised learning to reduce developer effort in exploring both objective spaces (quality attributes) and decision spaces (files). As a first step, a multi-objective search algorithm, based on NSGA-II [182], is executed to find a compromise between the multiple conflicting quality objectives and generates a set of non-dominated refactoring solutions. Then, an unsupervised clustering algorithm clusters the different trade-off solutions based on their quality metrics. Finally, another clustering algorithm is applied within each cluster of the objective space based on the code locations where the refactorings are recommended to help developers explore the impact of quality attributes while choosing the code fragments to refactor. The input for the second clustering is generated from the first clustering step, hence both algorithms are hierarchical. In other words, the developer can interact with our tool by exploring both the decision and objective spaces to identify relevant refactorings based on their preferences quickly. Thus, the developers can focus on their regions of interest in both the objective and decision spaces. The developers are, in general, first concerned about improving specific quality attributes then they will look for the refactorings that best target the files related to their current interests and ownership [194, 38]. Therefore, we followed this pattern in our approach by clustering first the objective space then we showed the developers the distribution of the refactorings into different decision space clusters for their preferred objective space cluster.

Our approach takes advantage of multi-objective search, clustering, and interactive computational intelligence. Multi-objective algorithms are powerful in terms of diversifying solutions and finding trade-offs between many objectives but generate many solutions. The clustering and interactive algorithms are useful in terms of extracting developers knowledge and preferences. Existing interactive search-based software refactoring techniques are mainly limited to objective space exploration without considering the decision space.

To evaluate our approach, we selected active developers to manually evaluate the effectiveness of our tool on 6 open source projects and one industrial system. Our results show that the participants found their desired refactorings faster and more accurately than the current state of the art of refactoring tools. This confirms our hypothesis that the second level of clustering (decision space) can help developers to quickly find relevant refactorings based on their preferences in terms of both quality objectives to improve and the location of these changes. A video demo of our interactive refactoring tool can be found at [195].

The main outcome of this contribution can be summarized as follows:

1. To the best of our knowledge, this contribution introduces one of the first search-based software engineering techniques that enables the interactive exploration of the objective and decision spaces while existing work focus only on either the objective space or the decision space and they often lack user interaction in the decision space. Our approach is not about a simple filtering of the refactorings based on the locations/files or a clustering of the Pareto front based on the locations. We enabled programmers to interactively navigate between both objective and decision spaces to understand how the refactorings are distributed if they are interested to improve specific quality objectives. Then, our approach can generate even more relevant suggestions after

extracting that knowledge from the exploration of the Pareto front.

2. Our contribution is beyond the adoption of an existing metaheuristic technique to refactoring. The proposed approach includes a novel algorithm to enable the exploration of both decision and objective spaces by combining two level of clustering algorithms with multi-objective search.

3. We implemented and validated our framework on a variety of open source and industrial projects. The results support the hypothesis that the combination of both the objective and decision spaces significantly improved the refactoring recommendations.

4.2 Interactive Refactoring Challenges

Refactoring is a human activity that is hard to automate due to its subjective nature and the high dependency on context. While successful tools for refactoring have been created, several challenges are still to be addressed to expand the adoption of refactoring tools in practice. To investigate the challenges associated with current refactoring tools, we conducted a survey, as part of an NSF I-Corps project, with 127 professional developers at 38 medium and large companies including eBay, Amazon, Google, IBM, and others [2, 1]. All these developers had a minimum of 11 years of experience in software maintenance tasks and especially refactoring. 112 face-to-face meetings were conducted based on semi-structured interviews to understand the challenges that developers are facing with existing refactoring tools.

From these interviews and our extensive industry collaboration, we learned that architects usually have a desired design in mind as a refactoring target, and developers need to conduct a series of low-level refactorings to achieve this target. Without guiding developers, such refactoring tasks can be demanding: it took one software company several weeks to refactor the architecture of a medium-size project (40K Lines of Codes (LOCs)) [2]. Several books [196, 13, 9] on refactoring legacy code and workshops on technical debt present the substantial costs and risks of large-scale refactorings. For example, Tokuda and Batory [197] proposed different case studies with over 800 applied refactorings, estimated to take more than 2 weeks.

There are two major strategies for refactoring in practice: (a) root-canal refactoring and (b) incremental refactoring. The root-canal refactoring is when project owners decide to heavily refactor their system, since some major issues were observed such as the inability to add new features without introducing clones. While root-canal refactoring is less frequent than incremental refactoring, it is still very important in practice. It is currently a major challenge in the software industry, especially with legacy systems such as the ones that we observed at Ford, eBay and so on.

The majority of the interviewees emphasized that root-canal refactoring to restructure the whole system is rare and they are mainly interested in refactoring files that they own rather than files owned by their peers. Refactoring is a complex problem and there are many reasons for why developers may adopt recommended refactorings, among them ownership and code metrics. Note that ownership does not mean a lot in the context of root-canal refactoring (unlike incremental refactoring) since developers may refactor code even though they do not own it. Most existing refactoring tools do not offer a capability of integrating developers' preferences, in terms of which files they may want to refactor, and purely rely on potential quality improvements. Fully automated refactoring usually do not lead to the desired architecture, and a designer's feedback should be considered. Moreover, prior work [198] shows that even some semi-automated tools are underutilized by developers. Over 77% of our interviewees reported that the refactorings they perform do not match the capabilities of low-level transformations supported by existing tools, and 86% of developers confirmed that they need better design guidance during refactoring: *"We need better solutions of refactoring tasks that can reduce the current time-consuming manual work. Automated tools provide refactoring solutions that are hard and costly to repair because they did not consider our design needs."*

Based on our previous experience on licensing refactoring research prototypes to industry, developers always have difficulties and concerns about expressing their preferences up-front as an input to guide refactoring suggestions. They prefer to get insights from some generated refactoring solutions then decide which quality attributes they want to improve. For instance, the number of code smells that are detected for systems is in the hundreds and we have seen reluctance about up-front selection of code smells for refactoring since it is hard for developers to understand the benefits of fixing these smells. Even worse, developer's preferences are not limited to just the quality metrics and their improvements but also where these refactorings will be applied. Our goal is to reduce the need for these up-front developer preferences since they are hard to define in practice by integrating the user's feedback within the different components of multi-objective algorithm for its next run, as described in section 4.3.5. If the developers are clear about their preferences up-front then they can adjust the fitness functions to target them. Many existing refactoring tools fail to consider the developer perspective, and the developer has no opportunity to provide feedback on the refactoring solution being recommended. Furthermore, as development must halt while the refactoring process executes, fully-automated refactoring methods are not useful for floss refactoring where the goal is to maintain good design quality while modifying existing functionality. The developers have to accept the entire refactoring solution even though they prefer, in general, step-wise approaches where the process is interactive, and they have control of the refactorings being applied. Step-wise approaches, unlike the fully automated ones, involve the developers in the loop so they can accept and reject refactoring solutions and express their preferences, thus they have more flexibility in choosing the final set of refactoring to be applied to the system. Determining which quality attribute should be improved, and how, is never a purely technical problem in practice. Instead, high-level refactoring decisions have to take into account the trade-offs between code quality, available resources, project schedule, time-to-market, and management support.

Based on our survey, it is challenging to aggregate quality objectives into one evaluation function to find good refactoring solutions since developers are not able, in general, to express

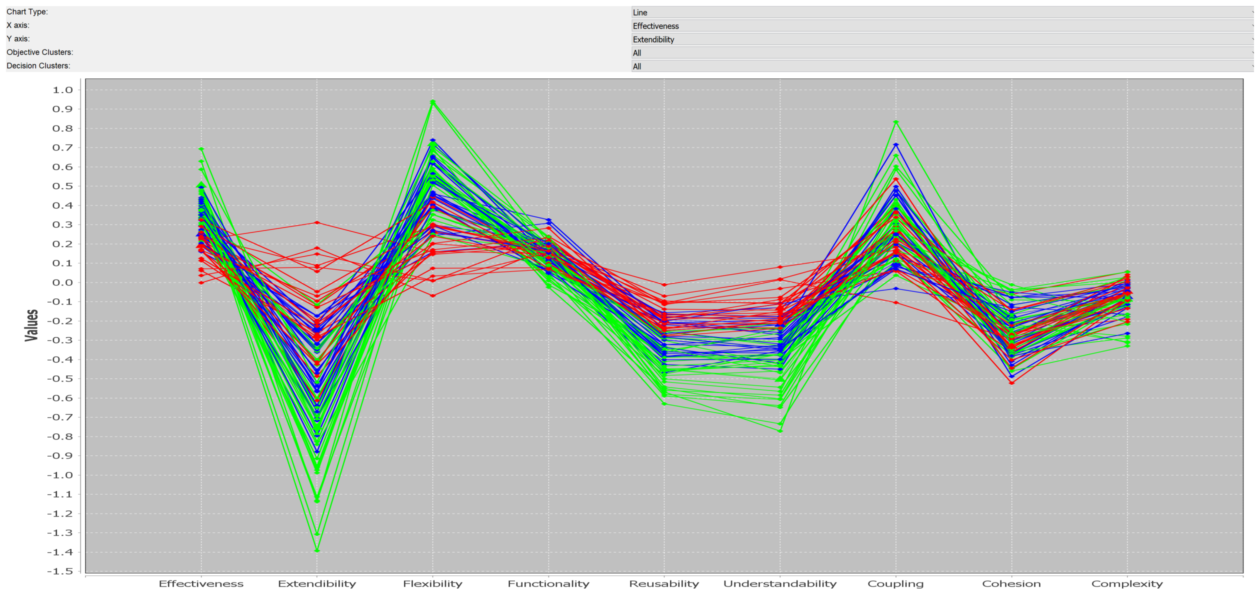


Chart Type: Line
X axis: Effectiveness
Y axis: Extendibility
Objective Clusters: All
Decision Clusters: All

Figure 4.1: The output of a multi-objective refactoring tool [1] finding trade-offs between QMOOD quality attributes on ganttproject v1.10.2 with clustering only in the objective space.

their preferences upfront. While recent advances on refactoring proposed tools support multiple preferences of developers based on multi-objective search, these tools still require the user to navigate through many solutions. Figure 4.1 shows an example of a Pareto front of non-dominated refactoring solutions improving the QMOOD [189] quality attributes of a Gantt Project generated using an existing tool [1]. QMOOD is a widely accepted software quality model, based on our collaborations with industry and existing studies [1, 2, 199, 200, 201, 188, 3]. While developers were interested in giving feedback for some refactoring solutions, they still find the interaction process time-consuming. Even when refactoring solutions are clustered based on the quality objectives, as shown in Figure 4.1, the number of solutions to be checked by developers can be substantial. Thus, they want to know how different the solutions are within the same objective space. It may be possible to find more than one refactoring solution that offers the same level of quality improvements but by refactoring different code locations/files. In fact, the objective space clustering is important for developers to understand which refactorings could help them to achieve their goals of improving specific quality attributes. However, each cluster will still include a considerable number of solutions since each solution contains a good number of refactorings. Thus, the

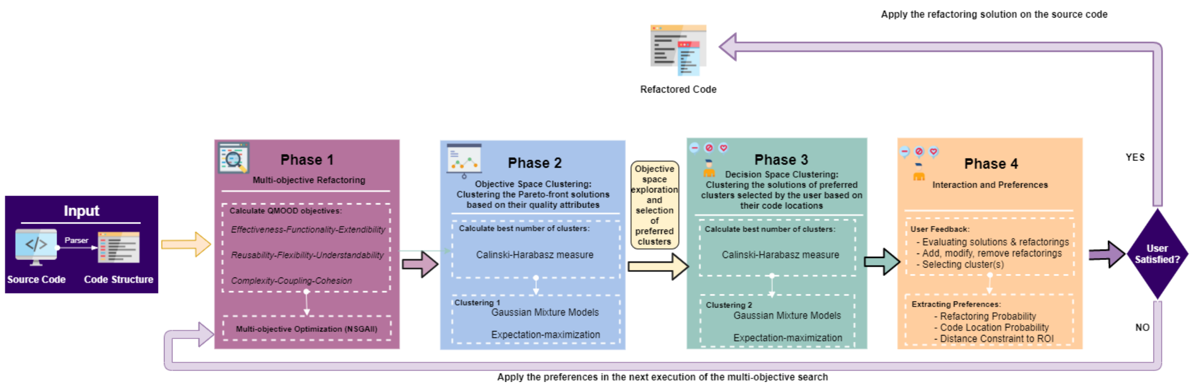

Apply the refactoring solution on the source code
Refactored Code
Input
Parser
Source Code
Code Structure
Phase 1
Multi-objective Refactoring
Calculate QMOOD objectives:
Effectiveness-Functionality-Extendibility
Reusability-Flexibility-Understandability
Complexity-Coupling-Cohesion
Multi-objective Optimization (NSGAII)
Phase 2
Objective Space Clustering: Clustering the Pareto-front solutions based on their quality attributes
Calculate best number of clusters:
Calinski-Harabasz measure
Clustering 1
Gaussian Mixture Models
Expectation-maximization
Objective space exploration and selection of preferred clusters
Phase 3
Decision Space Clustering: Clustering the solutions of preferred clusters selected by the user based on their code locations
Calculate best number of clusters:
Calinski-Harabasz measure
Clustering 2
Gaussian Mixture Models
Expectation-maximization
Phase 4
Interaction and Preferences
User Feedback:
- Evaluating solutions & refactorings
- Add, modify, remove refactorings
- Selecting cluster(s)
Extracting Preferences:
- Refactoring Probability
- Code Location Probability
- Distance Constraint to ROI
User Satisfied?
YES
NO
Apply the preferences in the next execution of the multi-objective search

Figure 4.2: Overview of our proposed approach: DOIMR

objective space clustering is necessary and the decision space clustering is complementary to the first phase. Existing refactoring techniques do not, however, enable developer interaction based on both the decision space and objective space; that is the main challenge of this work. For instance, the objective space exploration can help developers focusing on their targeted design quality improvements then the decision space can help them to focus on files they are owning or related to their current tasks or interests.

4.3 Approach Description: Enabling Decision and Objective Space Exploration for Interactive Multi-Objective Refactoring

Figure 4.2 describes our proposed approach which is composed of four major steps. In the first step, a multi-objective search algorithm is executed to find a set of non-dominated solutions between different conflicting quality objectives of QMOOD [129]. Then, the second step clusters these solutions based on these quality attributes. We call this procedure "objective space clustering". The third step takes, as input, every cluster identified from the user's choice in the objective space and execute another unsupervised learning algorithm to cluster the solutions based on their code locations. Hence, we call this "decision space clustering". Finally, developers can interactively choose among the clustered solutions to find a compromise that suits their preferences in both the decision and objective spaces. For instance, developers may select a cluster (from the objective space clustering) that corresponds to their quality improvement preferences. Then, the second clustering will show them how the solutions in the preferred objective space cluster are different in the decision space. For example, the user can easily avoid looking at many solutions that are similar in the decision space (modifying almost the same code locations) based on the second clustering. Note that our algorithm is hierarchical, thus the input of the second clustering algorithm (decision space) is the set of clusters generated by the first clustering algorithm (objective space) that are selected as preferred ones by the user. The multi-objective search algorithm runs for a number of iterations to finally generate refactoring solutions to the user. If the developer is not satisfied with the solutions that are recommended from these iterations, s/he can explore the clustering results and express their preferences and needs; then another run of the multi-objective algorithm will take place for a number of iterations taking into consideration the developer's preferences (more details are presented in the next sections). This process is iterative until the user is satisfied with a final set of refactoring solutions that is aligned with his preferences.

The next sections will explain in further detail the steps of our methodology.

4.3.1 Phase 1: Multi-Objective Refactoring

The search for a refactoring solution requires the exploration of a large search space to find trade-offs between 6 different quality objectives. The multi-objective optimization problem can be formulated mathematically in this manner:

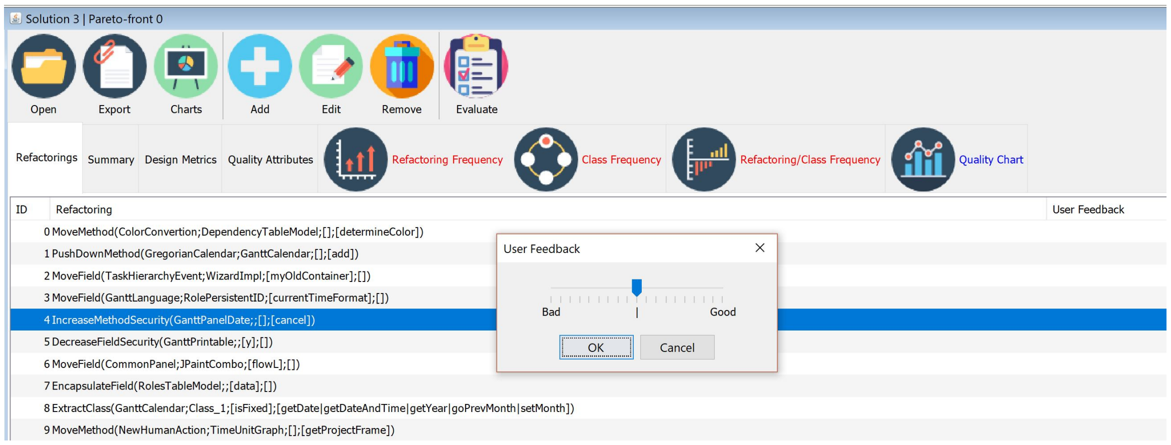
$$\begin{aligned} &Minimize && F(x) = (f_1(x), f_2(x), ..., f_M(x)),\\ &Subject\ to && x \in S,\\ & && S = \{x \in R^m : h(x) = 0, g(x) \geq 0\}; \end{aligned}$$

where S is the set of inequality and equality constraints, g and h are real valued functions defined on S, x is an N vector of decision variables, and the functions f_i are *objective* or *fitness* functions. In multi-objective optimization, the quality of an optimal solution is determined by dominance. The set of feasible solutions that are not dominated with respect to each other is called *Pareto-optimal* or *Non-dominated* set.

In the following subsections, we briefly summarize the adaptation of multi-objective search to the software refactoring problem.

Solution Representation. We encode a refactoring solution as an ordered vector of multiple refactoring operations. Each operation is defined by an action (e.g., move method, extract class, etc.) and its specific controlling parameters (e.g., source and target classes, attributes, methods, etc.) as described in Table 4.3. We considered a set of the most important and widely used refactorings in our experiments: Extract Class/SubClass/SuperClass/Method, Move Method/Field, PullUp Field/Method, PushDown Field/Method, Encapsulate Field and Increase/Decrease Field/Method Security. We selected these refactoring operations because they have the most impact on QMOOD quality attributes [202]. During the process of population initialization or a mutation operation of the algorithm, the refactoring operation and its parameters are formed randomly. Due to the random nature of this process, it is crucial to evaluate the feasibility of a solution meaning to preserve the software behavior without breaking it. This evaluation is based on a set of specific pre- and post-conditions for each refactoring operation as described in [124]. Figure 4.3 shows an example of a concrete refactoring solution proposed by our approach for GanttProject v1.10.2, including several refactorings applied to different code locations.

Fitness Functions. We used the Quality Model for Object-Oriented Design (QMOOD) [192] as a means of estimating the effect of a refactoring operation on the quality of the software. This model is developed based on the international standard for software product quality measurement and is widely used in the industry. QMOOD is a comprehensive way to assess software quality and includes four levels. Using the first two levels—*Object-oriented Design Properties* and *Design Quality Attributes*—as fitness functions, we formulated the



ID	Refactoring	User Feedback
0	MoveMethod(ColorConvertion;DependencyTableModel;[];[determineColor])	
1	PushDownMethod(GregorianCalendar;GanttCalendar;[];[add])	
2	MoveField(TaskHierarchyEvent;WizardImpl;[myOldContainer];[])	
3	MoveField(GanttLanguage;RolePersistentID;[currentTimeFormat];[])	
4	IncreaseMethodSecurity(GanttPanelDate;;[];[cancel])	
5	DecreaseFieldSecurity(GanttPrintable;;[y];[])	
6	MoveField(CommonPanel;JPaintCombo;[flowL];[])	
7	EncapsulateField(RolesTableModel;;[data];[])	
8	ExtractClass(GanttCalendar;Class_1;[isFixed];[getDate|getDateAndTime|getYear|goPrevMonth|setMonth])	
9	MoveMethod(NewHumanAction;TimeUnitGraph;[];[getProjectFrame])	

Figure 4.3: Example of a refactoring solution proposed by our tool for GanttProject v1.10.2.

problem as discovering refactorings to improve the design quality of a software system. The fitness functions we calculate are Understandability, Functionality, Reusability, Effectiveness Flexibility, Extendibility, Complexity, Cohesion, and Coupling. We measured the relative change of these quality attributes after applying a refactoring solution as follows:

$$FitnessFunction_i = \frac{Q_i^{after} - Q_i^{before}}{Q_i^{before}} \tag{4.1}$$

where Q_i^{before} and Q_i^{after} are the value of the quality metric i before and after applying a refactoring solution, respectively.

Table 4.1 and 4.2 describe the QMOOD metrics and their computation formulas used in our optimization approach.

4.3.2 Phase 2: Objective Space Clustering

One of the most challenging and tedious tasks for a user during any multi-objective optimization process is decision making. Since many Pareto-optimal solutions are offered, it is up to the user to select among them, which requires exploration and evaluation of the Pareto-front solutions.

The goal of this step is to cluster and categorize solutions based on their similarity in the objective space. These clusters of solutions help give the user an overview of the options.

Table 4.1: QMOOD metrics and their computation formulas.

QMOOD Metrics	**Definition / Computation**
Reusability	$-0.25 * Coupling + 0.25 * Cohesion + 0.5 * Messaging + 0.5 * DesignSize$
Flexibility	$0.25 * Encapsulation - 0.25 * Coupling + 0.5 * Composition + 0.5 * Polymorphism$
Understandability	$-0.33 * Abstraction + 0.33 * Encapsulation - 0.33 * Coupling + 0.33 * Cohesion - 0.33 * Polymorphism - 0.33 * Complexity - 0.33 * DesignSize$
Functionality	$0.12 * Cohesion + 0.22 * Polymorphism + 0.22 * Messaging + 0.22 * DesignSize + 0.22 * Hierarchies$
Extendibility	$0.5 * Abstraction - 0.5 * Coupling + 0.5 * Inheritance + 0.5 * Polymorphism$
Effectiveness	$0.2 * Abstraction + 0.2 * Encapsulation + 0.2 * Composition + 0.2 * Inheritance + 0.2 * Polymorphism$

Therefore, this technique gives the users more explicit initial exploration steps where they can initiate the interaction by evaluating each cluster center or representative member. Based on our previous refactoring collaborations with industry, developers are always highlighting the time-consuming and confusing process to deal with the large population of Pareto-front solutions: "where should I start to find my preferred solution?". This observation is valid for many Search-based software engineering (SBSE) applications using multi-objective search [2].

Clustering is an unsupervised learning method to discover meaningful underlying structures and patterns among a set of unlabelled data. It puts the data into groups where the similarity of the data points within each group is maximized while minimizing the similarity between groups.

Determining the optimal number of clusters is a fundamental issue in clustering techniques. One method to overcome this issue is to optimize a criterion where we try to minimize or maximize a measure for the different number of clusters formed on the data set. For this purpose, we used the Calinski Harabasz (CH) Index, which is an internal clustering

Table 4.2: Design metrics description.

Design Metric	**Design Property**	**Description**
Design Size in Classes (DSC)	Design Size	Total number of classes in the design.
Number Of Hierarchies (NOH)	Hierarchies	Total number of "root" classes in the design (*count(MaxInheritenceTree (class)=0)*)
Average Number of Ancestors (ANA)	Abstraction	Average number of classes in the inheritance tree for each class.
Direct Access Metric (DAM)	Encapsulation	Ratio of the number of private and protected attributes to the total number of attributes in a class.
Direct Class Coupling (DCC)	Coupling	Number of other classes a class relates to, either through a shared attribute or a parameter in a method.
Cohesion Among Methods of class ($CAMC$)	Cohesion	Measure of how related methods are in a class in terms of used parameters. It can also be computed by: *1 − LackOfCohesionOfMethods()*
Measure Of Aggregation (MOA)	Composition	Count of number of attributes whose type is user defined class(es).
Measure of Functional Abstraction (MFA)	Inheritance	Ratio of the number of inherited methods per the total number of methods within a class.
Number of Polymorphic Methods (NOP)	Polymorphism	Any method that can be used by a class and its descendants. Counts of the number of methods in a class excluding private, static and final ones.
Class Interface Size (CIS)	Messaging	Number of public methods in class.
Number of Methods (NOM)	Complexity	Number of methods declared in a class.

validation measure based on two criteria: compactness and separation [203]. We selected the CH index due to the small size of the number of solutions to cluster (our data), and it is known to provide quick clustering solutions with acceptable quality for similar problems. CH assesses the clustering outcomes based on the average sum of squares between individual clusters and within clusters. Therefore, we execute the clustering algorithm on the Pareto-front solutions with various numbers of components as input. The CH score is calculated for each execution, and the result with the highest CH score is recognized as the optimal clustering.

After determining the best number of clusters, we employ a probabilistic model-based clustering algorithm called "Gaussian Mixture Model" (Gaussian Mixture Models (GMMs)). GMM is a soft-clustering method using a combination of Gaussian distributions with different parameters fitted on the data and more details about this algorithm can be found in [204]. The parameters are the number of distributions, Mean, Co-variance, and Mixing coefficient.

The optimal values for these parameters are estimated using the Expectation-Maximization (EM) algorithm [205]. EM trains the variables through a two-step iterative process.

After the convergence of EM, the membership degree of each solution to a fitted Gaussian or cluster is kept for the preference extraction step. Furthermore, to find a representative member of each cluster, we measure the corresponding density for each solution and select the solution with the highest density.

To calculate the probability distribution function of different Gaussian components, we compute the Mahalanobis distance between data points and its estimated mean vector for all clusters. We allow to choose full covariance matrices in order to model each cluster as an ellipsoid with arbitrary orientation and stretch. In practice, using full covariance matrices improves the performance of the GMM.

4.3.3 Phase 3: Decision Space Clustering

Our approach gives developers the ability to pinpoint their preferences in a different space than the optimization space related to the location of refactorings. In the exploration of the decision space, user preferences are defined for the set of controlling parameters (mainly code elements to be refactored) that each refactoring has (see Table 4.3). After selecting a preferred objective space cluster, the developer may want to see "the distribution of the solutions within that region of interest". In other words, the clustering in the decision space will show developers the refactoring solutions that improve the quality at the same level (within the same objective space cluster) but targeting different parts of the systems. To do this, we group the solutions by their similarity in the decision space and present them to the developer as depicted in Figure 4.4 where only two clusters were found in the decision space. In each of these two clusters, the solutions composing it are introducing refactorings into similar locations with comparable impact on the different quality attributes. These solutions in the decision space are clustered based on the refactoring locations and their frequency. In fact, Figure 4.4 shows the projection on the objective space of the solutions clustered based on the criteria of the decision space (each color is one decision space cluster); a user can click on the preferred solution to see the criteria of the decision space including the code locations. The developer can combine both kinds of information together (impact of the solution on quality and the code locations) to decide which solution to explore further.

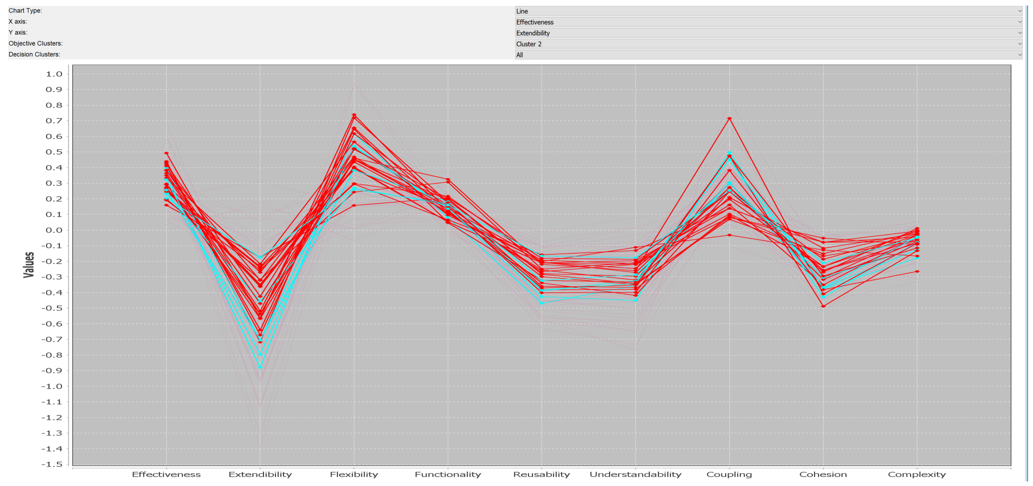


Chart Type: Line
X axis: Effectiveness
Y axis: Extendibility
Objective Clusters: Cluster 2
Decision Clusters: All

Figure 4.4: Clustering based on code locations (decision space) of the refactoring solutions of one region of interest in the objective space of GanttProject v1.10.2.

Table 4.3: Refactoring operations with their controlling parameters.

Refactorings	**Controlling parameters**
Move Method	(sourceClass, targetClass, method)
Move Field	(sourceClass,targetClass,field)
Pull Up Field	(sourceClass,targetClass,field)
Pull Up Method	(sourceClass,targetClass,method)
Push Down Field	(sourceClass,targetClass,field)
Push Down Method	(sourceClass,targetClass,method)
Inline Class	(sourceClass,targetClass)
Inline Method	(sourceClass,sourceMethod,targetClass,targetMethod)
Extract Method	(sourceClass,sourceMethod,targetClass,extractedMethod)
Extract Class	(sourceClass,newClass)

To get an optimal grouping of solutions in the decision space of where refactorings are applied, we use a procedure similar to the one used in the objective space with additional pre-processing steps to project the solutions on the decision space. We define a projection operator based on the frequency of changes to the classes by the refactorings and their locations (refactored files). Since refactoring operations affect classes differently, where some make changes only at the same class level while others have a source class and a target class, we only count source classes in our work to have a consistent representation for all vectors and to create a new representation for the refactoring vector in the decision space. In this new domain space, the solutions are represented as vectors of integers where the refactored classes are the dimensions of the space, and the values are the number of refactoring operations for that class. The projection operator is used for the entire Pareto-front and enables having two different representations of the same solution set. Note that the number of refactored classes depends on the size of the refactoring solutions. Since we considered the same minimum and maximum size thresholds of refactoring solutions for all executions of the algorithm, the time to generate the clusters is similar even for larger projects since the size is not based on all code elements of the project but just those in the refactoring solutions. A larger set of modified code elements may generate more clusters to explore, which can make the interaction more time-consuming. Additionally, the decision space clustering heavily depends on how many code elements are refactored within each solution. If the majority of the solutions in the Pareto front are refactoring almost the same code elements (for instance, one class) then mainly one big cluster will be generated in the decision space. It is true that a large refactoring solution may have a higher probability to modify larger code elements than a smaller one but it is more accurate to estimate the number of possible clusters in the decision space based on the code elements that are refactored by the solutions in the Pareto front.

The main contribution of our work is enabling the exploration of a diverse set of refactoring solutions within the same objectives space. This amounts to having multiple solutions that are neighbors in the objective space but completely different in the decision space. To do this, we go through all the clusters determined in the previous step and then use the GMM clustering algorithm with the same steps described above to group similar solutions in the decision space. Thus, developers can improve the code toward their preferred objectives

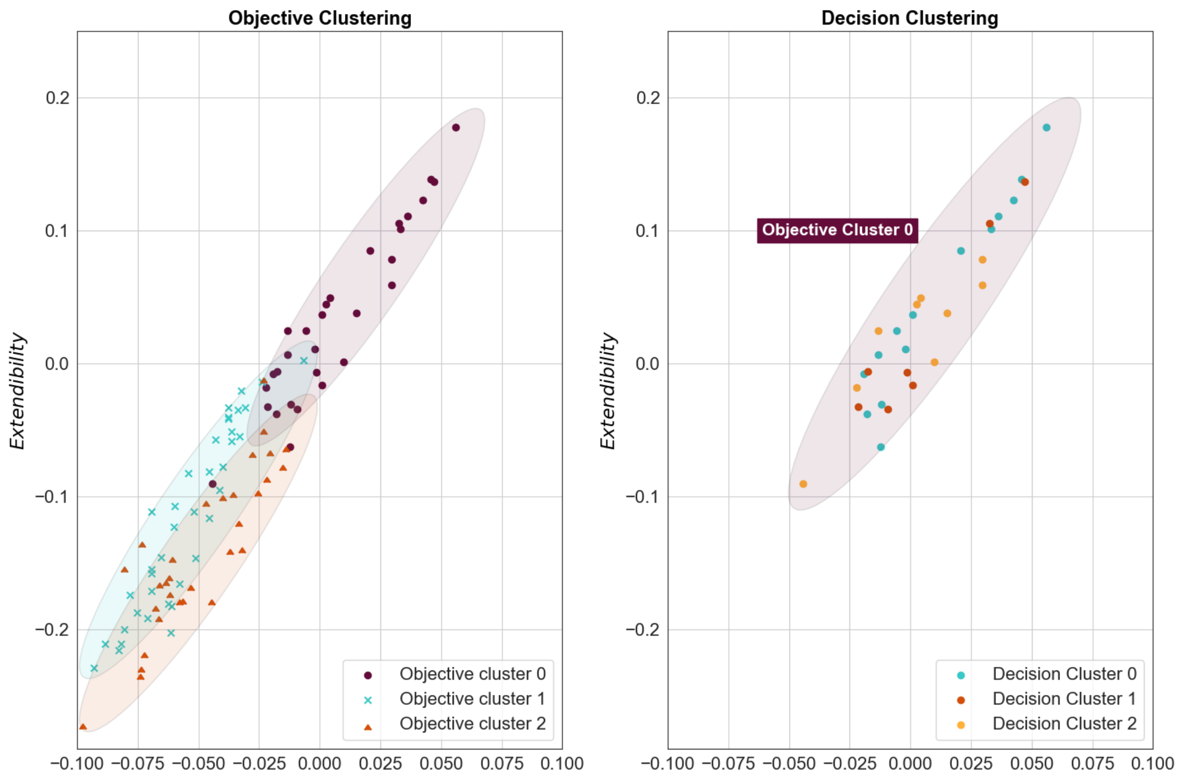


Figure 4.5: Illustration of the clustered solutions in the objective space and the decision space

while only refactoring the parts of the code that interest them.

Figure 4.5 shows an example of our approach (DOIMR) where after generating the Pareto-front for the effectiveness and extendibility objectives, the developer can select a cluster in the objective space for further exploration. Then, a developer can explore the clusters and observe that within this cluster, there are three different clusters in the decision space. The region of interest can be highlighted, and the developer can select solutions that correspond to their interest to create further preferences that can be integrated in the optimization process to converge to the desired optimum. For better visualization of the clustered solutions, our tool offers a feature for two-dimensional views.

4.3.4 Phase 4: Developer Feedback and Preference Extraction

The results of the Bi-Space clustering algorithm are presented to developers in the form of an interactive chart where they can visualize the cluster of their choice in the objective and decision spaces. This presentation helps them get a complete picture of the diversity of the refactoring solutions and the various compromises they may offer. Our goal is to minimize the effort spent by developers to interact with the system and select a final set of refactorings.

Looking at the solutions, developers can evaluate every solution based on their preferences. The granularity offered by our representation enables developers to make evaluations at the cluster level (selecting one or more clusters in the objective space), solution level (selecting solutions within a chosen cluster) and refactoring level (choosing to accept, reject, or modifying some refactorings within the chosen solution as shown in Figure 4.3.). The score obtained reflects developer preferences and serves to determine their region of interest.

At the solution level, the developer is capable of inspecting every refactoring and modifying it. Refactoring operations can be added, deleted, modified, or re-ordered. The information collected afterward is used to calculate a score at the solution level by averaging the scores for every refactoring, and at the cluster levels by averaging the scores of the solutions. The user can reorder the refactorings during the interaction process to fix those that they become invalid, due to the violation of pre-conditions, after removing or modifying other refactorings in the sequence. As described in the solution representation section, these conditions are checked when generating new solutions including the application of change operators. It is possible that the order need to be changed again by the user during the new interactions phase with new solutions since the purpose of reordering is not mainly related to the quality improvements or locations but more to keep the refactoring sequence valid if removing/modifying some refactorings require to change the order again.

We calculate the score of a solution and a cluster after the developer interacts with the solutions and provides his feedback in terms of rejecting, accepting, deleting and reordering the solutions. Thus, the scores are extracted from the developer during every interaction independently. If the new population contains some exact same solutions from the previous interaction then the solution already has the score calculated from the previous interaction.

In this way, we can characterize the developer's region of interest as the cluster with the highest score. Information about the preferred classes, refactorings, and quality metrics is extracted and used to create preferences that can be considered in the optimization process. Therefore, the search becomes guided in both the decision and the objective spaces, and we can converge on a developer's preferred solution faster.

For this purpose, we compute the weighted probability of refactoring operations (RWP) and target classes of the source code (CWP) as follow:

$$RWP_p = \frac{\sum_{s_i \in c_j} \gamma_{ij} \times (|r_p \in s_i|)}{\sum_{r_m \in Ref} \sum_{s_i \in c_j} \gamma_{ij} \times (|r_m \in s_i|)} \tag{4.2}$$

$$CWP_q = \frac{\sum_{s_i \in c_j} \gamma_{ij} \times (|cl_q \in s_i|)}{\sum_{cl_m \in Cls} \sum_{s_i \in c_j} \gamma_{ij} \times (|cl_m \in s_i|)} \tag{4.3}$$

where j is the index of selected cluster, s_i is the solution vector, γ_{ij} is the membership weight of solution i to the cluster j, r is refactoring action, Ref is the set of all refactoring operations, and Cls is the set of all classes in the source code. For every interaction, we compute the probabilities RWP and CWP again without considering previous values because the preferences are already considered when generating the new solutions and we are interested in knowing the developer's feedback about the new solutions thus the new clusters.

4.3.5 Integrating Developer's Feedback

If the user decides to continue the search process, then the generation and selection of the solutions in the next iteration of the multi-objective search is based on (1) the probability formulas of both refactorings and their locations extracted from the preferred decision and objective clusters which are used in the selection step and change operators; and (2) the initial population of the next iteration of the search algorithm which is seeded from the solutions of the preferred cluster. These are the two key factors to integrate user's preferences. More details about the different components of multi-objective optimization are described in the following:

- *Preference-based initial population:* The solutions from preferred clusters will make up the initial population of next iteration as a means of customized search starting point. In this way, we initiate the search from the region of interest rather than randomly. New solutions need to be generated to fill and achieve the pre-defined population size. Instead of random creation of the refactoring operations (refactoring action and target class) based on a unify probability distribution, we utilize RWP and CWP as a probability distribution. In other words, we copy the solutions from the preferred cluster of the previous round and we randomly create new solutions using the probability distributions to reach the expected population size.

- *Preference-based mutation:* We use a bit flip mutation with mutation probability fixed to 0.4. For every solution that is selected to be mutated, instead of randomly selecting refactoring operations and controlling parameters from equally probabilities distribution, we considered preferred refactoring operations which have higher RWP and CWP. The refactorings with higher RWP are the first candidates to be considered for replacing selected refactorings by the mutation operator and the locations with higher CWP are selected for the controlling parameters to be changed for the selected refactorings.

- *Preference-based selection:* the selection operator tends to filter the population and assign higher chance to the more valuable ones based on their fitness values. In order to consider the user preferences in this process, we adjusted this operator to include closeness to the reference solution as an added measure of being a valuable individual of the population. That means the chance of selection is related to both fitness values and distance to the region of interest as:

$$Chance(s_i) \propto \frac{1}{dist(s_i, CR_j)}, Fitness(s_i) \tag{4.4}$$

where $dist()$ indicates Euclidean distance and CR_j is the representative solution of cluster j. The representative solution is the centroid of the preferred cluster. All the of the six used fitness functions are aggregated in $Fitness(s_i)$ by calculating the average. The selection operator is computed on the final region of interest of the developer which includes the results of both decision and objective space clusterings. Since the two clustering algorithms are hierarchical, the cluster j is the user's preferred decision space cluster.

The above-mentioned customized operators aid to keep the stochastic nature of the optimization process and at the same time take the user preferred refactoring and target code locations (classes) into account.

Our proposed approach will help the developer to understand the diversity of the refactoring solutions when visualizing the clusters thus it will help the user to locate her/his region of interest in both the objective and decision spaces. The goal of the interactions and clustering is to gradually reduce the number of refactoring solutions to be explored by the users based on their preferences. If the developer is still interested to apply more refactorings after selecting the final solution, the tool can be re-executed on the new system after refactoring to find other potential solutions.

4.4 Evaluation

4.4.1 Research Questions

We defined three main research questions to measure the correctness, relevance, and benefits of our decision and objective space interactive clustering-based refactoring (DOIMR) tool comparing to existing approaches that are based on interactive clustering-based refactoring only in the objectives space (Alizadeh et al.) [2], interactive multi-objective search (Mkaouer et al.) [3, 1], fully automated multi-objective search (Ouni et al.) [4] and fully automated deterministic tool not based on heuristic search (JDeodorant) [125]. A tool demo of our tool and supplementary appendix materials (questionnaire, setup of the experiments, statistical analyses, and detailed results) can be found in our study's website [1].

The research questions are as follows:

- **RQ1:** Does our approach make more relevant recommendations for developers, as compared to existing refactoring techniques?
- **RQ2:** Does our approach significantly reduce the number of relevant refactoring recommendations and the user interaction effort, as compared to existing interactive refactoring approaches?
- **RQ3: Qualitative Analysis.** To what extent the user preferences, interaction and identified region of interest are similar?

4.4.2 Experimental Setup

We considered a total of seven systems, summarized in Table 4.4, to address the above research questions. We selected these seven systems because they are of reasonable size, have been actively developed over the past 10 years, and have been extensively analyzed by the other tools considered in this work. UTest[2] is a project of our industrial partner used for identifying, reporting, and fixing bugs. We selected that system for our experiments since five developers of that system agreed to participate in the experiments, and they are very knowledgeable about refactoring—they are part of the maintenance team. Table 4.4

[1]A demo and supplementary appendix materials can be found at the following link: https://sites.google.com/view/tse2020decision

[2]SEMA Inc.

Table 4.4: Statistics of the studied systems.

System	**Release**	**#Classes**	**KLOC**
ArgoUML	v0.3	1358	114
JHotDraw	v7.5.1	585	25
GanttProject	v1.10.2	241	48
UTest	v7.9	357	74
Apache Ant	v1.8.2	1191	112
Azureus	v2.3.0.6	1449	117
JFreeChart	v1.0.9	521	170

provides information about the size of the subject systems (in terms of number of classes and KLOC).

To answer RQ1, we asked a group of 35 participants to manually evaluate the relevance of the refactoring solutions that they selected using four other tools. The first tool of Alizadeh et al. is an approach based on only objective clustering of the Pareto front [2], using the interactive multi-objective search. The second tool is an interactive multi-objective refactoring approach proposed by Mkaouer et al. *et al.* [3, 1], but the interactions were limited to the refactorings (accept/reject) and there is no clustering of the Pareto front or learning mechanisms from the interaction data. Thus, the comparison with these tools will help us to evaluate our main contribution that is built on the top of existing multi-objective refactoring algorithms: the combined use of decision and objective space exploration for interactive refactoring. We have also compared our DOIMR approach to two fully-automated refactoring tools: Ouni *et al.* [4] and JDeodorant [125]. Ouni *et al.* [4] proposed a multi-objective refactoring formulation based on NSGA-II that generates a solution to maximize the design coherence and refactoring reuse from previous releases. JDeodorant [125] is an Eclipse plugin to detect bad smells and apply refactorings. As JDeodorant supports a lower number of refactoring types with respect to the ones considered by our tool, we restrict our comparison with it to those refactorings. We used these two tools to evaluate the relative benefits of our interactive features in helping developers identifying relevant refactorings.

We preferred not to use measures such as antipatterns or internal quality indicators as proxies for estimating the relevance of refactorings since the developers' manual evaluation already includes a review of the impact of suggested changes on the quality. Furthermore, not all the refactorings that improve quality attributes are relevant to the developers, which is one of the main motivations of this work. The only rigorous way to evaluate the relevance of our tool is the manual evaluation of the results by active developers. This manual evaluation score, MC, consists of the number of relevant refactorings identified by the developers over the total number of refactorings in the selected solution. Due to the subjective nature of refactoring and the large size of considered systems, it is almost impossible to estimate the recall. There is no unique solution to refactor a code/design; thus, it is challenging to construct a gold-standard for large-systems, which makes calculating the recall very challenging.

Participants were first asked to fill out a pre-study questionnaire containing six questions. The questionnaire helped to collect background information such as their role within the company, their programming experience, and their familiarity with software refactoring. The list of questions of all the questionnaires and the obtained results can be found in the online appendix. Although the vast majority of participants were already familiar with refactoring as part of their jobs and graduate studies, all the participants attended a two-hour lecture on refactoring by the organizers of the experiments. The details of the selected participants can be found in Table 4.5, including their programming experience in years, familiarity with refactoring, etc. These participants were recruited based on our networks and previous collaborations with 4 industrial partners. They all had a minimum of 6 years experience post-graduation and were working as active programmers with strong backgrounds in refactoring, Java, and software quality metrics.

Each participant was asked to assess the meaningfulness of the refactorings recommended after using the five tools on distinct 5 systems (one tool per system), to avoid a training threat to validity. In this case, none of the participants get more familiar with a specific system or a tool during the validation. We have also randomized the order of evaluated tools between the participants to ensure a fair comparison. The participants not only evaluated the suggested refactorings but were asked to configure, run, and interact with the tools on the different systems. The only exceptions were related to the five participants from the industrial partner, where they agreed to evaluate only their industrial software. We assigned tasks to the participants according to the studied systems, the techniques to be tested and developers' experience. Each of the five tools has been evaluated 5 times on each of the seven systems. Thus, the total number of manual evaluations is 175 among all the 7 projects and 5 tools. Our aim is to find a trade-off between the statistical power and reducing the training and fatigue threats. Thus, we asked each participant to evaluate 5 distinct tools on 5 different projects to avoid that their performances will be impacted by the training effect of the system or/and refactoring tool.

To answer RQ2, we measured the time (T) that developers spent to identify the best refactoring strategies based on their preferences and the number of refactorings (NR). Furthermore, we evaluated the number of interactions (NI) required on the Pareto front for all interactive refactoring approaches. This evaluation will help to understand if we efficiently reduced the interaction effort. For this research question, we decided to limit the comparison to only the interactive multi-objective work of Mkaouer et al. [3, 1] and Alizadeh et al. [2] since they are the only ones offering interaction with the users, and it will help us understand the real impact of the decision space exploration (not supported by existing studies) on the refactoring recommendations and interaction effort. However, for the execution time, we compared our tool with non-interactive approaches as well.

TO answer RQ3, our experiments involved the 35 participants where each of the 7 projects is evaluated using the 5 tools however only two of these tools can generate regions of interests (clusters). Thus, we evaluated if the participants selected the same regions of interests on the 7 projects using the two clustering-based interactive tools. We considered two regions of interests are similar/overlapping if the coordinates of their centroid is almost same by calculating the euclidean distance. We have also evaluated the frequency of common refactorings

Table 4.5: Selected Participants.

System	#Subjects	Prog. Exp. Avg. (Years) [Avg-Min-Max]	Refactoring Exp.
ArgoUML	5	[7.5 - 6 - 8.5]	Very High
JHotDraw	5	[8 - 6.5 - 9]	Very High
Azureus	5	[9.5 - 7.5 - 11.5]	High
GanttProject	5	[7 - 6 - 8.5]	High
UTest	5	[15.5 - 13 - 19.5]	Very High
Apache Ant	5	[9 - 6 - 12.5]	Very High
JFreeChart	5	[7 - 6 - 9.5]	Very High

among the selected final solutions by the users to identify any common patterns.

4.4.3 Parameter Setting

It is well known that many parameters compose computational search and machine learning algorithms. Parameter setting is one of the longest standing grand challenges of the field. We have used one of the most efficient and popular approaches for parameter setting of evolutionary algorithms, which is Design of Experiments (DoE). Each parameter has been uniformly discretized in some intervals. Values from each interval have been tested for our application. Finally, we pick the best values for all parameters. Hence, a reasonable set of parameter's values have been experimented. This process is done for each of the studied algorithms while the interactive module is disabled.

The stopping criterion was set to 100,000 evaluations for all optimization and search algorithms to ensure fairness of comparison (without counting the number of interactions since it is part of the users' decision to reach the best solution based on their preferences).

The parameters of the multi-objective algorithm are as follows: Single point crossover probability = 0.7; Bit flip mutation probability = 0.4, where the probability of gene modification is 0.5 and stopping criterion was set to 100,000 evaluations. We also set the initial population size to 100 and utilized Binary selection operator. The minimum and maximum length of solution vectors are limited to 10 and 30, respectively.

Furthermore, we used the maximum number of iterations = 1000 and convergence threshold = 0.0001 for the GMM clustering phase. We calculated these parameters using the same DoE approach in a way to make sure that log likelihood function is converged for all studied systems. For instance we picked the minimum number of iterations that guarantees the convergence of clustering algorithm for all systems.

4.4.4 Results

Results for RQ1. Figure 4.6 summarizes the manual validation results of our DOIMR approach compared to the state of the art, as evaluated by the participants. It is clear from the results that interactive approaches generated much more relevant refactorings, as compared with the automated tools of Ouni et al. and JDeodorant. Among the interactive approaches, DOIMR outperformed the other interactive approaches of Mkaouer et al. and Alizadeh et al. which supports the idea that information that the developer used from the decision space, such the code locations where refactorings were applied and the refactorings frequency, was helpful. On average, for all of our seven studied projects, 91% of the proposed refactoring operations were considered to be useful by the subjects. The remaining approaches have an average of 83%, 71%, 67%, and 56% respectively for Alizadeh et al. (interactive with objective space clustering), Mkaouer et al. (interactive multi-objective approach), Ouni et al. (fully automated multi-objective approach) and JDeodorant (deterministic non-search-based approach). The highest MC score is 100% for the Azureus and Gantt projects, and the lowest score is 91% for the industrial system UTest. This lowest score can be explained by the fact that the participants are very knowledgeable about the evaluated system. The participants were not guided on how to interact with the systems, and they mainly looked at the source code to understand the impact of recommended refactorings.

We found that automated refactorings generate a lot of false positives. Both the Ouni et al. and JDeodorant tools recommended a large number of refactorings compared to the interactive tools, and many of them are not interesting for the context of the developers, and so the developers reject these refactorings, even though they may be correct. For instance, the developers of the industrial partner rejected several recommendations from these automated tools simply because they were related to stable code or code fragments outside of their interests. The majority of them will not change code out of their ownership as well. Furthermore, they were not interested to blindly change anything in the code just to improve quality attributes. Compared to the remaining interactive approaches, we found that some of the refactoring solutions of DOIMR will never be proposed by Mkaouer et al. or Alizadeh et al. since they are selected because of their extensive refactoring on specific code fragments that developers may found essential to improve their quality based on the features included in these classes. In fact, one of the main challenges of multi-objective search is the noise introduced by sacrificing some objectives and trying to diversify the solutions. Thus, the decision space exploration can help the developers know the most diverse refactoring solutions among one preferred cluster in the objective space. Thus, developers did not waste time on evaluating refactoring solutions that are similar but related to entirely different code files.

To better investigate the comparison of our approach to the closest work of Alizadeh et al. based only on the objectives space exploration, we qualitatively evaluated the role of the decision space exploration to increase the relevance of refactoring recommendations. Based on the participants feedback during the post study interviews, 26 of the interviewees highlighted that the final step of the decision space exploration helped them to understand differences between the refactoring solutions targeting their goals such as improving specific quality attributes. It was not practical for them to check all the solutions of the preferred objective space cluster. Thus, the decision space highlighted the solutions that are truly different (modifying different code locations) but still achieving the same levels of quality improvements. For instance, some developers preferred solutions that modified a minimum number of code locations but still reached the same level of quality improvements. Others preferred solutions that modified the files that they owned. Still other developers found the refactorings addressing diverse code locations, including long refactorings sequences, are best since they want to make major changes independently of the cost. And other developers

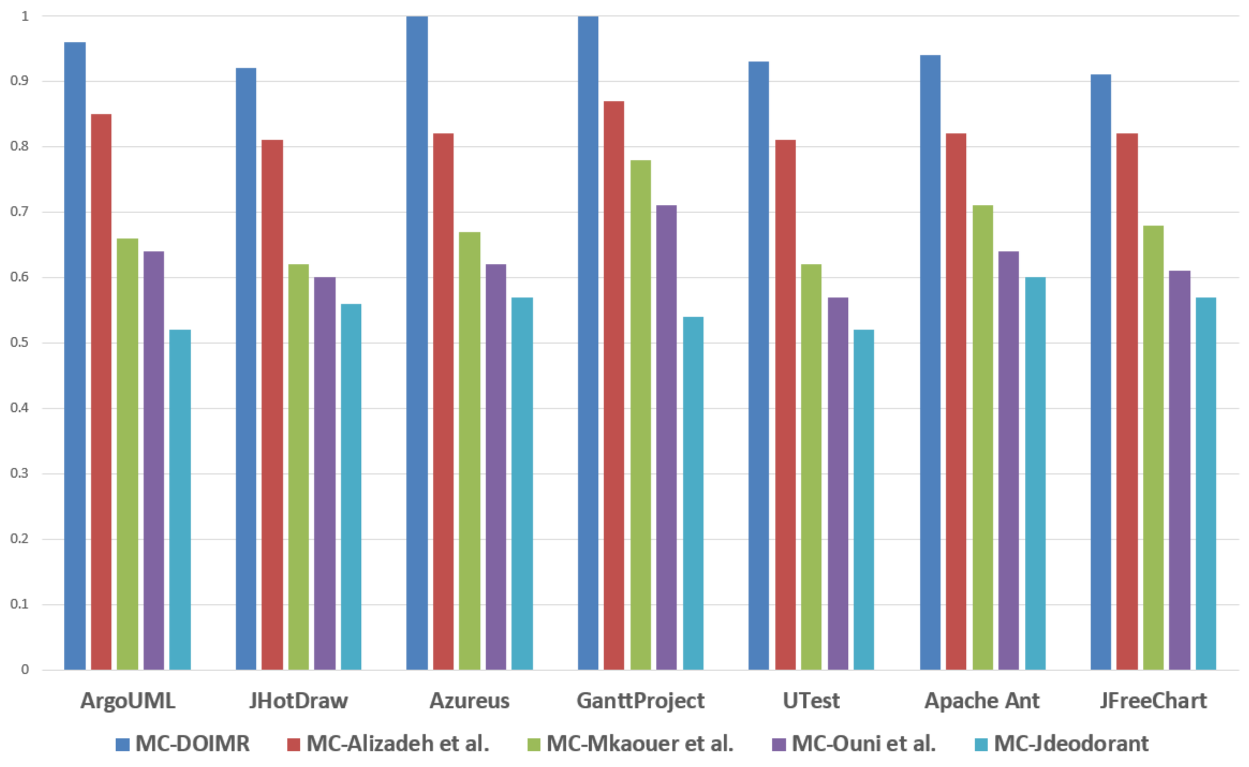


Figure 4.6: Median manual evaluations, MC, on the 7 systems.

selected solutions that can be associated with recent pull-requests or those under review. Thus, the main advantage of the decision space clustering is to help the users understand, with low effort, which refactoring strategy may help them achieve their goal based on their context. For most cases, it was sufficient to look at the center of the clusters to understand the differences between solutions that can target the same objectives.

To conclude, our DOIMR approach outperformed the four other refactoring approaches in terms of recommending relevant refactoring solutions for developers (RQ1).

Results for RQ2. Figures 4.7, 4.8, and 4.9 give an overview of the number of refactorings for the selected solution, number of required interactions, and the time, in minutes, using our tool, the interactive clustering approach of Alizadeh et al., and the interactive multi-objective approach of Mkaouer et al. However, for the execution time, we compared our tool with non-interactive approaches as well. Based on the results of Figure 4.7, it is transparent that our approach significantly reduced the number of recommended refactorings compared

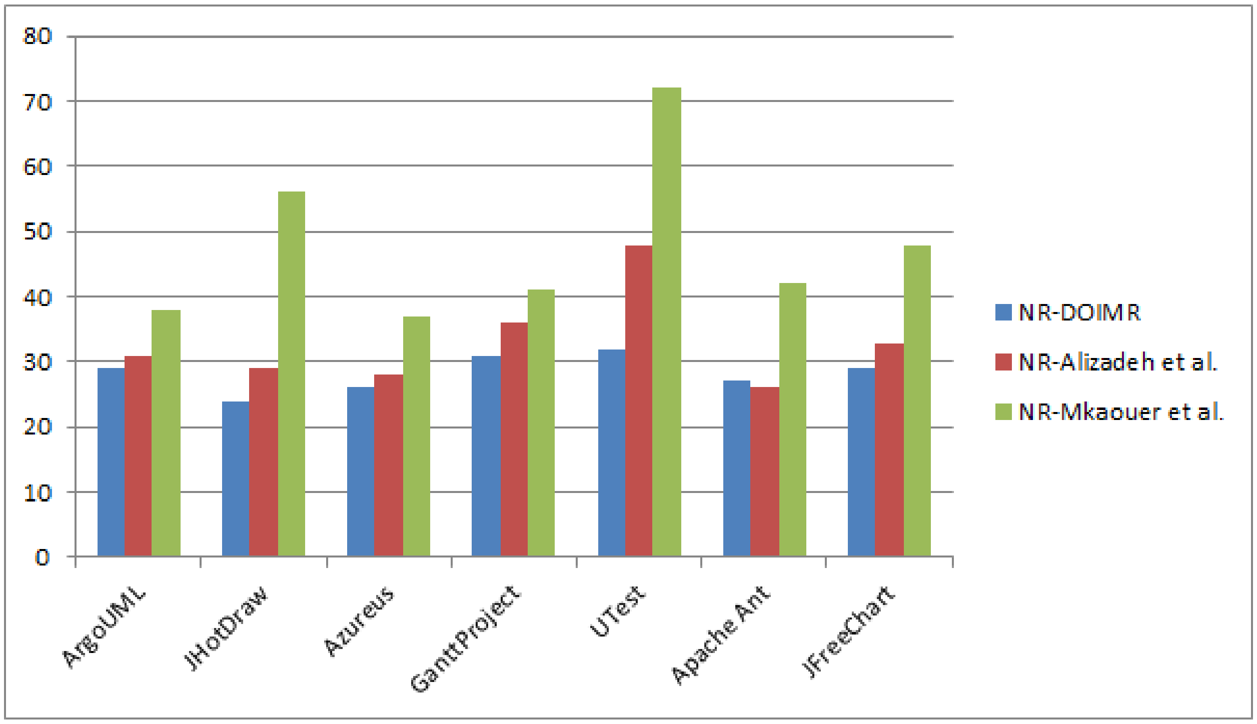


Figure 4.7: The median number of recommended refactorings, NR, of the selected solution on the 7 systems.

to the other interactive approaches while increasing the manual correctness as described in RQ1. The highest number of refactorings was observed on the industrial system with 32 refactorings using DOIMR, 48 using Alizadeh et al. and 72 refactorings using Mkaouer et al. This result may be explained by the size and the quality of this system along with the fact that it was evaluated by some of the original developers of UTest. The lower number of recommended refactorings using DOIMR, compared to the other interactive approaches, is related to the elimination of the noise in multi-objective search not only in terms of objectives but the relevant code locations to be refactored (decision space). It is normal to see fewer refactorings when the search space is reduced to a smaller number of files, which was the case of DOIMR.

Figure 4.8 shows that DOIMR required far fewer developer interactions than the other interactive approaches. For instance, only 13 interactions were required to modify, reject and select refactorings on Azureus using our approach, while 23 and 38 interactions respectively were needed for Alizadeh et al. and Mkaouer et al. The reduction of the number of interactions is mainly due to the smaller number of solutions to explore, after the selection of a preferred cluster in both the objective and decision spaces.

The participants also spent less time to find the most relevant refactorings on the various systems compared to the other interactive and non-interactive approaches, as described in Figure 4.9. The execution time of our approach includes the execution of the multi-objective search, both clusterings, and the different phases of interaction until the developer is satisfied with a specific solution. The execution time of Alizadeh et al. included all the steps of multi-objective search, the objective space clustering, and the interactions while Mkaouer et al. included the multi-objective search and the user interactions. Thus, it is natural that the main differences in the execution time can be observed in the interaction effort. The average time of our approach is reduced by over 40 minutes (70%) compared to Mkaouer et al. for the case of JHotDraw. The reduction of the execution time is mainly explained by the rapid exploration of fewer solutions after looking mainly to the most diverse (different) solutions in the decision space of the preferred cluster in the objective space. In fact, our DOIMR tool has more components (clustering at both objective and decision spaces) than Alizadeh et al. and Mkaouer et al. but the clustering at both spaces significantly reduced the most time-consuming step (user interactions) since the clusterings, and multi-objective search algorithms are quick and executed in few minutes (between 2 and 4 minutes). The execution time is mainly affected by the developer's interaction effort. The developer's interaction effort is not only affected by the number of recommended refactorings, but it is also affected by the solutions that they need to explore and check manually. The decision space clustering of the preferred cluster from the objective space dramatically reduced the number of solutions to check which resulted in fewer interactions. For instance, a user can easily avoid checking many solutions within the same decision space cluster (modifying similar elements) that have similar impacts on the objectives. We note that the execution times included the interaction with the user.

Results for RQ3. Our experiments involved 35 participants where each of the 7 projects

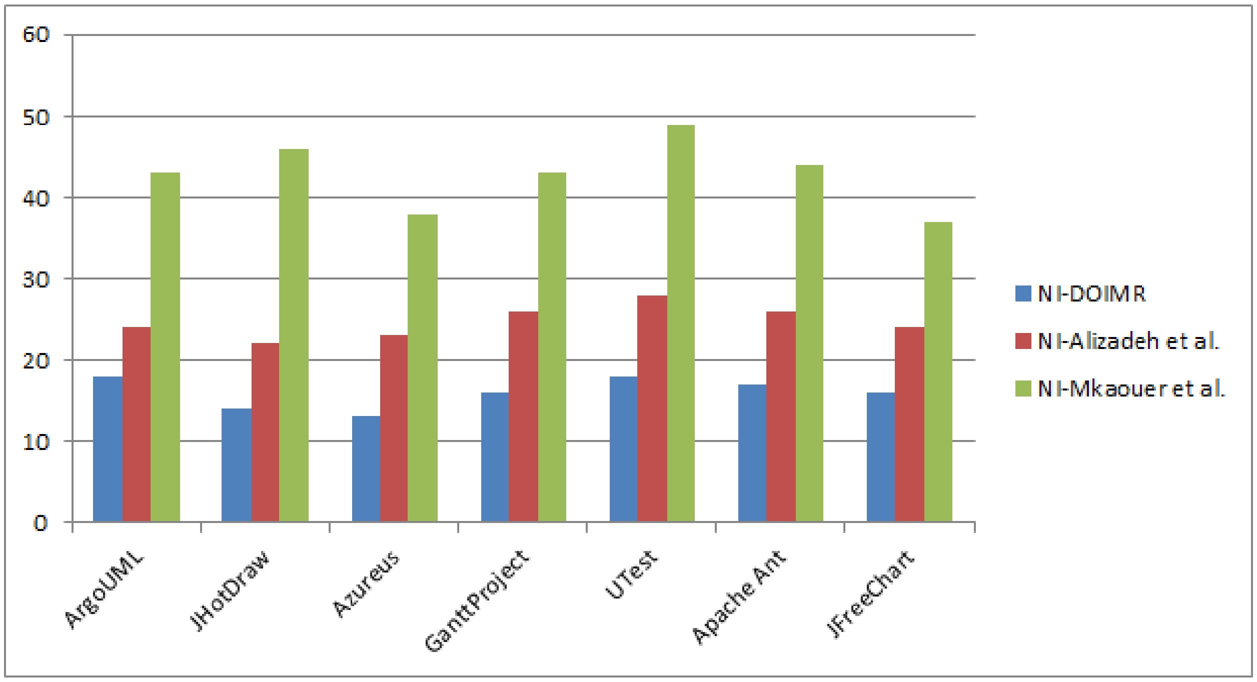


Figure 4.8: The median number of required interactions (accept/reject/modify/selection), NI, on the 7 systems.

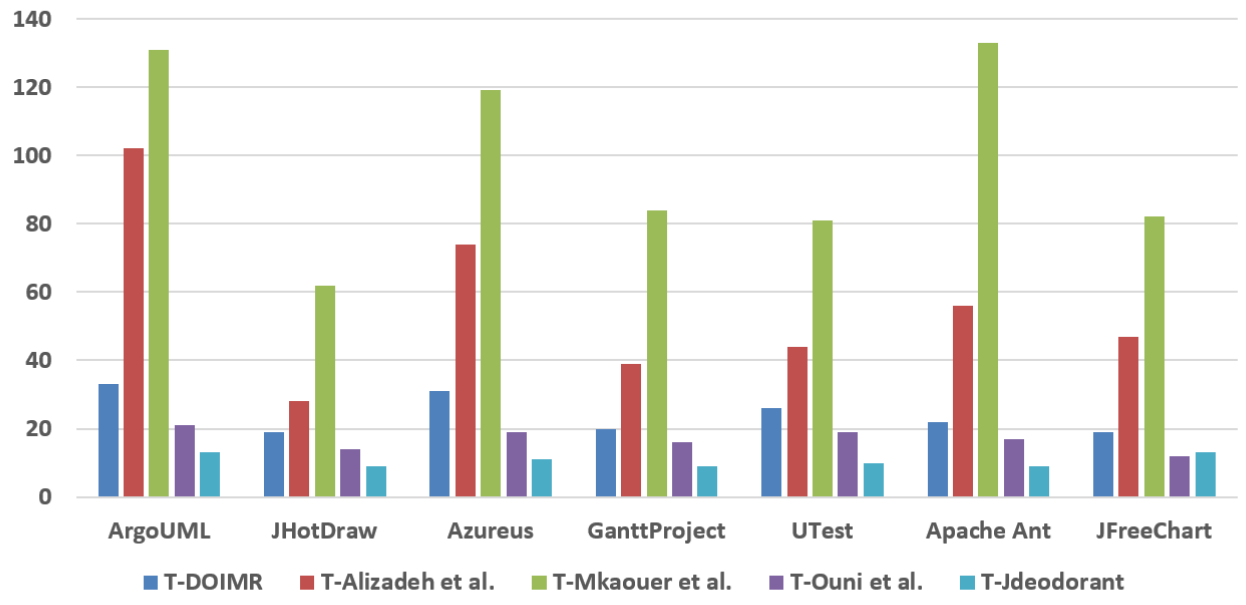


Figure 4.9: The median execution time, T, in minutes on the 7 systems.

is evaluated using the 5 tools however only two of these tools can generate regions of interests (clusters). Thus, we evaluated if the participants selected the same regions of interests on the 7 projects using the two clustering-based interactive tools as shown in Figure 4.10. Note that the minimum number of iterations is 2 for JHotdraw and the maximum is 9 for ArgoUML using our approach where feedback/interactions with the user are recorded. In each of these iterations, the user interacted with the proposed solutions to reject/modify/accept/reorder refactorings. The regions of interests can be only compared in the two tools for the objective space since only our approach generates clusters in the decision space. The overlap measure is calculated based on the number of common clusters that are selected by the participants divided by the total number of selected clusters by the participants. In fact, the overlap measure is the number of the clusters that are selected by multiple users similarity between the clusters for each participant and thus to understand the differences in the developers' preferences. We applied this measure separately on both the decision and objective spaces. The results show that an average of 61% of the selected regions of interests are the same which confirms that the decision space clustering helped developers to select their preferred solution since better refactoring solutions were observed using our approach even when the selected region of interest is the same in the objective space. Another interesting observation is that almost half of the selected region of interest in the objective space are different which means that developers may have different preferences when refactoring systems as explained in the previous comments. We have also checked if multiple participants select the same region of interest in the decision space by looking only at the results of our approach on the 7 projects (5 selected solutions per project using our tool by the participants since each of the 35 participants evaluated 5 distinct tools on 5 different projects). It is interesting to note that the overlap average in the regions of interests at the decision space is higher than the objective space with an average of 71% which can be explained by the fact that the diversity of solutions within a preferred cluster in the objective space is less than the diversity of solutions in the objective space.

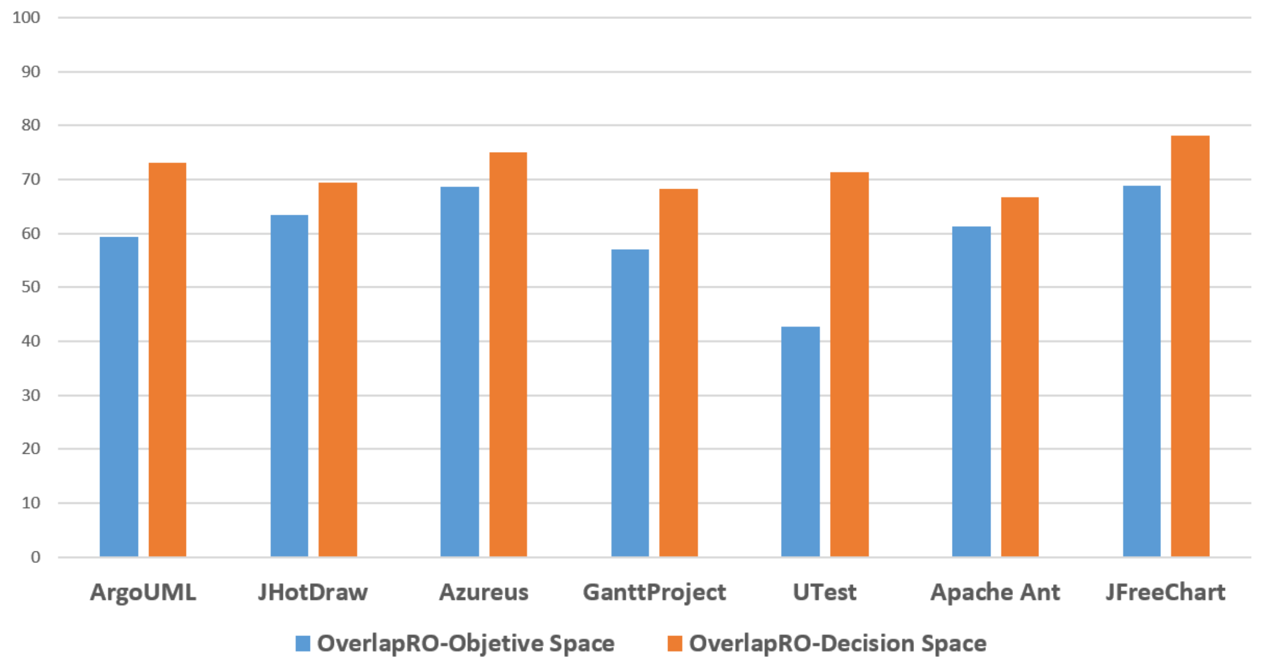


Figure 4.10: region of interest at the objective and decision space levels.

Since the execution of the two clustering algorithms is hierarchical, the final results are actually the combination of two clustering steps. We recorded in our tool all the interactions with the user and we found that all participants used both the objective and decision space clusters before selecting a final solution. In the post-study feedback, participants emphasized that both the decision and objective space interactions helped them to find a relevant solution. The common pattern was to establish their goals from refactoring the code and then they used the decision space to find a solution that matched their context (e.g. code reviews, root-canal refactoring, etc.).

To further investigate the preferences of the participants, Figure 4.11 summarizes the distribution of the refactoring types among the final selected refactoring solutions by the participants. It is clear that the preferred solutions mainly included Extract Class (22%), Move Method (19%) and Extract Method (17%). In fact, the impact of these refactoring types can be positive on many quality attributes such as extendability, reusability, etc.

Since the above results based on the medians are maybe more useful to compare the different interactive approaches, we present and discuss in the following the results per participant

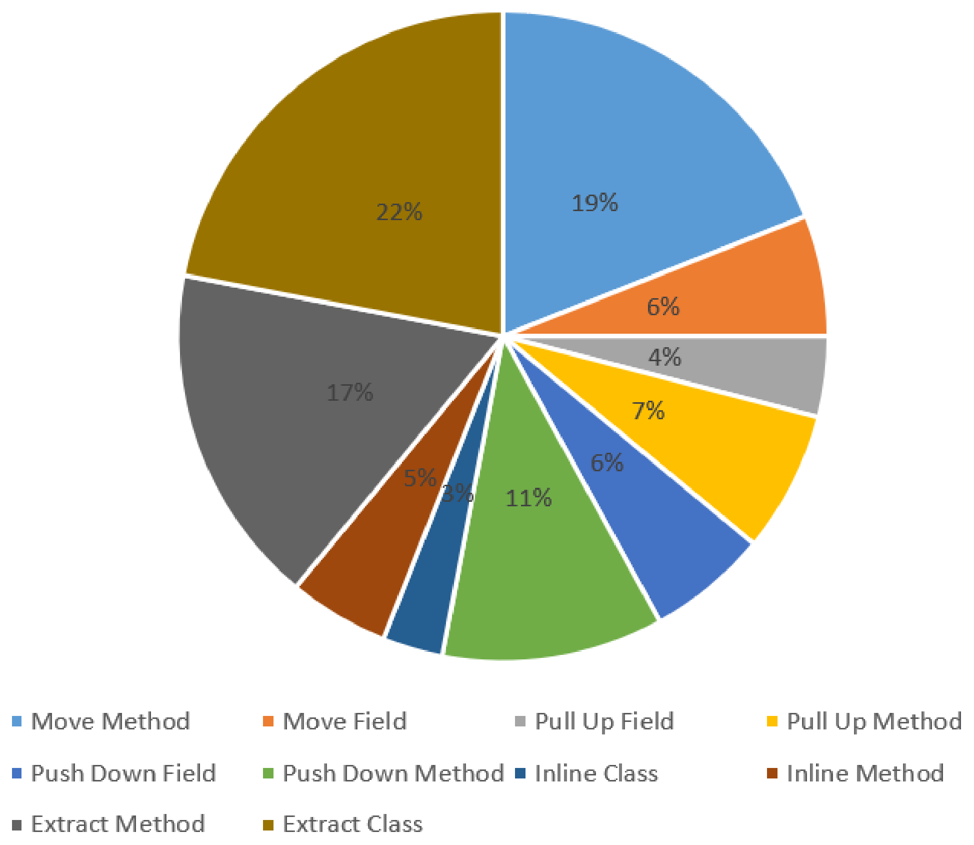


Figure 4.11: Refactoring types distribution among the solutions selected by the user

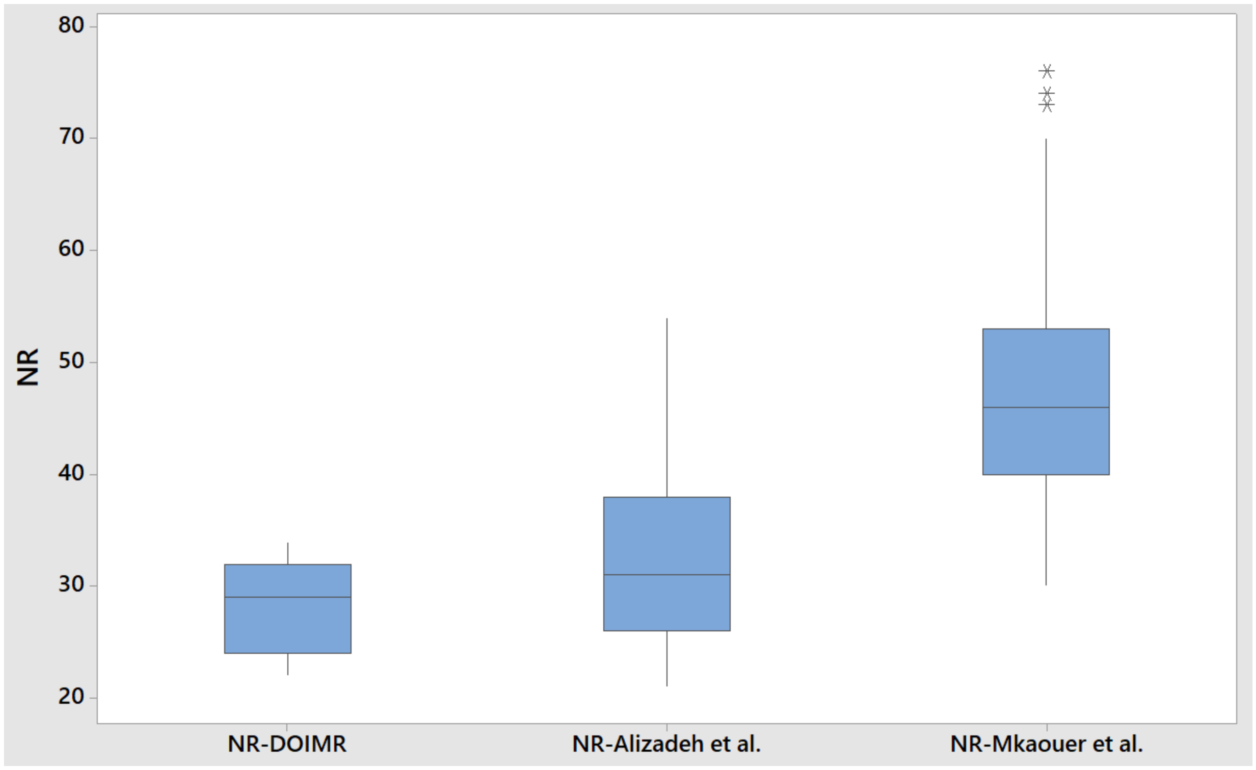


Figure 4.12: Distribution of the number of refactorings in the selected solutions for the 35 participants.

for the number of refactorings in the selected solution, number of required interactions and time spent to find a relevant solution. The box plots of Figure 4.12 shows that the size of the refactoring solutions selected by the participants tend to be similar for our approach (between 25 and 35 refactorings) on the different projects. However, the deviation is high for the approach of Mkaouer et al. but they tend to be larger than the clustering-based approaches which shows the value of using the clusters to better understand the preferences and guide the search towards relevant refactorings. The same observations apply for the time spent by developers and the number of interactions as described in Figures 4.13 and 4.14. In fact, a higher number of interactions will lead to higher time spent by the participants to find relevant refactoring solutions. While the time spent by the participants for using the tool of Mkaouer et al. is diverse, all of them spent more time than our approach for all the projects and participants.

Although the results show the outperformance of interactive approaches compared to

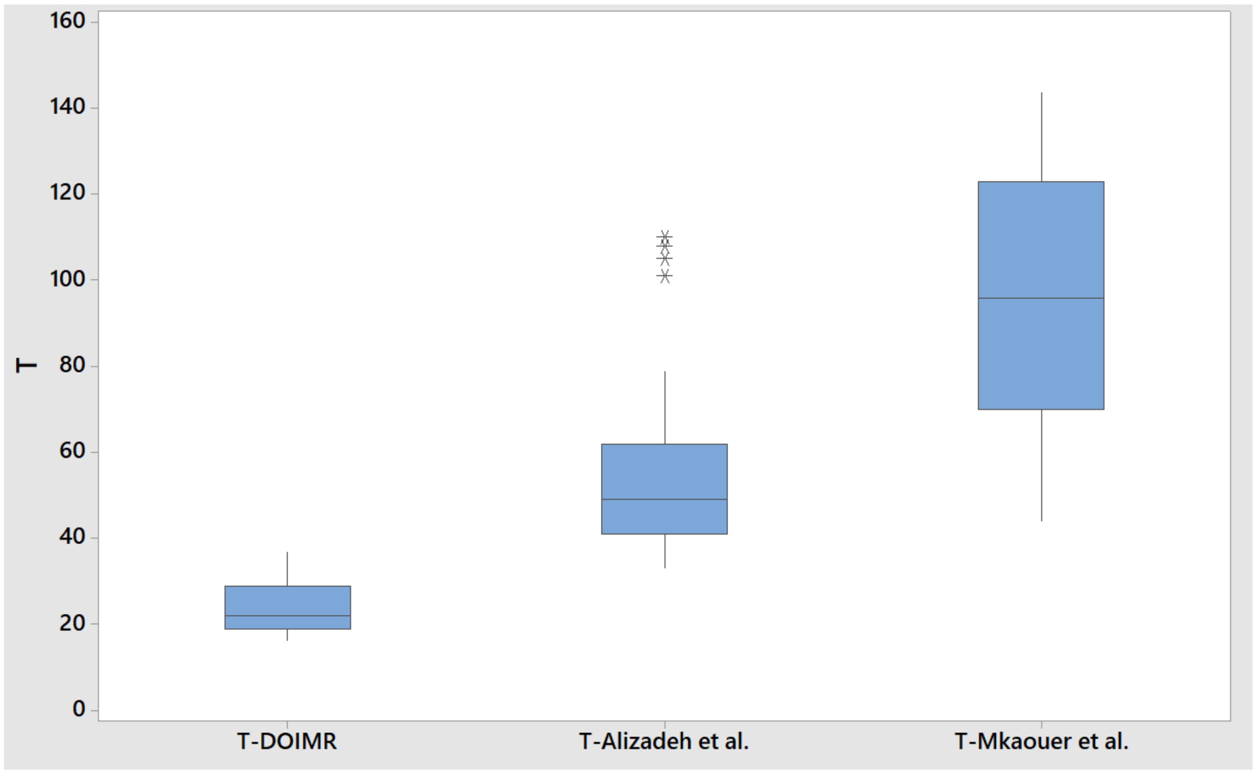


Figure 4.13: Distribution of the time spent to find a relevant solution for the 35 participants.

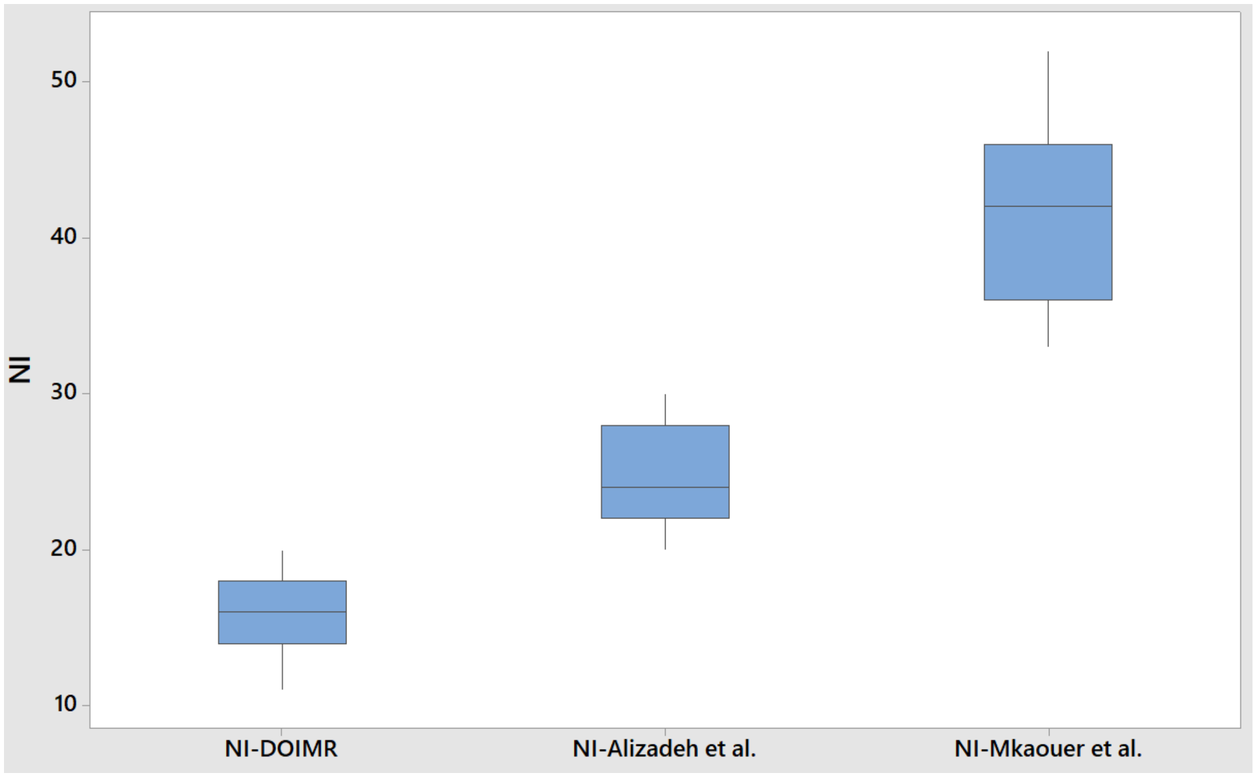


Figure 4.14: Distribution of the number of required interactions for the 35 participants.

automated ones based on different metrics, there are also some limitations related to the use of interactive approaches such as the fatigue despite that our approach significantly reduced the number of iterations with the decision space clustering. It is possible that users can be confused and provide inconsistent feedback which can negatively impact the behavior of the search in the next iterations. The visualization support is also critical to enable relevant feedback from developers to understand the impacts of the recommended refactorings. Another limitation of the interactivity is the difficulty to backtrack some interaction decisions provided to the search algorithm. Finally, the total execution time of interactive approaches is higher than automated ones as described in Figure 4.9.

Statistical Analysis Since meta-heuristic algorithms are stochastic optimizers, they can provide different results for the same problem instance from one run to another. We utilized statistical analysis to perform a comparison between several metaheuristic approaches in this study and to determine the reliability of the results obtained. The following statistical tests show that all the comparisons performed between our approach and existing ones are statistically significant based on all the metrics considered in our experiments.

We used one-way ANOVA statistical test with a 95% confidence level ($\alpha = 5\%$ to find out whether our sample results of different approached are different significantly. Since one-way ANOVA is an omnibus test, a statistically significant result determines whether three or more group means differ in some undisclosed way in the population. One-way ANOVA is conducted for the results obtained from various studied metaheuristic algorithm (independent variable - groups) to investigate and compare each performance metric (dependent variable) on each subject system (software project). We test the null hypothesis (H_0) that population means of each metric is equal for all methods ($\mu_{M1}^{metric} = \mu_{M2}^{metric} = \mu_{M3}^{metric} = \mu_{M4}^{metric}$ $where\ metric \in \{T, NI, NR, MC\}$ against the alternative (H_1) that they are not all equal and at least one method population mean is different.

There are some assumptions for one-way ANOVA test which we assessed before applying the test on the data:

Normal Distribution: Some of the dependent variables were not normally distributed for each method, as assessed by Shapiro-Wilk's test. However, the one-way ANOVA is fairly robust to deviation from normality. Since the sample size is more than 15 and the sample sizes are equal for all groups (balanced), non-normality is not an issue and does not affect Type I error.

Homogeneity of variances: The one-way ANOVA assumes that the population variances of the dependent variables are equal for all groups of the independent variable. If the variances are unequal, this can affect the Type I error rate. There was homogeneity of variances, as assessed by Levene's test for equality of variances ($p > 0.05$).

We have also checked the assumption of IID data within each group. In fact, the residuals from the model are approximately normal since the values are approximately similar. Intuitively, data values are IID if they are not related to each other and if they have the same probability distribution. Thus, the assumption of IID data is verified.

The results of one-way ANOVA tests indicates that The group means were statistically significantly different ($p < .0005$) and, therefore, we can reject the null hypothesis and accept the alternative hypothesis which says there is difference in population means between at least two groups.

The obtained value of F-statistics for each metric are as follows: $F_T = 99.18$, $F_{NI} = 327.41$, $F_{NR} = 40.96$, and $F_{MC} = 102.84$. In one-way ANOVA, the F-statistic is the ratio of variation between sample means over variation within the samples. The larger value of F-statistics represents the group means are further apart from each other and are significantly different. Also, it shows that the observation within each group are close to the group mean with a low variance within samples. Therefore, a large F-value is required to reject the null hypothesis that the group means are equal. Our obtained F-statistics results are correspond to very small p-values.

Since one-way ANOVA does not indicate the difference size, we also calculated the "Vargha-Delaney A" measure [206]. This measure clarifies the effect size (strength of as-

Table 4.6: Vargha-Delaney A measure for different metrics between our method(M1) and others.
Label of the methods: **M1** DOIMR (Our approach), **M2**=Alizadeh et al. [2], **M3**=Mkaouer et al. [3, 1], **M4**=Ouni et al. [4]

	T		NI		NR		MC		
Comparison	**M1-M2**	**M1-M3**	**M1-M2**	**M1-M3**	**M1-M2**	**M1-M3**	**M1-M2**	**M1-M3**	**M1-M4**
ArgoUML	0.94	0.89	0.91	0.86	0.93	0.87	0.91	0.86	0.86
JHotDraw	0.88	0.9	0.84	0.89	0.88	0.91	0.88	0.89	0.88
GanttProject	0.91	0.92	0.87	0.82	0.91	0.94	0.85	0.92	0.86
UTest	0.93	0.84	0.83	0.9	0.93	0.88	0.94	0.84	0.9
Apache Ant	0.89	0.88	0.88	0.81	0.86	0.82	0.9	0.86	0.83
Azureus	0.93	0.82	0.9	0.86	0.83	0.93	0.83	0.91	0.81
JFreeChart	0.83	0.91	0.92	0.83	0.86	0.86	0.92	0.94	0.92

sociation) and it estimates the degree of association between the independent factor and dependent variable for the sample. the A measure is a value between 0 and 1. When it is exactly 0.5, then the two methods achieve equal performance. When A is less than 0.5, the first method is worse, and when A is more than 0.5, the second method is worse. The closer to 0.5, the smaller the difference between the techniques, and the farther from 0.5, the larger the difference.

Table 4.6 shows the "Vargha-Delaney A" results for different metrics between our method and others on each subject system. Since Ouni et al. (M4) is a fully automated multi-objective search without the interactive component, it is only considered for MC metrics. Table 4.6 shows that our approach is better than all the other algorithms with an A effect size that is at least higher than 0.81 for all the 7 systems and the 4 considered metrics (T, NI, NR, MC). For instance, considering the execution time T, we find that our approach (M1) has an A effect size that is higher than 0.91 for ArgoUML,GanttProject, Azureus and UTest; and an A effect size higher than 0.83 for JHotDraw, JFreeChart and Apache Ant. This confirms our findings in RQ2 that the clustering at both spaces significantly reduced the execution time. Same observations apply to reduction of the number of recommended refactorings (NR) and number of interactions (NI). Overall, the high A effect size values for all the metrics and the 7 systems show that the DOIMR outperforms the state of the art refactoring techniques and the outperformance is significant.

4.5 Threats to Validity

Conclusion validity. The parameter tuning used in our experiments creates an internal threat that we need to evaluate in our future work. We have used one of the most efficient and popular approaches for parameter setting of evolutionary algorithms, which is Design of Experiments (DoE). Each parameter has been uniformly discretized in some intervals. Values from each interval have been tested for our application. Finally, we chose the best values for all parameters. Hence, a reasonable set of parameter values have been studied. Another conclusion threat is the number of interactions with the developers since we did not force them to use the same maximum number of interactions which may sometimes explain the out-performance of our approach. Moreover, the developers interacted with the different tools using their offered original graphical interfaces (UIs) which may represent another threat. In fact, developers may perform better with a given tool because it has a better user friendly graphical interface to understand the impact of the refactorings. However, the participants were given the same amount of time to use the tool (limited to three hours).

Internal validity. The variation of correctness and speed between the different groups when using our approach and other tools can be an internal threat since the participants have different levels of experience. To counteract this, we assigned the developers to different groups according to their programming experience to reduce the gap between the groups, and we also adopted a counter-balanced design. Regarding the selected participants, we took precautions to ensure that our participants represented a diverse set of software developers with experience in refactoring, and also that the groups formed had similar average skill sets in terms of refactoring area. To mitigate the training threat, we ensured that the participants (1) did not evaluate the same tool more than one time (even on different projects), (2) did not evaluate the same project more than one time, and (3) we used a random order between the participants for the sequence of tools to be evaluated on different systems. To mitigate the fatigue threat, we allowed participants to perform the experiments in multiple sessions (at least one tool per session).

Construct validity. The developers involved in our experiments may have had divergent opinions about the relevance of the recommended refactorings, which may impact our results. However, some of the participants are the original programmers of the industrial system, which may reduce the impact of this threat. Unlike fixing bugs, refactoring is a subjective process, and there is no unique refactor solution; thus, it is difficult to construct a gold-standard for large systems which makes calculating recall challenging. Does the deviation from an expected refactoring solution mean that the recommendation is wrong or simply another way to refactor the code?

External validity. The first threat is the limited number of participants and evaluated systems, which threatens the generalizability of our results. Besides, our study was limited to the use of specific refactoring types and quality attributes. Furthermore, we mainly evaluated our approach using classical algorithms such as NSGA-II, but other existing metaheuristics can be used. Future replications of this study are necessary to confirm our findings.

4.6 Conclusion

In this research work, we presented a novel way to enable interactive refactoring by combining the exploration of quality improvements (objective space) and refactoring locations (decision space). Our approach helped developers to quickly explore the Pareto front of refactoring solutions that can be generated using multi-objective search. The clustering of the decision space helped the developers identify the most diverse refactoring solutions among ones located within the same cluster in the objective space, improving some desired quality attributes. To evaluate the effectiveness of our tool, we conducted an evaluation with human subjects who evaluated the tool and compared it with the state-of-the-art refactoring techniques. Our evaluation results provide evidence that the insights from both the decision and objective spaces helped developers to quickly express their preferences and converge towards relevant refactorings that met the developers' expectations.

In our future work, we are planning to automatically learn from user interactions for fast convergence to good refactoring solutions. Besides we plan to expand our experiments with more systems and participants.

CHAPTER V

Refactoring Recommendation via Commit Messages Analysis

5.1 Introduction

Software restructuring or refactoring [48] is critical to improve software quality and developer's productivity, but it can be complex, expensive, and risky. As projects evolve, developers in a rush to deliver new features frequently postpone necessary refactorings until a crisis occurs [207]. By that time it often results in degraded performance, an inability to support new features, or even a failed system and significant losses [208, 209, 210]. Thus, several studies have been proposed to (semi-) automate the recommendation of refactorings to help developers improving the quality of their systems in a more timely fashion [211, 212, 28, 42, 37, 32, 213, 35, 214, 36].

While code-level refactoring is widely studied and well supported by tools [215, 216, 214, 1, 217], it remains a human activity which is hard to fully automate and requires developer insights. Such insights are important because developers understand their problem domain intuitively and may have a clear target end-state in mind for their system. A majority of existing tools and approaches rely on the use of quality metrics such as coupling, cohesion, and the QMOOD quality attributes [40] to first identify refactoring opportunities, and then to recommend refactorings to fix them. Many of the quality issues detected using structural metrics are known as code smells or antipatterns [41]. However, recent studies have shown that developers are not primarily interested in fixing antipatterns when they are performing refactoring [42].

In a recent survey of Alizadeh et al. [1, 2] with several software companies, 84% of interviewees confirmed that most of the automated refactoring tools recommend hundreds of code-level quality issues and refactorings, but these tools fail to adequately explain how these refactorings are relevant to a developer who is combining refactorings with other tasks such as fixing bugs and enhancing features. This observation is consistent with other studies [187, 218, **?**] showing that refactorings rarely happen in isolation. Without a rigorous understanding of the rationale for refactoring, recommendation tools may continue to suffer from a high false-positive rate and limited relevance to developers [219, 220, 221]. However, if a refactoring rationale can be automatically identified, this can guide refactoring recommendations to be more relevant and less ad hoc. Recent empirical studies show that while developers document their refactoring intention, they may miss relevant refactorings aligned with their rationale [219, **?**]. One of the main reasons is that manual refactoring is a tedious and time-consuming task which also explains the tendency of the developers to perform the minimum possible number of refactorings [1, 222]. Thus, it is critical to provide developers a semi-automated refactorings support that can understand their rationale and translate it into actionable refactorings recommendation. In this research work, we start from the observation that a majority of inconsistencies between documented and applied refactorings were due to poor refactoring decisions taken manually by developers [219, **?**]. Therefore, we think that there is a need for linking documentation to refactoring recommendations as well as a need for an automated system that can not only check the consistency of the developer-created descriptions of refactoring but also recommend further refactoring to meet their rationale. However, none of the existing studies have used this knowledge to guide the process of refactoring recommendation. Thus, we propose a novel approach, called **RefCom**, to capitalize on this previously unused resource.

Our ultimate goal is to recommend a set of refactoring solutions that enhance the improvements described in the commit messages or provide developers better ways to refactor their code based on the rationale found in the commits. RefCom identifies potential inconsistencies between developer intentions and actual applied refactorings and recommends an additional set of refactorings that better meet developer intentions and expectations. In fact, this contribution validated the first hypothesis that commit messages document refactorings applied by developers including their intention by answering the following research question:

RQ1: *To what extent are refactorings documented in commit messages?*

The second hypothesis validated in this contribution is the inconsistencies (or incomplete refactorings) between documented and applied refactorings in terms of expected impact/intention via answering the following research question:

RQ2: *To what extent do developers accurately document their refactoring and its rationale?*

These observed inconsistencies/gaps (RQ2) along with the fact that refactoring documentation is available at the commit level (RQ1) are the main motivations to refine existing refactoring recommendation tools. Thus, we selected our previous multi-objective refactoring recommendation tool [4] as a case study for this purpose while answering our following third research question:

RQ3: *To what extent can our approach recommend relevant refactorings based on commit analysis compared to existing refactoring techniques?*

However, it is possible to expand the outcomes of RQ1 and RQ2 to build better refactoring recommendation tools in general. To summarize, our contributions are not limited to recommending refactorings solutions using a straightforward multi-objective technique. We believe that RQ1 and RQ2 can advance the knowledge within the refactoring community. For the first two contributions RefCom uses Natural Language Processings (NLPs) and static and dynamic analysis to detect developers' intentions, the actual refactorings and the quality attributes improvement. For the third contribution, we used a multi-objective algorithm to recommend refactoring solutions to enhance the applied refactorings (after extracting developer's intention) or fix the detected inconsistencies.

We validated our approach on six open source projects containing a large number of commits. Our validation shows that RefCom outperforms both the actual refactorings applied by developers in their commits and existing refactoring tools based on antipatterns and static and dynamic analysis [4, 125]. Thus, the use of the knowledge extracted from commit messages is critical to better understand developer preferences.

The primary contributions of this research work can be summarized as follows:

1. This work introduces, for the first time, an approach,

   RefCom, based on commit messages to recommend refactorings. Thus, the recommendations are based on understanding the developers' intention to refactor the code from the commit messages rather than fixing antipatterns and improving the majority of quality metrics.

2. The proposed technique can either: (a) enhance some of the previously refactored files in the commits by providing better alternatives after extracting the refactoring rationale; or (b) recommend refactorings to address the quality issues mentioned in the commit messages when we did not find an actual improvement when checked the files before and after the commit.

3. The presented work reports the results of an empirical study on the implementation of our approach. The obtained manual evaluation results provide evidence to support the claim that our proposed approach is more efficient, on average, than existing refactoring techniques based on a benchmark of 6 open source systems in terms of the relevance of recommended refactorings especially for the case of incremental refactorings.

5.2 Motivation

The primary motivation for our work emerged from our interactions, as part of an NSF I-Corps project, with 127 professional developers at 38 medium and large-size companies including eBay, Amazon, Google, IBM, and others. The main goal of that study was to identify the challenges associated with current refactoring tools. These are discussed next.

Understanding the refactoring rationale is a key for relevant recommendations. Developers lack knowledge of why they should apply the refactorings recommended by existing tools and are frequently overwhelmed by hundreds of automatically generated antipatterns to fix and quality attributes to improve without any indication of their impact on their current context [223, 224, 4, 217]. While existing refactoring approaches are mainly based on static and dynamic analyses to find refactoring opportunities [125, 225], developers may not have the time and motivation to fix every quality issue. For instance, several developers we interviewed [1, 2, 226] mentioned that they are reluctant to apply refactorings on files that they do not "own" or that are not related to their current tasks. Without understanding and detecting developer intentions when they choose to refactor their code, refactoring recommendation techniques will continue to be underutilized [227].

Developers describe and document refactoring opportunities in commit messages. While several empirical studies [228, 229] have shown that over 62% of code reviews discuss maintainability issues to be addressed by refactoring, and only 23% are focused on bug-fixing, most existing work still relies primarily on static and dynamic analyses to identify refactoring opportunities and to explain the need for them. During our survey of industrial partners (for three projects) we found that an average of 38% of quality issues discussed in code reviews and commit messages could *not* be detected using existing traditional static and dynamic analysis tools for code smell detection. As described in Figures 2.2, 2.3 and 2.4, the developer documented their refactoring rationale in terms of improving the coupling that was detected both in the metrics change and the detected refactorings in that commit. Thus, a recommendation refactoring tool can use this information of both the quality attribute to improve and the improved code location (files) to find more refactorings that may fit with the current intention of the developer. But none of the existing studies have used commit message analysis to detect refactoring opportunities or to infer recommendations.

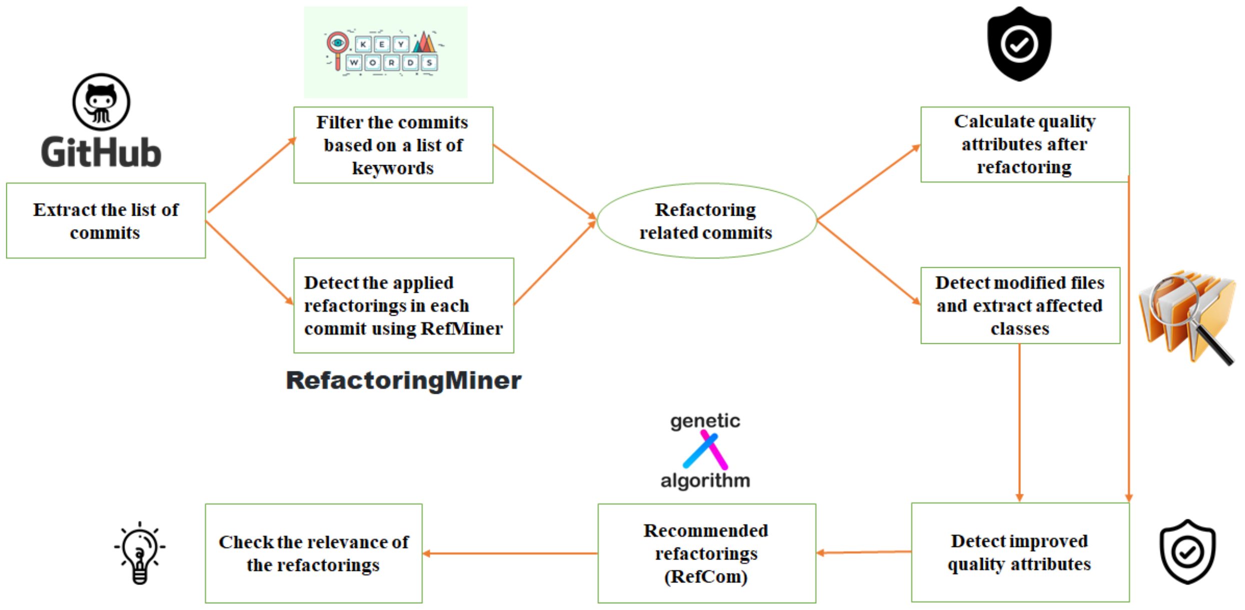

GitHub
Extract the list of commits
Filter the commits based on a list of keywords
KEY WORDS
Detect the applied refactorings in each commit using RefMiner
RefactoringMiner
Refactoring related commits
Calculate quality attributes after refactoring
Detect modified files and extract affected classes
genetic algorithm
Detect improved quality attributes
Recommended refactorings (RefCom)
Check the relevance of the refactorings

Figure 5.1: Approach Overview: RefCom.

Developers may not manually find the best refactoring strategy meeting their needs. Developers need documentation to comprehend refactoring and understand quality changes for code reviews, and to assess technical debt. We found that 46% of the commits in JHotDraw, Xerces, and three projects of one of our industrial partners, eBay, were related to refactoring, as detected using RefactoringMiner [230]. However, 39% of the documentation of their pull-request descriptions or commit messages was inconsistent with the actual quality changes observed in the systems after refactoring. We found that a majority of the inconsistencies in these projects was attributable to poor refactoring decisions taken manually by developers rather than to wrong documentation. Thus we need to link documentation with refactoring recommendations and we need an automated system that can check the consistency of the developer-created descriptions of refactorings and which can also recommend further refactorings for quality changes.

5.3 Approach: RefCom: Commit-Based Refactoring Recommendations

Figure 5.1 gives an overview of our RefCom approach consisting of three main components: the extraction of refactoring-related commits, the identification of refactoring ratio-

Algorithm 4 Commit-based multi-objective refactoring

```
1: Input: Sys: system to evaluate, $P_t$: parent population, Files: detected files from the
   commits analysis, Quality Attributes: detected quality attributes to improve from the
   commits analysis
2: Output: $P_{t+1}$
3: Begin
4: /* Test if any user interaction occurred in the previous iteration */
5: $S_t \leftarrow \emptyset, i \leftarrow 1$;
6: $Q_t \leftarrow Variation(P_t)$;
7: $R_t \leftarrow P_t \cup Q_t$;
8: $P_t \leftarrow evaluate(P_t, C_t, Sys)$;
9: $(F_1, F_2, ...) \leftarrow NonDominatedSort(R_t)$;
10: repeat
11:    $S_t \leftarrow S_t \cup F_i$;
12:    $i \leftarrow i + 1$
13: until $(|S_t| \geq N)$
14: $F_l \leftarrow F_i$; //Last front to be included
15: if $|S_t| = N$ then
16:    $P_{t+1} \leftarrow S_t$;
17: else
18:    $P_{t+1} \leftarrow \cup_{j=1}^{l-1} F_j$;
19:    /*Number of points to be chosen from $F_l$*/
20:    $K \leftarrow N - |P_{t+1}|$;
21:    /*Crowding distance of points in Fl */
22:    $Crowding - Distance - Assignment(F_l)$;
23:    $Quick - Sort(F_l)$;
24:    /*Choose $K$ solutions with largest distance*/
25:    $P_{t+1} \leftarrow P_{t+1} \cup Select(F_l, k)$;
26: end if
27: if $CommitsAnalysis \leftarrow TRUE$ then
28:    /* Select and rank the best front */
29:    $Filter - Solution(F_1, Files, Quality_Attributes)$;
30:    $Recommend - Solution(Commit)$
31: end if
32: End
```

nale from commits (where and why developers applied refactorings) and the recommendation of refactorings based on the extracted rationale from the commits to address the identified quality issues and meet the developer's intention. We describe, in the following, these three main components.

Table 5.1: An example of a solution: sequence of refactorings recommended by RefCom

Operation	Source/entity	Target entity
Move Method	ctrl.booking.BookingController::handleLodgingViewEvent (java.awt.event.ActionEvent):void	ctrl.booking.LodgingModel
Extract Class	ctrl.booking.SelectionModel:: -flightList+ addFlight():void+clearFlight():void	ctrl.booking.FlightList
Move Method	ctrl.booking.BookingController::createBookings():void	ctrl.CoreModel

5.3.1 Refactoring Related Commit Extraction

The refactoring related commits are the union of the results of the RefactoringMiner detection, keywords extraction and QMOOD improvement evaluation. We decided to unify the data from these sources for the following reasons: (1) RefactoringMiner can help to identify the applied refactorings even if they did not improve quality metrics or they were not documented, (2) the keywords extraction can help to detect commits related to refactorings even there were no refactorings detected by RefactoringMiner or no observed quality improvements (inconsistencies detection), and (3) the QMOOD improvements can help not only in identifying commits related to refactoring even if they were not documented in the commits but also in understanding the impact of the applied refactorings. Additionally, we determined that the combination of the keywords, quality changes, and RefactoringMiner is sufficient to filter the commits since we have also manually inspected some of them as well. In fact, we selected the commits that are identified by only one of the three strategies (RefactoringMinder, QMOOD improvements or keywords). We considered commits that are confirmed by at least two out of these three strategies as having already a very high probability to be related to refactorings. Thus, we inspected manually all the commits that are only detected with exclusively one of the three strategies. The total number of commits in that category are around 23% (319 commits).

RefactoringMiner can detect non-documented refactorings in the commit messages, and the use of the keywords is useful to identify the claims and intentions of developers which may not be translated into actual refactorings. The automated check of quality changes can also help to identify refactoring-related commits and check if the developers actually addressed the quality issues described in the commit messages. To summarize, the documented refactorings are in general the ones that are described in the commit messages and eventually could be detected using the keywords. Furthermore, we are able to detect the refactorings related commits using both RefactoringMinder and the QMOOD improvements. In fact, these refactorings related commits may not be described in the commits message but they are detected because they contained identified refactorings or they improved the quality.

5.3.2 Identifying Refactoring Rationale from Commits

Identifying refactoring rationale has two parts. The first part is the detection of the files that are refactored by developers in a commit. The second part is the identification of changes in the QMOOD quality attributes then comparing these changes with the information in the commit message.

For the first part, we used the GitHub API to identify the changed files in each commit. In the second part, we compared the QMOOD quality attribute values before and after the commit to capture the actual quality changes for each file. Once the changed files and quality attributes were identified, we checked if the developers *intended* to actually improve these files and quality attributes. In fact, we preprocessed the commit messages and we used the names of code elements in the changed files and the changed quality metrics as keywords to match with words in the commit message. Once the refactoring rationale is automatically detected using this procedure, we continue with the next step to find better refactoring recommendations that can fully meet the developer's intentions and expectations. In case that no quality changes were identified at all then a warning will be generated to developers that the manually applied refactorings are not addressing the quality issues described in his commit message.

5.3.3 Refactoring Recommendations

After the identification of the refactoring rationale from the history of commits as described in the previous step, we adopted an existing multi-objective algorithm for refactoring [4] to search for relevant refactoring solutions improving both the detected files and changed quality attributes. A refactoring solution, as shown in Table 5.1, consists of a sequence of n refactoring operations involving one or multiple source code elements of the system to refactor. For every refactoring, pre- and post-conditions are specified to ensure the feasibility of the operation [9]. We selected multi-objective algorithm adaptation due to the conflicting quality attributes that are considered in this study. In fact, our adaptation of multi-objective algorithm takes as objectives the 6 QMOOD quality attributes. Furthermore, multi-objective search has the advantage of generating a diverse set of solutions, thus we can filter the recommendations automatically based on the preferred files and quality attributes of the developer (extracted from the commits as described in the previous step) without the need to run the refactoring recommendation algorithm multiple times. For instance, if the refactoring rationale extracted from commits focused on improving both understandability and reusability in specific Class A and Class B, we execute our multi-objective algorithm using all the 6 quality attributes then we filter the Pareto front based on the two main criteria that are contained in the extracted refactoring rationale. First, we make sure that the selected solution is the one that provides the highest improvement in the quality attributes extracted from the commits during our analysis step (e.g. understandability and reusability). Second, the optimal solution should also refactor the detected changed files in the commits (e.g. Class A, Class B.

For more details about the multi-objective refactoring algorithm, the reader can refer to [4].

The adopted multi-objective refactoring tool is based on the non-dominated sorting genetic algorithm (NSGA-II) [231] to find a trade-off between the six QMOOD quality attributes. A multi-objective optimization problem can be formulated as follow :

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & F(x) = (f_1(x), f_2(x), ..., f_M(x)) \\ \text{subject to} \quad & x \in S \\ & S = \{x \in R^m : h(x) = 0, g(x) \geq 0\} \end{aligned} \tag{5.1}$$

where S is the set of inequality and equality constraints and the functions f_i are *objective* or *fitness* functions. In multi-objective optimization, the quality of a solution is recognized by dominance. The set of feasible solutions that are not dominated by any other solution is called *Pareto-optimal* or *Non-dominated* solution set.

NSGA-II is a multi-objective evolutionary algorithm operating on a population of candidate solutions which are evolved toward the Pareto-optimal solution set. As described in Algorithm 4, the first iteration of the process begins with the complete execution of NSGA-II adapted to our refactoring recommendation problem based on the fitness functions representing each of the quality attributes. In the beginning, a random population of encoded refactoring solutions, P_0, is generated as the initial parent population. Then, the children population, Q_0, is created from the initial population using crossover and mutation. Parent and children populations are combined to form R_0. Finally, a subset of solutions is selected from R_0 based on the crowding distance and domination rules. This selection is based on elitism which means keeping the best solutions from the parent and child population. Elitism does not allow an already discovered non-dominated solution to be removed. After the identification of the non-dominated refactoring solutions, we apply a filter on them consisting of the detected changed files from the commit(s) and the desired quality attributes, also extracted from the commit(s). These identified refactorings are assigned to each of the commits that have been modified by the developers.

5.3.4 Running Example

To demonstrate a practical example of our proposed approach, we analyzed a real-world software repository on GitHub. For this purpose, we executed our tool on a repository called "Inception_D". This project provides a semantic annotation platform offering intelligent annotation assistance and knowledge management. It is a large project including over 5000 commits.

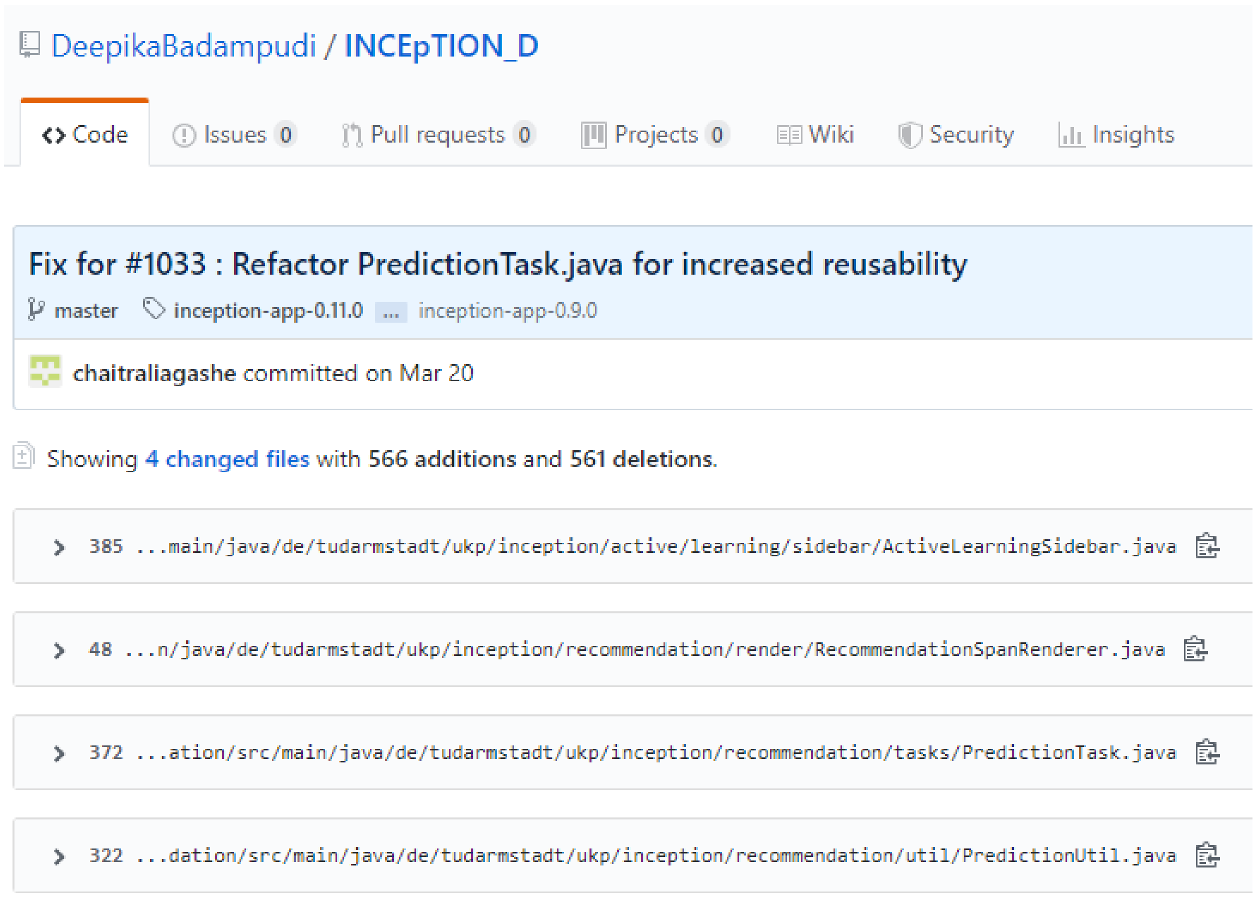
As a first step of our approach, we analyzed and filtered the commits of the mentioned repository and we extracted the refactoring-related commits. Figure 5.2 represents the commit where the developer(s) documented the changes as *" Refactor PredictionTask.java for increased reusability"*. It is clear from the developer's documentation that his intention was to improve the reusability of that class. This information helped in identifying the refactoring rationale. Our refactoring recommendation component takes as an input the modified classes which is, in this commit, "PredictionTask.java" and "Reusability" as a quality attribute to improve. Figure 5.4 shows the list of refactorings that were recommended by our tool to enhance/extend the developer's list of applied refactoring as shown in Figure 5.3. Three out of the four recommended refactoring solutions contained the specific modified file as a parameter. To show the usefulness and the impact of our recommended solutions, RefCom generates charts for comparaison between *the before developer's changes*, *the after developer's changes* and *the after RefCom refactorings* values of each QMOOD quality attributes. Figure 5.5 highlights that RefCom clearly provided much better alternatives than the actual manual refactorings applied by the developer. For instance, the reusability attribute was significantly improved—almost 15 times more than the improvement introduced by the developer's changes.

5.4 Evaluation

5.4.1 Research Questions

To validate our proposed approach, we defined the following three research questions:

- **RQ1.** To what extent are refactorings documented in commit messages?
- **RQ2.** To what extent do developers accurately document their refactoring and its rationale?
- **RQ3.** To what extent can our approach recommend relevant refactorings based on commit analysis compared to existing refactoring techniques?



Fix for #1033 : Refactor PredictionTask.java for increased reusability

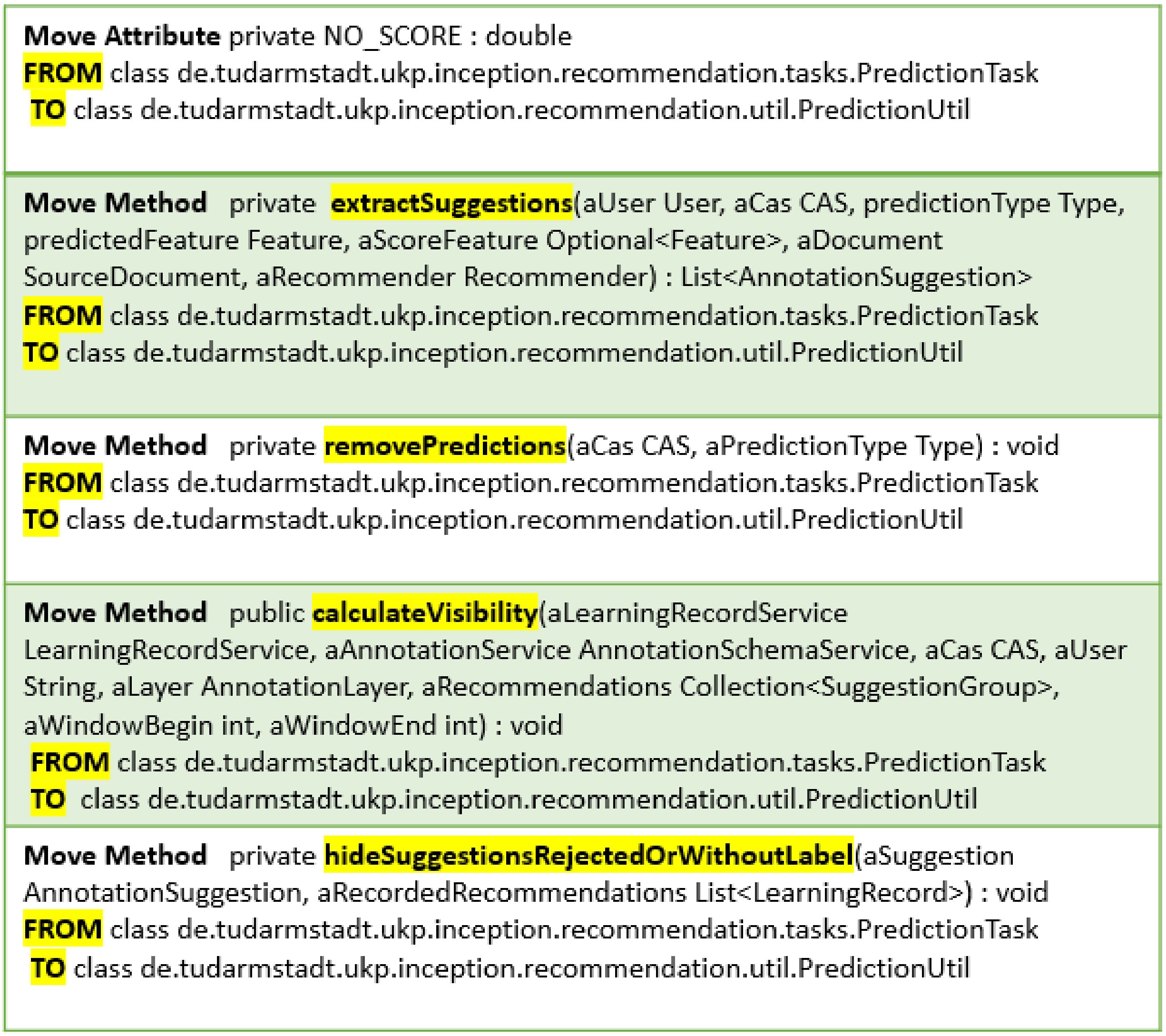
master inception-app-0.11.0 ... inception-app-0.9.0

chaitraliagashe committed on Mar 20

Showing 4 changed files with 566 additions and 561 deletions.

```
385 ...main/java/de/tudarmstadt/ukp/inception/active/learning/sidebar/ActiveLearningSidebar.java
48 ...n/java/de/tudarmstadt/ukp/inception/recommendation/render/RecommendationSpanRenderer.java
372 ...ation/src/main/java/de/tudarmstadt/ukp/inception/recommendation/tasks/PredictionTask.java
322 ...dation/src/main/java/de/tudarmstadt/ukp/inception/recommendation/util/PredictionUtil.java
```

Figure 5.2: The analyzed commit message from "Inception_D"



Move Attribute private NO_SCORE : double
FROM class de.tudarmstadt.ukp.inception.recommendation.tasks.PredictionTask
TO class de.tudarmstadt.ukp.inception.recommendation.util.PredictionUtil

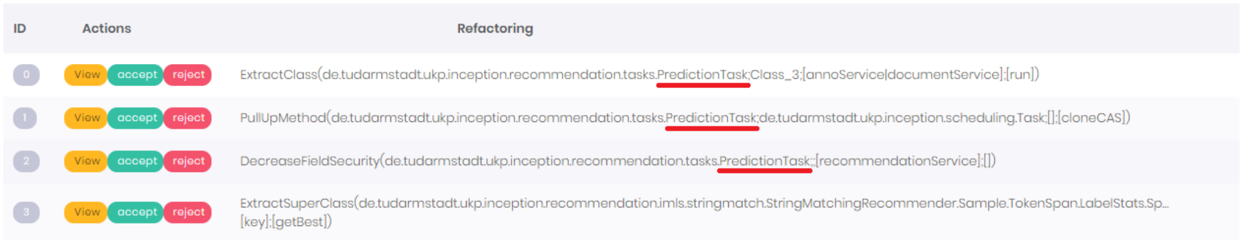
Move Method private **extractSuggestions**(aUser User, aCas CAS, predictionType Type, predictedFeature Feature, aScoreFeature Optional<Feature>, aDocument SourceDocument, aRecommender Recommender) : List<AnnotationSuggestion>
FROM class de.tudarmstadt.ukp.inception.recommendation.tasks.PredictionTask
TO class de.tudarmstadt.ukp.inception.recommendation.util.PredictionUtil

Move Method private **removePredictions**(aCas CAS, aPredictionType Type) : void
FROM class de.tudarmstadt.ukp.inception.recommendation.tasks.PredictionTask
TO class de.tudarmstadt.ukp.inception.recommendation.util.PredictionUtil

Move Method public **calculateVisibility**(aLearningRecordService LearningRecordService, aAnnotationService AnnotationSchemaService, aCas CAS, aUser String, aLayer AnnotationLayer, aRecommendations Collection<SuggestionGroup>, aWindowBegin int, aWindowEnd int) : void
FROM class de.tudarmstadt.ukp.inception.recommendation.tasks.PredictionTask
TO class de.tudarmstadt.ukp.inception.recommendation.util.PredictionUtil

Move Method private **hideSuggestionsRejectedOrWithoutLabel**(aSuggestion AnnotationSuggestion, aRecordedRecommendations List<LearningRecord>) : void
FROM class de.tudarmstadt.ukp.inception.recommendation.tasks.PredictionTask
TO class de.tudarmstadt.ukp.inception.recommendation.util.PredictionUtil

Figure 5.3: The manual refactoring applied by the developer in the commit



ID	Actions	Refactoring
0	View accept reject	ExtractClass(de.tudarmstadt.ukp.inception.recommendation.tasks.PredictionTask;Class_3;[annoService|documentService];[run])
1	View accept reject	PullUpMethod(de.tudarmstadt.ukp.inception.recommendation.tasks.PredictionTask;de.tudarmstadt.ukp.inception.scheduling.Task;[];[cloneCAS])
2	View accept reject	DecreaseFieldSecurity(de.tudarmstadt.ukp.inception.recommendation.tasks.PredictionTask;;[recommendationService];[])
3	View accept reject	ExtractSuperClass(de.tudarmstadt.ukp.inception.recommendation.imls.stringmatch.StringMatchingRecommender.Sample.TokenSpan.LabelStats.Sp... [key];[getBest])

Figure 5.4: The List of refactorings recommended by RefCom

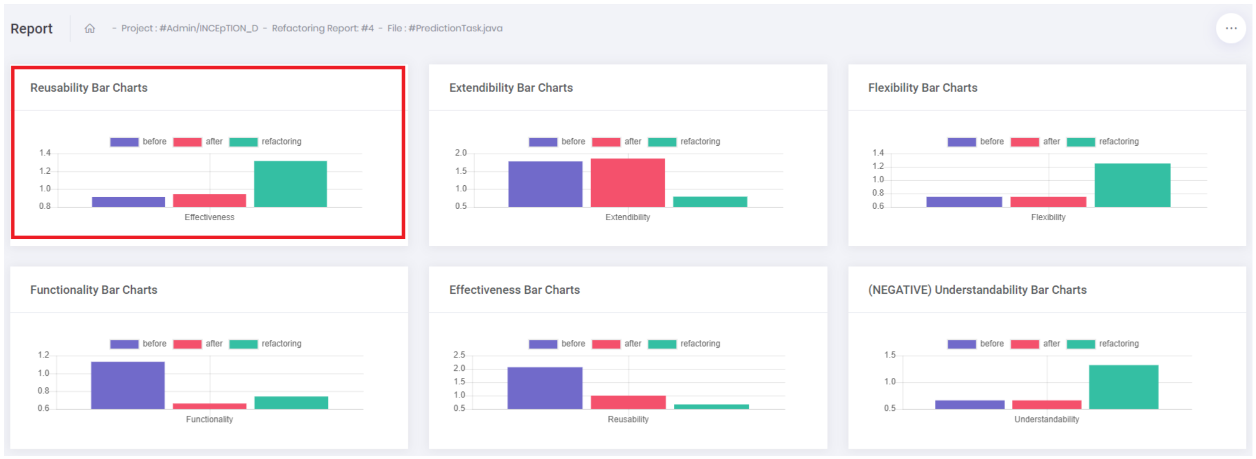


Figure 5.5: QMOOD quality before and after the commits comparing the manual refactorings and RefCom

While the *first* research question will validate our first hypothesis about developers document their refactoring rationale in commit messages, the *second* research question will validate the second hypothesis that developers spend the minimum of manual refactorings effort to fix the identified quality issues, thus there are inconsistencies (or incomplete refactorings) between documented and applied refactorings in terms of expected impact/intention. The third question will evaluate the relevance of the recommended refactorings after integrating the two above insights into our refactoring tool to make actionable recommendations. A demo of our refactoring tool, Refcom, can be found in [232].

5.4.2 Experimental Setting

To address the research questions, we analyzed the six open source systems in Table 5.2. Atomix is a fault-tolerant distributed coordination framework. Btm is a distributed and complete implementation of the JTA 1.1 API. Jgrapht is a graph library that provides mathematical graph-theory objects and algorithms. JSAT is a set of algorithms for pre-processing, classification, regression, and clustering with support for multi-threaded execution. Pac4j is a security engine. Tablesaw includes a data-frame, an embedded column store, and hundreds of methods to transform, summarize, or filter data. We selected these projects because of their size, number of commits, and applied refactorings.

Table 5.2: Summary of the evaluated systems.

N	Project Name	LOC	Number of Classes	Total Commits	Refactoring related commits	Total number of refactorings
1	atomix	182280	1459	4237	343	12909
2	btm	34232	187	975	150	522
3	jgrapht	158665	526	2902	204	2202
4	JSAT	182267	436	1561	236	1457
5	pac4j	31916	302	2282	127	3130
6	tablesaw	52837	224	1930	327	3143

To answer RQ1, we computed the ratio of the number of refactoring related commits to the total number of commits. Then, we counted the number of documented refactorings among these identified refactoring related commits. Documented refactorings are the commit messages that contain documentation about refactoring. These documented refactorings are detected using keywords. However, refactoring related commits are the commits found after the union of the results of RefactoringMiner [230] detection, keywords extraction (same list of keywords previously mentioned) and the observed quality attribute changes between commits detected using our dedicated parser. A commit can be considered as a "refactoring related commit" , while it does not contain refactoring documentation (in the commit message) because it may contain either refactorings detected by RefactoringMiner or included quality improvements (when comparing before/after refactoring). In addition to evaluate the number of refactoring related commits and documented refactorings, we have also evaluated the main quality attributes that are documented in refactoring related commits to understand the most important ones that developers document. The detection of the documented quality attributes is carried out by searching for quality attributes names and their roots in the commit messages. Finally, we investigated the number of commits that introduce significant changes in the quality attributes, but which developers did not document.

To answer RQ2, we checked all the quality attributes by analyzing the code, and not only the ones claimed/documented by developers in their commits. There are two main reasons for checking all the quality attributes improvement. First, it helped identifying the refactoring related commits that contain documented quality attributes but there were no actual observed improvement of the quality attributes before and after the commit. Second, checking all the quality attributes improvement helps detecting the commit that does not claim a quality attribute but still is related to refactoring. In fact, we have used Refactoring-Miner [230] and our tool for code analysis to detect the situations where quality attributes changes and applied refactorings were not documented. These are opportunities for refactoring solutions that better address these quality attributes.

To answer RQ3, we used the outcomes of the two prior research questions to identify developer refactoring rationale per commit: what files did they want to refactor? And what quality attributes did they want to improve? Then, we used that rationale to guide and filter the refactoring recommendations generated using our approach based on multi-objective search. We compared the automated refactorings using RefCom to the manual refactorings applied by the developers in the commits in terms of quality improvements. Then, we compared the recommended refactorings to two existing studies [125, 4] using a relevance measure. The relevance of the refactorings is defined as the number of refactoring recommendations accepted by developers participating in our experiments divided by the total number of recommended refactorings.

We asked 24 developers to evaluate the meaningfulness of the refactorings recommended by Refcom and by the approach of Ouni *et al.* [4] and JDeodorant [125] for pull-requests on the six subject systems. We followed a random order of the three tools when the results were manually inspected. All the experimental techniques generate sequences of refactoring operations that make sense when considered together rather than when looking at them in isolation. However, it is not an option to ask a developer to assess the meaningfulness of all the refactoring operations generated for a given system. For this reason, we started by filtering for each system the sequences of refactoring operations impacting the files of a set of pull-requests to make a fair comparison between both tools. Then, the developers manually evaluated the outcomes of both tools for the commits of each pull-request.

Each participant was then asked to assess the meaningfulness of the sequences of refactoring operations. We made sure that each participant only evaluated refactoring sequences recommended by the three competitive techniques on one system. The rationale for such a choice is that an external developer would need time to acquire system knowledge by inspecting its code, and we did not want participants to have to comprehend the code from multiple systems since this would introduce a training effect in our study.

To support such a complex experimental design, we built a Java Web-app that automatically assigns the refactored pull-requests to be evaluated to the developers. The Web-app showed each participant one sequence of refactoring operations on a single page, providing the developer with (i) the list of refactorings (*e.g.* move method m_i to class C_j, then push down field f_k to subclass C_j, *etc.*), (ii) the code of the classes impacted by the sequence of refactorings, and (iii) the complete code of the system subject of the refactoring with the generated refactoring sequence. The web page showing the refactoring sequence asked participants the question *Would you apply the proposed refactorings?* with a choice between *no* (*i.e.*, the refactoring sequence is not meaningful), or *yes* (*i.e.*, the refactoring sequence is meaningful and should be implemented). Moreover, participants were optionally allowed to leave a comment justifying their assessment. The Web-app was also in charge of:

Balancing the evaluations per system. We made sure that each system received roughly the same number of participants evaluating the different refactored pull-requests/commits (files associated/modified by these pull-requests) by the three approaches.

Keeping track of the time spent by participants in the evaluation of each refactoring sequence/pull-request. The time spent by participants was counted in seconds since the moment the Web-app showed the refactoring on the screen to the moment in which the participant submitted their assessment. This feature was done to remove participants from our data set who did not spend a reasonable amount of time in evaluating the refactorings. We consider less than 90 seconds a reasonable threshold to remove noise (*i.e.*, we removed all evaluation sessions in which the participant spent less than 90 seconds in analyzing a single refactoring sequence).

Collecting demographic information about the participants. We asked their programming experience (in years) overall and in Java, and a self-assessment of their refactoring experience (from very low to very high). All of the participants were hired based on our current and previous extensive industry collaborations on refactoring. Despite that we contacted open source developers, we did not receive from them a timely response or did not answer at all which is a common challenge and threat in human studies within software engineering research [233]. We made sure that all the selected participants from industry are experienced in refactoring and used before these open source systems/libraries.

Table 5.3: Participants involved to answer RQ3.

System	**#Partic.**	**Avg. Prog. Experience**	**Avg. Java Experience**	**Avg. Refact. Exp.(1-5)**
atomix	4	9	9	4.0 (high)
btm	4	8	7	3.5 (medium)
jgrapht	4	10	9	3.8 (medium)
JSAT	4	9	7	3.5 (high)
pac4j	4	7.5	7	4.5 (very high)
tablesaw	4	9	9	3.5 (high)

Table 5.3 shows the participants involved in our study and how they were distributed in the evaluation of the refactoring sequences generated for the six systems.

5.4.3 Results

Results for RQ1. Since our work is based on the assumption that developers write commit messages to document some of the applied refactorings, we identified first the commits related to refactorings then we checked those that documented the applied refactorings in the commit messages.

Table 5.4 summarizes our findings. It is clear that all the six open source projects have extensive refactorings applied in previous commits: an average of over 30% of all commits. The Atomix system has the highest number of commits related to refactoring. We found that 211 commit messages documented the applied refactorings, which is more than 60% of commits containing refactorings. The same observation can be applied to the remaining systems. While developers extensively apply refactorings, they may not document all of them. Still there are enough commits including refactoring documentation to identify further opportunities for refactoring.

Table 5.4: An overview of the documented commits related to refactoring on the six open source systems.

Project	Total number of commits	Commits related to refactoring	Docuented commits related to refactoring	Commits identified with RefacotoringMiner	Commits identified with Quality Improvements
atomix	4237	343	211	233	174
btm	975	150	52	55	46
jgraphft	2902	204	107	87	40
jsat	1561	236	113	58	65
pac4j	2282	127	84	65	33
tablesaw	1930	327	159	116	63

We also investigated the main quality attributes of QMOOD that were documented by developers in the commit messages when refactorings were applied to improve those attributes. As described in Figure 5.6, we found understandability to be the most common quality documented by developers in commit messages. In 4 of the 6 open source systems it is the most common quality attribute documented by developers. For instance, the developers mentioned the rationale of understandability in messages in 53% of the commits improving the Atomix system. Reusability is the second most documented rationale, on average, in the six systems. It is also normal that developers document the rationale of the refactorings in combination with the features that were modified (functionality).

To conclude, we found that developers do document refactorings and they extensively apply refactorings over the commits of all six open source systems. Our results show that developers mention quality attributes as a rationale for their refactorings in over 50% of commits related to refactoring that are documented, which is enough to find opportunities for enhanced refactorings.

Results for RQ2. Figure 5.7 shows that developers are documenting their intention to refactor the code to address quality issues in the commit messages; however we did not find any quality improvements when we analyzed the quality changes in the files of these commits. For the Btm system, we found that only 32 out of 149 commits related to refactoring have actual quality changes. Only 60 out 236 commits related to refactorings have actual quality changes despite developers commenting on applying refactorings in their commit messages.

It is clear that developers highlight their intention to refactor the code with its ratio-

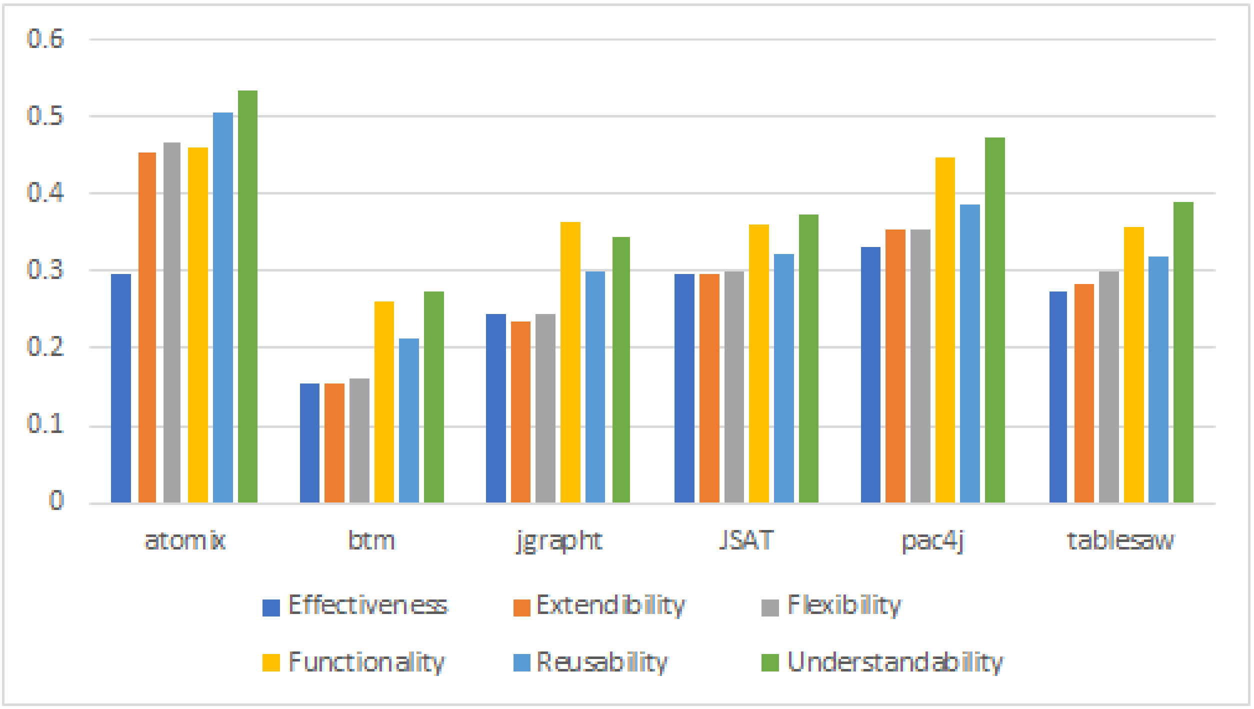


Figure 5.6: The percentage of documented quality attributes per system among the commits improving the quality attributes.

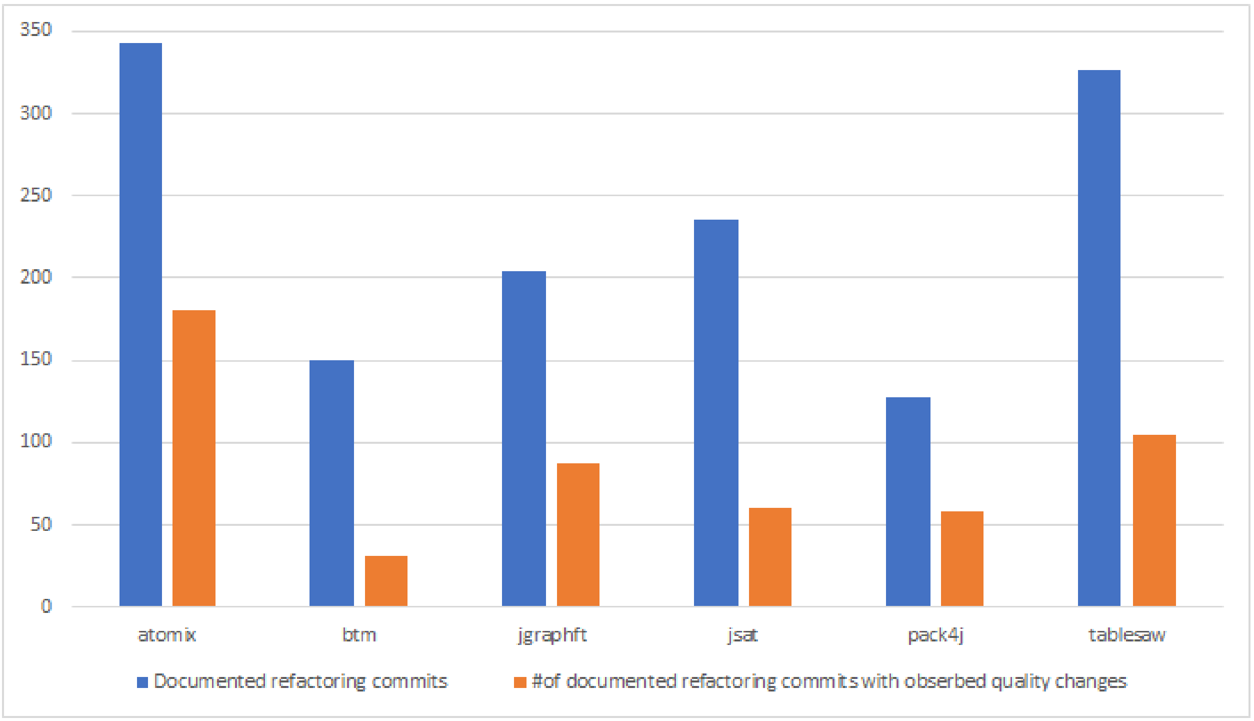


Figure 5.7: Missed documented refactoring opportunities in the 6 systems.

nale; however no actual quality improvements have been observed in many commits. This conclusion is one of the main motivations for RQ3.

Results for RQ3. After validating the two hypotheses of the previous research questions, we implemented our Refcom tool for improving the QMOOD quality attributes by integrating a filter to guide the refactoring recommendations based on rationale identified in the previous research questions (what quality attributes and which files do developers want to improve?). Figure 5.6 shows that developers documented refactorings with the intention of improving all the 6 quality attributes but with different levels of frequency. For instance, it is clear that developers focused on improving both understandability and reusability in project atomix. Thus, we executed our multi-objective algorithm using all the 6 quality attributes then we filter the Pareto front based on the two main criteria that are contained in the extracted refactoring rationale. First, we make sure that the selected solution is the one that provides the highest improvement in the quality attributes extracted from the commits during our analysis step (e.g. understandability and reusability in project atomix). Second, the optimal solution should also refactor the detected changed files in the commits. We compared our results with two existing refactoring tools. Ouni *et al.* [4] proposed a multi-objective refactoring formulation based on NSGA-II that generates a solution to maximize the design coherence and refactoring reuse from previous releases. JDeodorant [125] is an Eclipse plugin to detect bad smells and apply refactorings.

Figure 5.8 highlights the out-performance of RefCom compared to the tools of Ouni et al. [4] and JDeodorant [125]. In fact, most refactorings recommended by our approach are relevant, and all of them were successfully applied for the case Atomix system on the expected files and achieved high-quality improvements, based on the feedback from the participants.

By looking at the comments left by participants when justifying their assessments, thirteen out of the twenty four developers highlighted in their comments about the refactoring sequences that they found the refactorings relevant because they are completing the effort started by the submitter of the developer as described in the commit messages. For example,

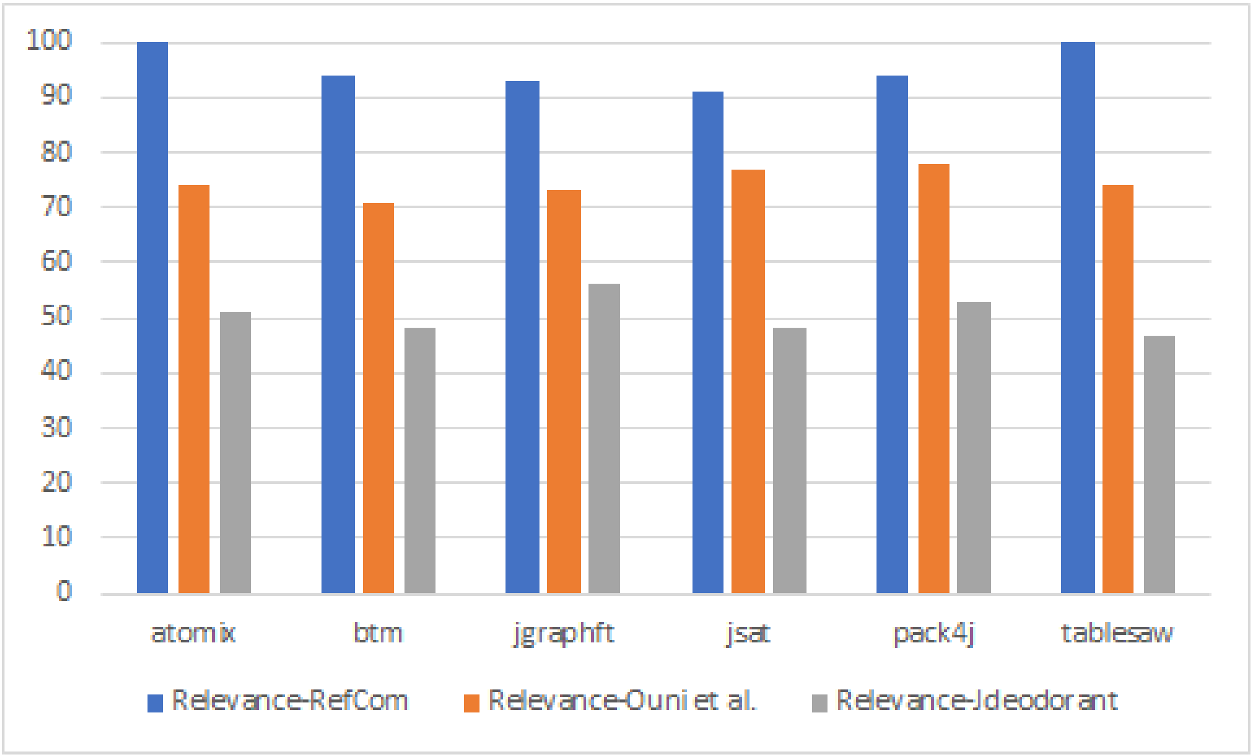


Figure 5.8: The relevance of the recommended refactorings by RefCom compared to existing refactoring approaches.

one of the developers wrote in a comment: *"I found these refactorings really improving the reusability of this class which is the main intention of the developer but he just applied couple of move methods. I found the tool recommendation even better to improve the reusability."*. We found this comment as important qualitative evidence of only the value of RefCom in terms of analyzing the recently closed pull-requests to identify changed files and fix the identified quality issues in these files.

Thus RefCom provided relevant refactoring recommendations based on the commit analysis, outperforming existing approaches to recommend refactorings.

5.5 Threats to Validity

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

The threats to internal validity can be related to the list of keywords that we used to identify the commits where developers documented refactorings. However, the impact of this threat was limited by considering the use of RefactoringMiner to identify the actual refactorings applied by developers. The parameters tuning of the optimization algorithm used in our experiments may create an internal threat that needs to be evaluated in future work since the parameter values used in our experiments were found by trial and error.

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 [154], it is possible that some quality changes may not be detected using QMOOD.

External validity refers to the generalizability of our findings. We performed our experiments on 6 open-source systems belonging to different domains. However, we cannot assert that our results can be generalized to other applications and other developers. Moreover, we found that only 32 out of 149 commits related to refactoring have actual quality changes which limits the generalizability of our findings and requires more experiments. Another threat could be the number of subjects (24 developers) used for validation. Future replications of this study are necessary to confirm our findings.

5.6 Conclusion

We presented a first attempt to recommend refactorings by analyzing commit messages. The salient feature of the proposed **RefCom** approach is its ability to capture developers need, from their commit messages, and propose to them refactorings to enhance their changes to better address quality issues. To evaluate the effectiveness of our technique, we applied it to six open-source projects and compared it with state-of-the-art approaches that rely on static and dynamic analysis. Our results show promising evidence on the usefulness of the proposed commit-based refactoring approach.

Future work will involve validating our technique with additional refactoring types, programming languages and a more extensive set of projects and commits to investigate the general applicability of the proposed methodology. We will also check the relevance of integrating commit messages in finding and recommending refactoring opportunities then fixing them based on different refactoring recommendations tools beyond our previous work.

CHAPTER VI

Refactoring Documentation Bot

6.1 Introduction

Documentation is a recommended practice in software development and maintenance to help developers understand the code quickly and improve their productivity[43]. According to a study [44], 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 [45, 46, 47].

Refactoring [234] is used to improve the quality of code while preserving its behavior. Tom Mens et al. [50] defined the different steps of refactoring including the detection, prioritization, recommendation, testing and documentation. While existing refactoring studies extensively addressed the first four steps [235, 236, 217], 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 [237, 238, 239, 240]. 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 [235]. 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 [219]. 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. [43] shows that documentation needs to be prioritized for refactoring.

In this chapter, 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 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 [241].

The primary contributions of this research work can be summarized as follows:

1. The presented work 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 pressented work 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.

6.2 Problem Statement & 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 [43]. 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 [235]. 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.

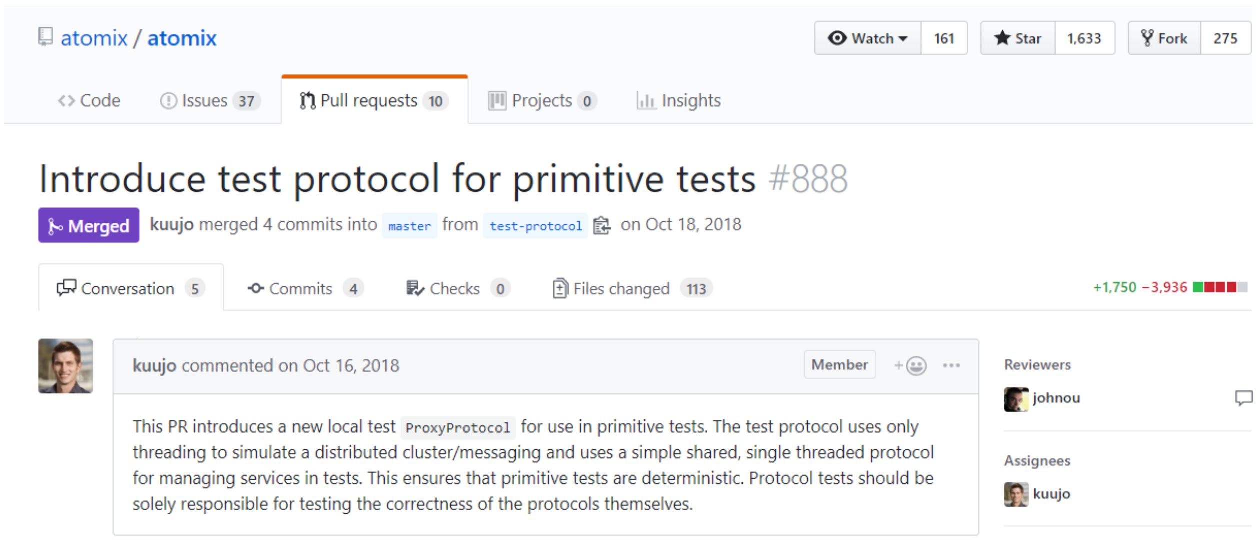
We used the 59 software engineering antipattern terms defined in Wikipedia [1]. Then, we searched these antipattern terms in all the commit messages from 1,984 popular projects (including C and Java) in GitHub. Only 0.13% of the 8.4 million commit messages mention any antipattern term. This shows that 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 6.2 shows that in the pull request captured in Figure 6.1, 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"

[1]https://en.wikipedia.org/wiki/Anti-pattern#Software_engineering



Introduce test protocol for primitive tests #888

Merged kuujo merged 4 commits into `master` from `test-protocol` on Oct 18, 2018

kuujo commented on Oct 16, 2018 — Member

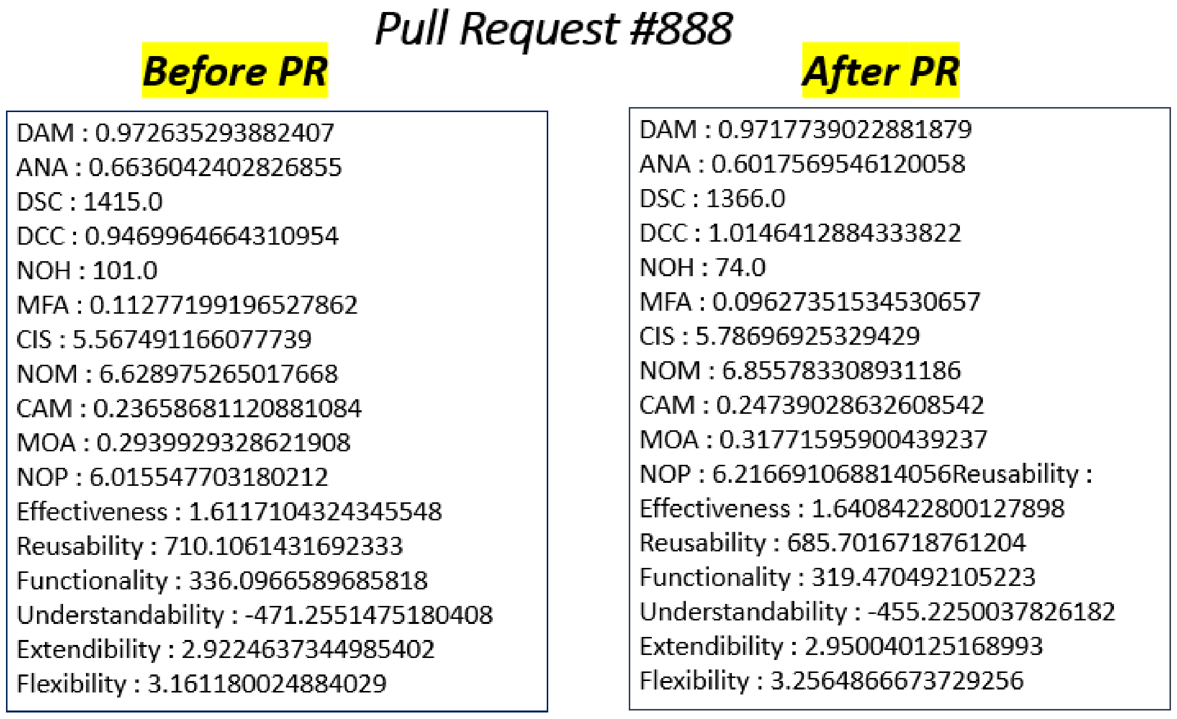
This PR introduces a new local test `ProxyProtocol` for use in primitive tests. The test protocol uses only threading to simulate a distributed cluster/messaging and uses a simple shared, single threaded protocol for managing services in tests. This ensures that primitive tests are deterministic. Protocol tests should be solely responsible for testing the correctness of the protocols themselves.

Figure 6.1: Pull Request with Poor Documentation

improvement are often neglected by developers in their documentation as described in Figure 6.3 that shows the significant quality improvements before and after the pull request.

6.3 Approach: Refactoring Documentation Bot

Figure 6.4 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 [78], ChangeScribe [76, 77], and others [102, 103], 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



Pull Request #888

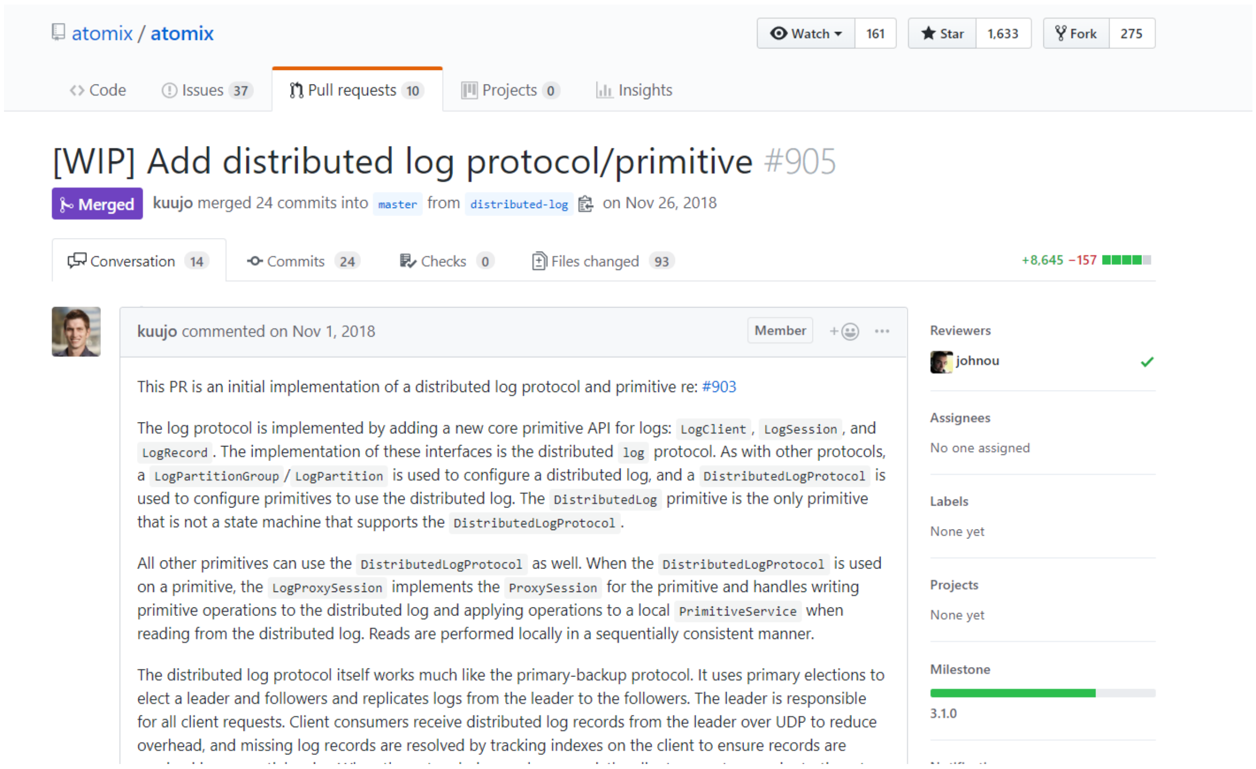
Before PR

DAM : 0.972635293882407
ANA : 0.6636042402826855
DSC : 1415.0
DCC : 0.9469964664310954
NOH : 101.0
MFA : 0.11277199196527862
CIS : 5.567491166077739
NOM : 6.628975265017668
CAM : 0.23658681120881084
MOA : 0.2939929328621908
NOP : 6.015547703180212
Effectiveness : 1.6117104324345548
Reusability : 710.1061431692333
Functionality : 336.0966589685818
Understandability : -471.2551475180408
Extendibility : 2.9224637344985402
Flexibility : 3.161180024884029

After PR

DAM : 0.9717739022881879
ANA : 0.6017569546120058
DSC : 1366.0
DCC : 1.0146412884333822
NOH : 74.0
MFA : 0.09627351534530657
CIS : 5.78696925329429
NOM : 6.855783308931186
CAM : 0.24739028632608542
MOA : 0.31771595900439237
NOP : 6.216691068814056Reusability :
Effectiveness : 1.6408422800127898
Reusability : 685.7016718761204
Functionality : 319.470492105223
Understandability : -455.2250037826182
Extendibility : 2.950040125168993
Flexibility : 3.2564866673729256

Figure 6.2: The quality metrics change in the pull request.



[WIP] Add distributed log protocol/primitive #905

kuujo merged 24 commits into `master` from `distributed-log` on Nov 26, 2018

kuujo commented on Nov 1, 2018

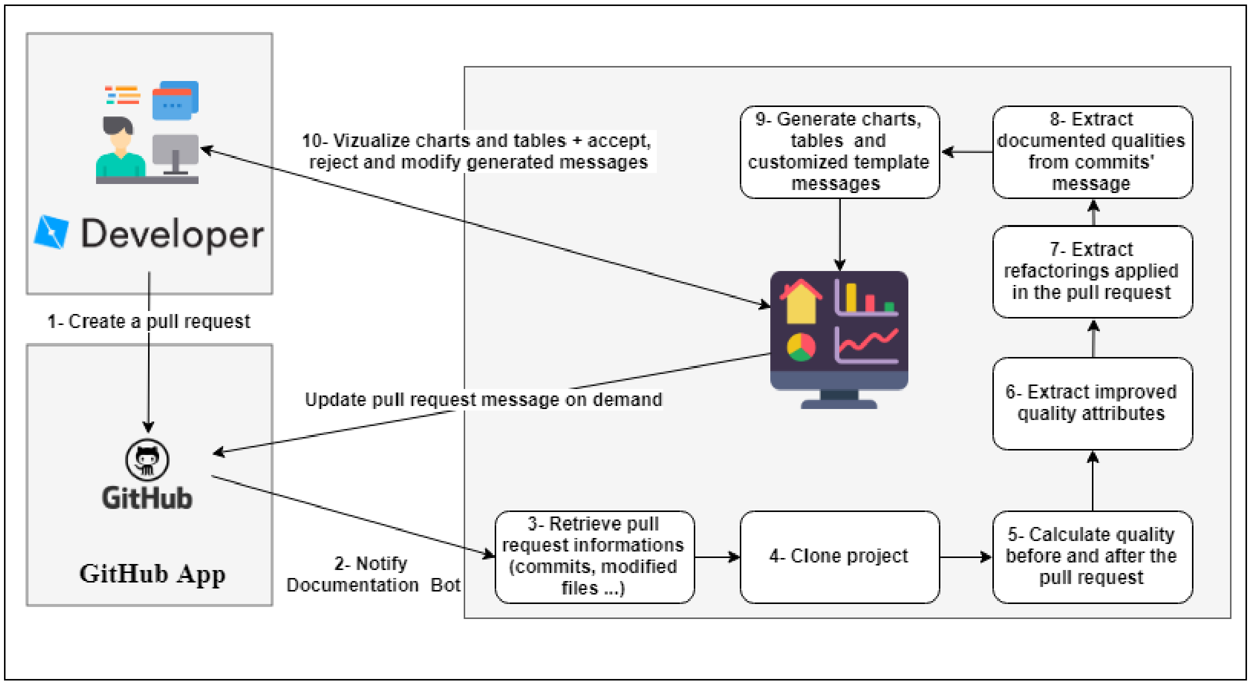
This PR is an initial implementation of a distributed log protocol and primitive re: #903

The log protocol is implemented by adding a new core primitive API for logs: `LogClient`, `LogSession`, and `LogRecord`. The implementation of these interfaces is the distributed `log` protocol. As with other protocols, a `LogPartitionGroup` / `LogPartition` is used to configure a distributed log, and a `DistributedLogProtocol` is used to configure primitives to use the distributed log. The `DistributedLog` primitive is the only primitive that is not a state machine that supports the `DistributedLogProtocol`.

All other primitives can use the `DistributedLogProtocol` as well. When the `DistributedLogProtocol` is used on a primitive, the `LogProxySession` implements the `ProxySession` for the primitive and handles writing primitive operations to the distributed log and applying operations to a local `PrimitiveService` when reading from the distributed log. Reads are performed locally in a sequentially consistent manner.

The distributed log protocol itself works much like the primary-backup protocol. It uses primary elections to elect a leader and followers and replicates logs from the leader to the followers. The leader is responsible for all client requests. Client consumers receive distributed log records from the leader over UDP to reduce overhead, and missing log records are resolved by tracking indexes on the client to ensure records are

Figure 6.3: PR with only functional changes are documented



Developer
1- Create a pull request
GitHub
GitHub App
10- Vizualize charts and tables + accept, reject and modify generated messages
Update pull request message on demand
2- Notify Documentation Bot
9- Generate charts, tables and customized template messages
8- Extract documented qualities from commits' message
7- Extract refactorings applied in the pull request
6- Extract improved quality attributes
3- Retrieve pull request informations (commits, modified files ...)
4- Clone project
5- Calculate quality before and after the pull request

Figure 6.4: Approach Overview: Refactoring Documentation Bot

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.

The documentation-Bot is a Spring Boot application that is implemented as a GitHub App [242]. 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 6.4.

6.3.1 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 [235] 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 [235].

Table 6.1: 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

6.3.2 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, 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 6.1. These keywords have been already defined in the literature based on different surveys including Microsoft developers [243].

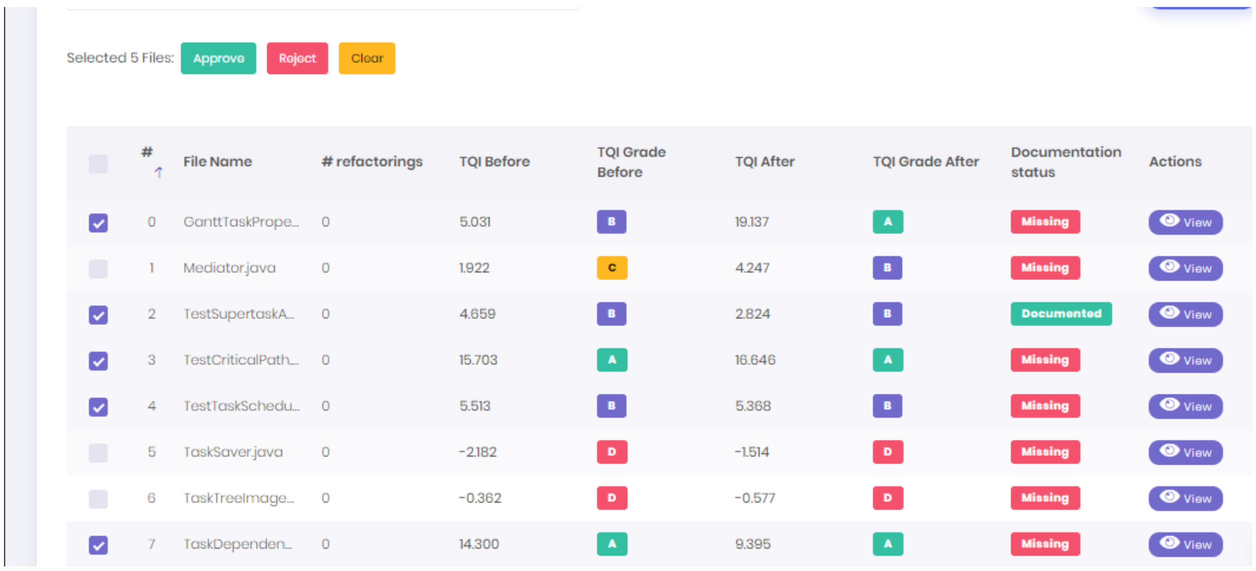
The combination of keywords and the detected refactorings along with the name of the modified files represent a sufficient set of features that help us checking whether the specific quality attributes and refactorings detected in the previous step are documented in any of the commit messages and the 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 [219]. 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.

6.3.3 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 6.10. 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).

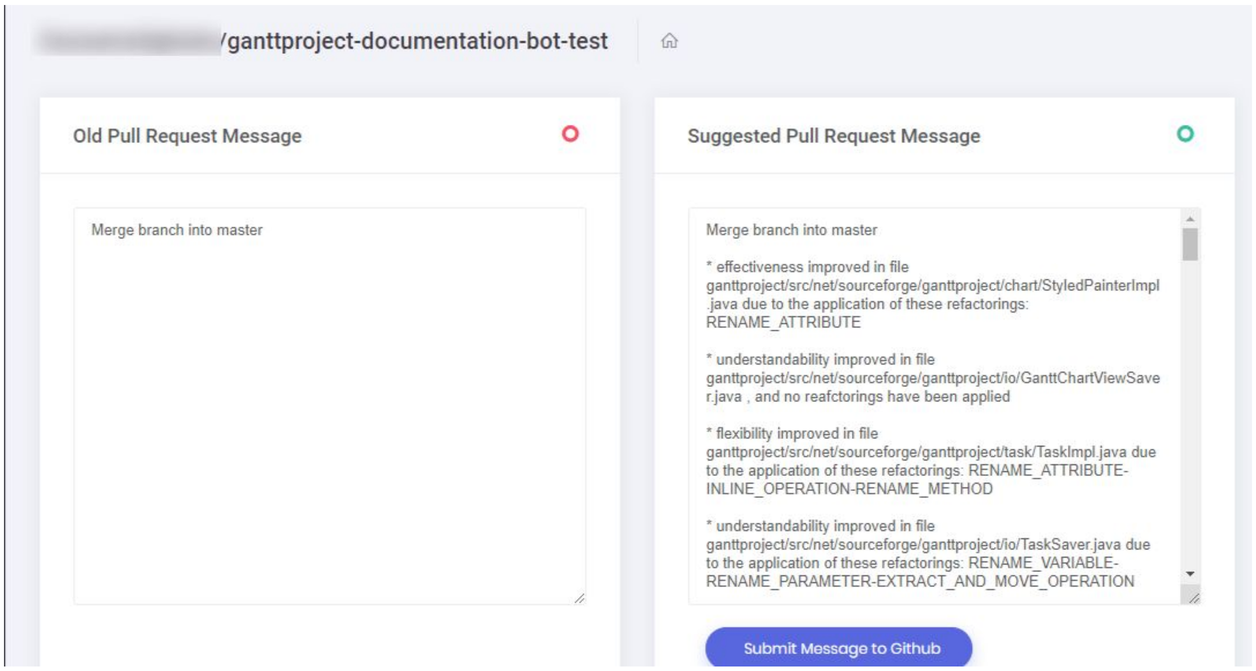
The generated refactoring documentation will follow a specific set of rules template as described in Figure 6.6: Our message will be composed of the location (file name), the refactoring applied and the quality attributes that have significantly changed and the developers missed them in their documentation. 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 6.7.



	#	File Name	# refactorings	TQI Before	TQI Grade Before	TQI After	TQI Grade After	Documentation status	Actions
✓	0	GanttTaskPrope...	0	5.031	B	19.137	A	Missing	View
	1	Mediator.java	0	1.922	C	4.247	B	Missing	View
✓	2	TestSupertaskA...	0	4.659	B	2.824	B	Documented	View
✓	3	TestCriticalPath...	0	15.703	A	16.646	A	Missing	View
✓	4	TestTaskSchedu...	0	5.513	B	5.368	B	Missing	View
	5	TaskSaver.java	0	-2.182	D	-1.514	D	Missing	View
	6	TaskTreeImage...	0	-0.362	D	-0.577	D	Missing	View
✓	7	TaskDependen...	0	14.300	A	9.395	A	Missing	View

Figure 6.5: Developer's interaction with the Refactoring Documentation Bot


[redacted]/ganttproject-documentation-bot-test

Old Pull Request Message

Merge branch into master

Suggested Pull Request Message

Merge branch into master

* effectiveness improved in file ganttproject/src/net/sourceforge/ganttproject/chart/StyledPainterImpl.java due to the application of these refactorings: RENAME_ATTRIBUTE

* understandability improved in file ganttproject/src/net/sourceforge/ganttproject/io/GanttChartViewSaver.java , and no reafctorings have been applied

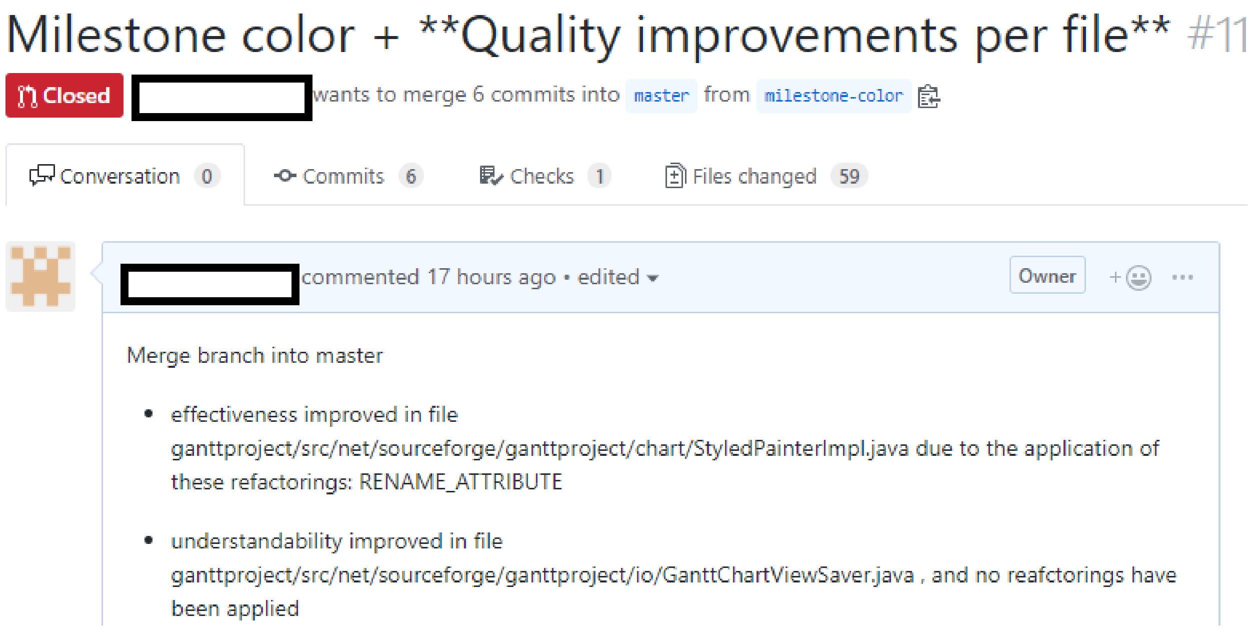
* flexibility improved in file ganttproject/src/net/sourceforge/ganttproject/task/TaskImpl.java due to the application of these refactorings: RENAME_ATTRIBUTE-INLINE_OPERATION-RENAME_METHOD

* understandability improved in file ganttproject/src/net/sourceforge/ganttproject/io/TaskSaver.java due to the application of these refactorings: RENAME_VARIABLE-RENAME_PARAMETER-EXTRACT_AND_MOVE_OPERATION

Figure 6.6: Developer's Pull Request Description vs. our Bot's Description

6.4 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 source systems. A demo of the refactoring documentation bot can be found in [241]. In this section,


Milestone color + **Quality improvements per file** #11

Closed [redacted] wants to merge 6 commits into master from milestone-color

Conversation 0 Commits 6 Checks 1 Files changed 59

[redacted] commented 17 hours ago • edited Owner

Merge branch into master

- effectiveness improved in file ganttproject/src/net/sourceforge/ganttproject/chart/StyledPainterImpl.java due to the application of these refactorings: RENAME_ATTRIBUTE
- understandability improved in file ganttproject/src/net/sourceforge/ganttproject/io/GanttChartViewSaver.java , and no reafctorings have been applied

Figure 6.7: A generated pull request description submitted on GitHub by our bot

we first present our research questions and validation methodology followed by experimental setup. Then we describe and discuss the obtained results.

6.4.1 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)?

6.4.2 Experimental Setting and Studied Projects

To address the different research questions, we used the 5 open source systems in Table 6.2. We selected these projects because of their size, number of commits, applied refactorings, 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 6.2.

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.

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.

Table 6.2: 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

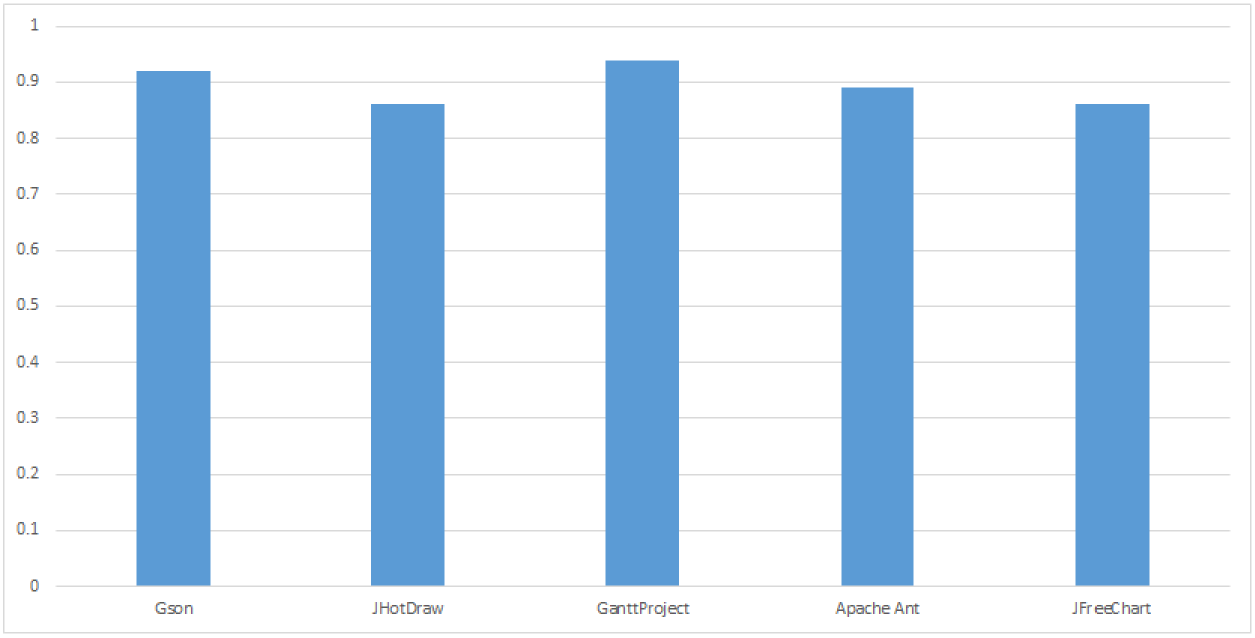


Figure 6.8: The average manual correctness score on the different 5 systems as evaluated by the participants.

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.

6.4.3 Results

Results for RQ1. Figure 6.8 summarizes our findings regarding the correctness of the generated pull-request and commit messages on the 5 systems. We found that a considerable

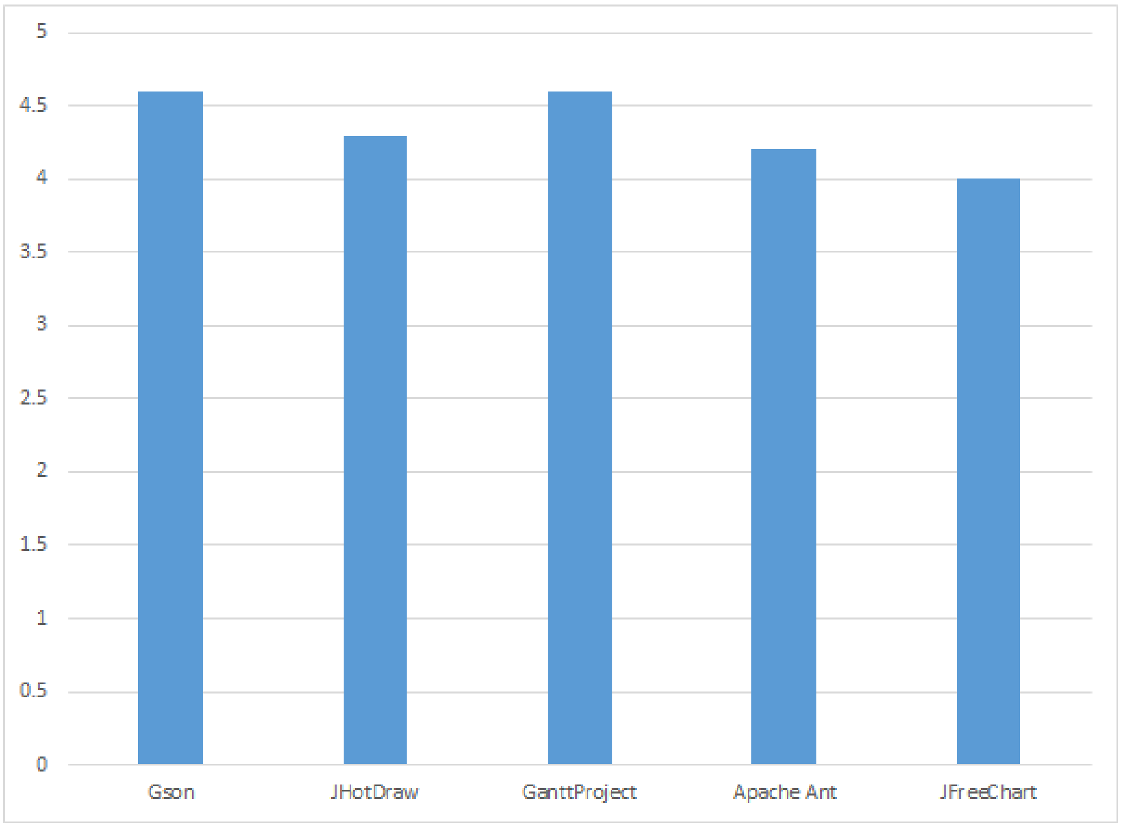


Figure 6.9: The average manual relevance score on the different five systems

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 6.9. 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 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 6.10 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 document 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:

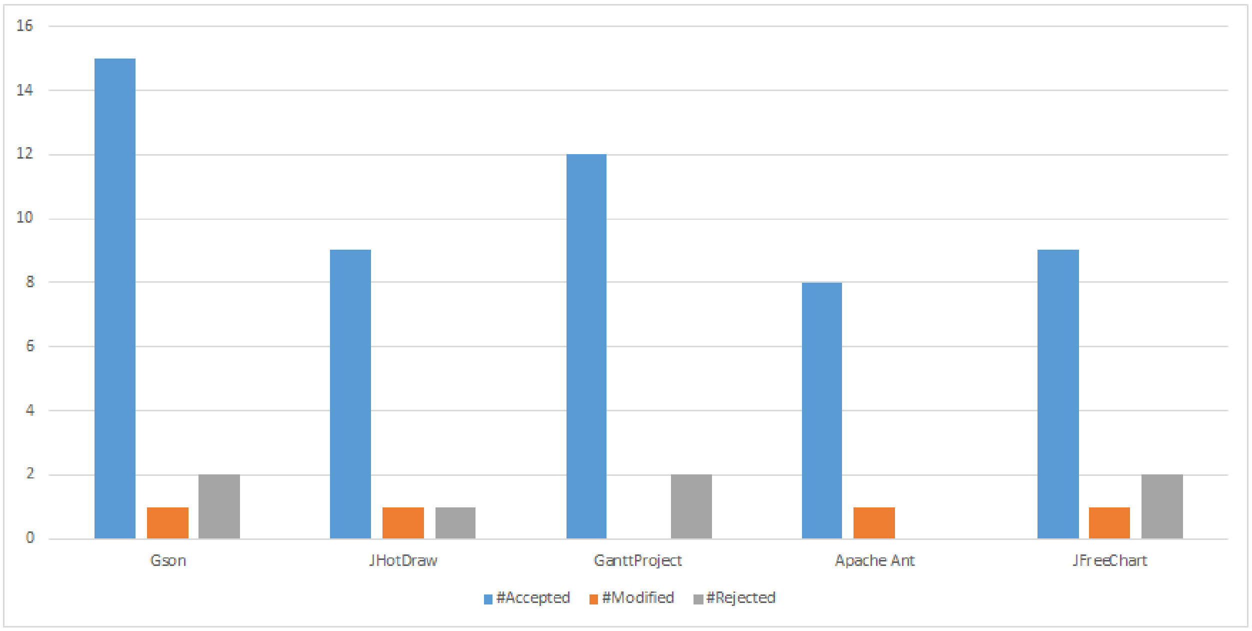


Figure 6.10: The average number of AR (percentage of accepted messages), NMR (percentage of modified messages) and NRR (percentage of rejected messages) on the different five systems.

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

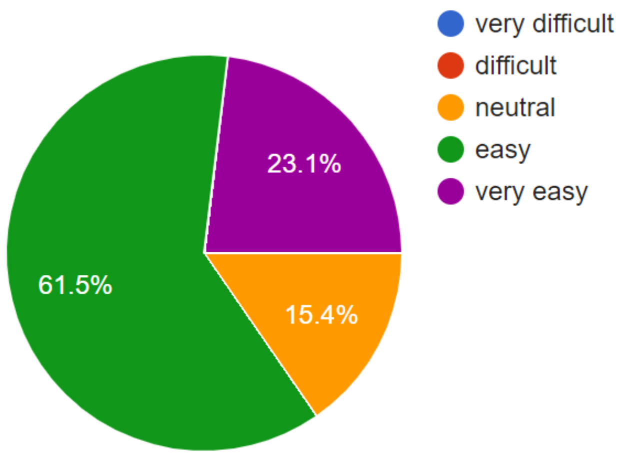


Figure 6.11: Distribution of the opinions of the participants about the usability of our refactoring documentation bot

umented. Another suggested improvement is to expand the tool to generate documentation for both functional and non-functional changes.

Figure 6.11 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 6.12. 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.

6.5 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

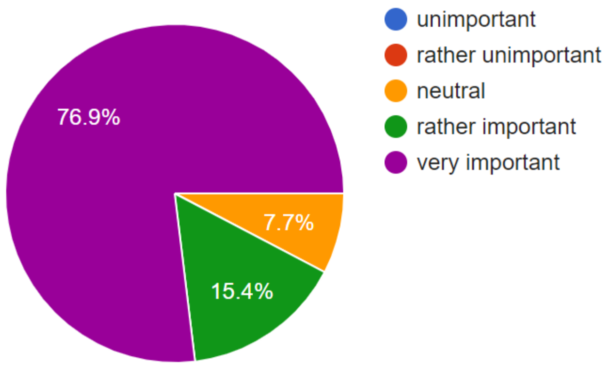


Figure 6.12: Distribution of the opinions of the participants about the importance of our refactoring documentation bot

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 [154], 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.

6.6 Conclusion

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.

CHAPTER VII

4W+H Model for Refactoring Documentation: A Practitioners' Perspective

7.1 Introduction

Software refactoring, defined as code restructuring while preserving the behavior [48], is guided by decisions from developers for various reasons such as improving quality and preventing bugs [49, 48, 50, 51, 52]. Several empirical studies [53, 54, 55, 56] show that refactoring is complex and time-consuming, and usually involves a sequence of dependent code actions to address challenging quality issues [52]. The effective understanding and documentation of refactorings can play a critical role in reducing and monitoring the *technical debt* [57, 58, 59, 60] by different stakeholders including executives, managers, and developers. In particular, refactoring documentation can help developers, managers, and executives keep track of applied refactorings, their rationale, and their impact on the system. Refactorings can be well-understood without documentation when they are carried out in isolation. However, refactoring always happens as a group of dependent refactoring operations and can even be mixed with functional changes. For instance, fixing major issues such as architecture hotspots may require a long sequence of refactorings applied in one commit and it is hard to capture that rationale without explicit documentation. Unlike existing techniques for generating commit messages to atomic changes based on deep-learning or textual similarities, we advocate for an empirical foundation that can be used to evaluate the quality of refactoring documentation and template/standards to generate it.

The commit messages and the pull-requests descriptions are becoming the most common ways to document code changes, including refactoring, in modern collaborative coding platforms, e.g., GitHub [244]. Recent research studies [67, 245, 75] have advocated for the development of automated recommendation systems to generate commit and pull-request messages. Thus, several automated techniques for the generation and recommendation of documentation of diffs and atomic changes (e.g., [66, 74, 75, 67, 76, 77, 78]) have been recently proposed. However, most of the current development workflows/pipelines in industry are lacking tools/steps to document refactorings and quality changes/technical debt. To the best of our knowledge, there are no standards to document refactorings or any prior empirical studies about understanding refactoring documentation. The current set of commonly used tools offer to see and document diff-changes but not dependent atomic changes/diffs.

Only two studies are related to refactoring documentation to (1) check if developers document refactorings in commit messages using key-words [100], and (2) generate refactoring documentation based on a pre-defined template to describe the changes [75]. None of these existing studies proposed a solid empirical foundation on what information is (or is not) useful to developers when documenting refactorings based on surveys with practitioners.

We advocate that a critical and fundamental step in providing an efficient support for developers in documenting refactoring is to discover the specific pieces of information, called components, that are necessary to include in commit messages to describe introduced refactorings. Thus, the main goal of this research work is to discover these components in which developers decompose the refactoring documentation. We formed our intuition that it could be decomposed into multiple components, each one addressing different aspects of the refactoring, by taking inspiration from recent empirical work with developers on understanding the rationale of code changes [98] and documenting functional changes [246].

We used three complementary methods in our study. As a first step, we discovered how developers decompose refactoring documentation by interviewing 14 software developers to establish a model for refactoring documentation. In a second step, we performed a survey to ask an additional 75 developers about their experiences with requiring, finding, and documenting the different components of our refactoring documentation model. During these two steps, we answered the following three research questions (RQ):

RQ1. *What are the refactoring documentation components in commit messages?*

RQ2. *How important is refactoring documentation in general, and each of its components in particular?*

RQ3. *What is the developer's experience with finding and documenting each of these refactoring's components?*

The third step of commits analysis aims to compare the explicit and implicit preferences of developers when documenting refactorings, and also evaluate different variations of the identified documentation model. We asked each of the participants during the interviews to evaluate the documentation quality of around 100 refactorings-related commit messages from our data-set using the Likert Scale before starting the discussions or showing our initial components. Thus, all the 1193 commit-messages were evaluated by the interviewees in terms of quality with an overlap of 207 random-commit-messages. Each of these 207 commit message was evaluated by two-participants to calculate the Cohen's-Kappa statistics for the agreement between the participants. We have also conducted a manual inspection of 1,193 commit messages extracted from open-source projects to compare the final refactoring documentation model, obtained after the interviews and the survey, with actual refactoring documentation extracted from open-source projects. To further study the correlation of the five components of our final model 4W+H and its variations with the quality of refactoring commit messages, used two statistical tests of Goodman and Kruskal, and Chi-Square. Thus, we answered the following research question based on these two above scenarios:

RQ4. *Would comparing the experience of developers in requiring, finding, and evaluating the refactorings documentation with samples of actual refactoring commit*

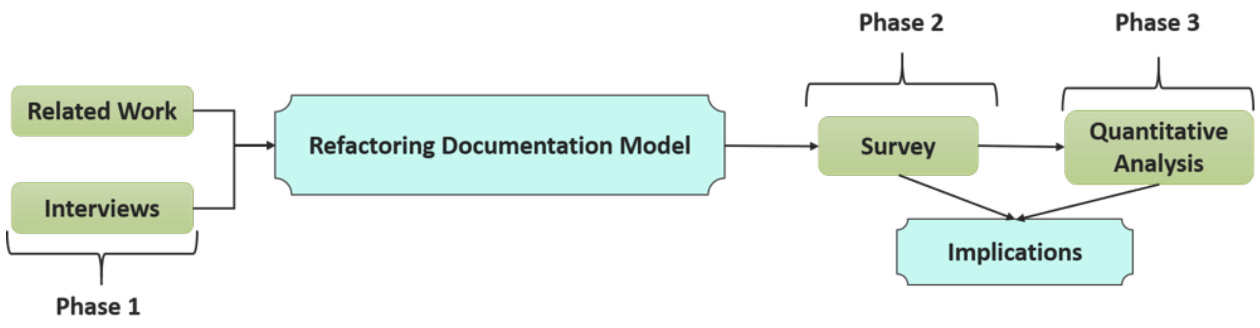


Figure 7.1: Research Design

messages reveal some inconsistencies?

Our findings can enable (1) researchers to automatically improve, assess, and generate refactoring documentation, (2) educators to teach and emphasize the different important components of refactoring documentation, and (3) practitioners to use a standard format in documenting and discussing refactorings. Though we identified a set of essential components of refactoring documentation, adoption of the components remains context-dependent in practice. Using our model, software development teams can design their organization-specific guidelines to include or exclude the proposed components for refactoring documentation.

Replication Package. All material and data used in our study as well as the developers' anonymized answers are available in our replication package [247].

7.2 Study Design

As described in Figure 7.1, we first used face-to-face unstructured interviews with 14 developers to discover the components of refactoring documentation and thereby, design an initial model. Before starting the discussions or showing (after the discussion) our initial components, the interviewees have also conducted an analysis of 1,193 refactoring commit messages to evaluate their quality. The purpose of this step was to stimulate their memories/experiences with refactorings-related-commits using these diverse-examples and use the data to validate our documentation model later. These interviews allowed rich conversations and insights. Then, we extended the obtained initial documentation model with a larger number of 75 practitioners using a survey. Furthermore, the practitioners answered our questions about their experiences in finding and documenting these different refactoring components. The use of these mixed methods has been widely employed by several other studies of software developers [248, 249, 250, 251].

7.2.1 Research Questions

RQ1: *What are the components of the refactoring documentation in commit messages and pull-request descriptions?* First, we asked practitioners about the possible components of refactoring documentation that correspond to the information pieces to form a high-quality description of refactorings. Our aim is to find a set of components for refactoring documentation to help developers in improving the quality of their refactoring documentation in code commits.

RQ2: *How important is refactoring documentation in general, and each of its components in particular?* Since it may not be possible to force developers to document all the components, we investigated the level of importance/need for each component for practitioners to ensure a high quality of refactoring documentation. The outcome of this research question can help in building automated tools to check the quality of refactoring documentation and to warn practitioners about missing details in their commit messages that include refactorings.

RQ3: *What is the developers' experience with finding and documenting refactoring components?* The aim of this research question is to understand the effort spent by practitioners to find and document each of the identified components of the refactoring documentation model. Identifying these components that are hard to find and document can emphasize the need for tools to generate them.

RQ4: *Would comparing the experience of developers in requiring, finding, and evaluating the refactorings documentation with samples of actual refactoring commit messages reveal some inconsistencies?* We compared in this research question the outcomes of both the interviews and the survey with actual documentation of refactorings extracted from open source systems as well as industrial projects. We have also conducted correlation analysis between the discovered components and the quality of commit messages. Observations from the comparison between the practitioners' need and actual refactoring documentation found through commit messages analysis could lead us to the areas of improvement for researchers and practitioners to address them and also validate the proposed refactoring documentation model.

7.2.2 Phase 1: Interviews with Developers to Design a Refactoring Documentation Model

Interviews Setup. The goal of the first phase of our research is to build a model for refactoring documentation. Toward this goal, we scheduled pilot sessions to arrange our individual interviews with developers. These unstructured interviews consisted of three main parts. In the first part, we asked the participants to evaluate the quality of refactoring commit messages based on their experience. Then, we collected possible components that can form a complete and high-quality refactoring documentation. Finally, the third part consisted of investigating these practitioners' experience in finding and documenting the identified components along with their importance.

Before starting the individual interview sessions, we carried out an in-depth analysis of previous studies that are related to code changes documentation in general [245, 244, 47, 243, 93, 49, 98]. We identified the most common and frequent components in software documentation in general to distill the existing knowledge for refactorings, since they are special cases of code changes. Our main observation is that most of the previous studies were considering *What are the applied code changes?* and *Why these code changes were made?* as important and common components to be documented when describing general code changes. Thus, we considered these two components as our first initial refactoring documentation model to be reviewed and extended by practitioners.

We started our individual interviews by providing 100 examples of commit messages documenting refactorings extracted from open-source projects. We asked each participant to evaluate the quality of 100 different refactorings-related commit message from our dataset using the Likert Scale (1:very poor, 5:very good). Then, we asked them to tell us some real-world situations when they needed to carefully check a sequence of code refactorings to understand it. These steps helped stimulate the practitioners' memories as well. As a next step, we asked them to think about possible components to document refactorings after describing their experiences in requiring, finding, and documenting refactorings that they applied in the past. Then, the participant created their own model. After that, we showed them an initial model of the two components of the refactoring documentation that we established from existing studies on documentation of code changes. We discussed with the participants the differences and they showed us at least one example of any new component that they suggested.

To prevent any potential bias in the presented initial refactoring documentation model, we ensure that it was based on the existing research literature on refactoring documentation rather than reflecting our opinions. Furthermore, we showed the initial model to practitioners after they provided us their own components for refactoring documentation. Finally, we instructed them to use their own components when they extended the initial model. Our final model of refactoring documentation of code commits included 5 components (see Table 7.2), which is more extensive than the initial model (limited to 2 components). We found that there was a clear consensus among all 14 participants based on the interviews about these 5 components, but 3 of them mentioned 3 other possible components (highlighted in gray in Table 7.2). Thus, we decided to focus on the 5 components mentioned by all the participants in the survey with a large number of participants, as discussed in the next section, for the following reasons: (1) We wanted to make the conducted survey reasonable in terms of the time and not too long, (2) our goal is to design a first refactoring documentation model that is robust but can be extended later, and (3) we still validated quantitatively all 8 components, but without involving practitioners in the survey. We will provide more details in the results section.

Selection and hiring of participants. We advertised our study in mailing lists that covered 36 developers from industrial partners and top 30 developers from open-source projects, of our refactoring dataset (phase 3), selected based on their number of commits including refactoring (extracted using RefMiner). Out of the 21 participants who responded, we selected 16 of them (9 from industry partners and 7 from the open-source projects) based on the number of years of experience (should be more than 5 years), current positions (required a current active position in industry) and availability for virtual face-to-face interviews. Table 7.1 shows an overview of the background of the selected 14 participants after eliminating 2 participants due to their very limited feedback. They are all considered experts in refactoring in the development teams they work based on their several years of experience and extensive involvement in code rewriting projects.

Table 7.1: Demographic Information about the 14 interviewees.

Current Position	**# Participants**	**Avg. Exp. (Years)**	**Avg. Refactoring Exp. (1-5)**
Technical Lead	5	11.5	Very High (4.7/5)
Developer	6	8	Very High (4.7/5)
Manager/Executive	3	10	High (4.1/5)

7.2.3 Phase 2: Survey about the Identified Refactoring Documentation Components

After defining a model for refactoring documentation, we designed a survey aimed at a relatively large number of developers to understand their experiences and opinions about the importance of these identified components and their level of difficulty in finding and documenting each of them.

Figure 7.2 depicts the flow of our survey which was carried out using Qualtrics[1]. The

[1]https://www.qualtrics.com

numbered black boxes present the different sections in which participants answer questions. The survey starts with a welcome section where we explain the goal of the study and approximate completion time. We also provide an example of a commit message extracted from a GitHub repository of an open-source project for each of these refactoring documentation components.

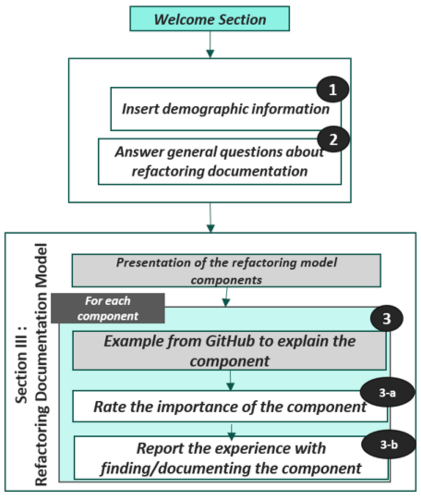

Welcome Section
1
Insert demographic information
2
Answer general questions about refactoring documentation
Section III : Refactoring Documentation Model
Presentation of the refactoring model components
For each component
3
Example from GitHub to explain the component
3-a
Rate the importance of the component
3-b
Report the experience with finding/documenting the component

Figure 7.2: Design of the survey.

We refined our survey through pilot sessions to remove ambiguities and reduce its completion time. The survey included several Likert scale-style questions about each component of our final refactoring documentation model. The final survey included the following sections.

- Section-I: Demographics: This section presents basic demographic questions to the participants. In particular, we ask participants about their current position and total experience in terms of years related to software development, as well as their level of expertise in refactoring, software quality assurance, continuous integration, software development, and code review (step ❶ in Figure 7.2). The collected information helps portray a clear and professional profile of our participants.

- Section-II: Refactoring Documentation in practice: This section contains nine questions related to refactoring documentation, documentation tools, and the need for refactoring documentation (step ❷ in Figure 7.2).

- Section-III: Refactoring Documentation Model: For this section we used our refactoring documentation model that we built in Phase I to understand the developers' experience with each of the components composing it.

In the survey, we introduced each component individually with a detailed description that includes examples extracted from GitHub (step 3 in Figure 7.2). We asked participants to evaluate the importance, difficulty of documenting/finding, and the frequency of documenting/finding the component based on their experience (step 3-a and step 3-b). Finally, we asked whether the refactoring documentation model is comprehensive and whether they would like to propose any modifications.

We used the snowball sampling of the interviews for our survey by reaching out to our industry partners and asking them to advertise it to their contacts. We also advertised it via social media, including Twitter, Facebook, etc. We analyzed 75 survey responses, after having discarded six responses based on the short time that they spent to take the survey (less than 10 minutes).

7.2.4 Phase 3: Quantitative Analysis

7.2.4.1 Data Collection and Sampling

As the first step in performing our quantitative analysis, we collected a total of 330,101 commit messages of 7,492 open-source projects downloaded from GitHub, containing 1,208,970 refactorings. Then, we extracted the applied refactoring by analyzing code changes of every extracted commit using RefMiner [252]. We selected RefMiner because it has demonstrated high precision and a recall score of more than 90% [235] for segregating refactoring changes from other changes.

To get our final sample for the manual analysis step, we adopted a guided sampling procedure by defining three criteria. First, we made sure that our samples of commit messages were submitted by *different committers.* The second criterion was that our projects should be different in *size* and *activity*, *e.g.* refactorings count and commits count. The third criterion was mainly related to the *refactoring types,* where we wanted to inspect commit messages that contained different types of applied refactorings (*e.g.,* extract class, move method, etc.) supported by RefMiner in its last version [235]. After applying the above-defined criteria, we performed further filtering on the 1,300 refactoring-related commit messages by discarding the commits that were empty, too short (one word), or full of special characters (e.g., '????? ?????').

Our final diverse sample consisted of 1,193 commit messages extracted from 612 open source projects with different natures and submitted by 893 different programmers. It is available as part of the replication package [247].

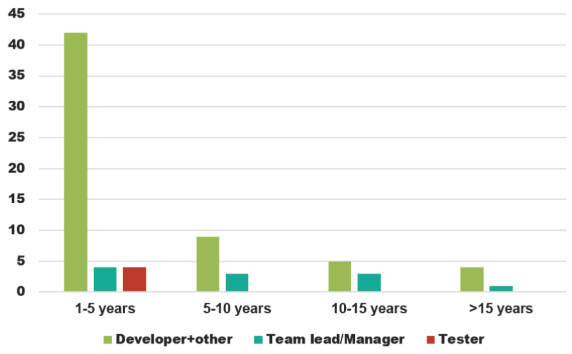
7.2.4.2 Manual Analysis of Commit Messages

After collecting refactoring-related commit messages and filtering them, our second step was to manually analyze each of the commit messages and annotate them based on the final refactoring documentation model obtained from the interviews. For each commit message, we manually tagged 0 or 1 for each of the 8 components (how, why, where, when, what, who, benefits, severity) if the corresponding component was documented in the commit message. The manual analysis process took 8 days in total and was performed by the authors of this research work.

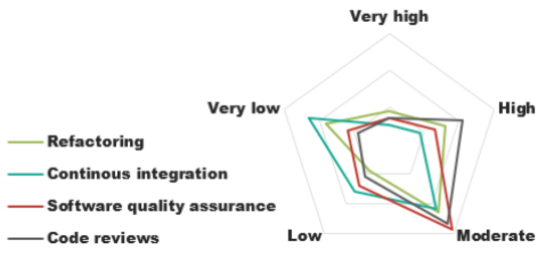
The manual analysis of commit messages to identify the components was performed by two of the authors independently for all the messages. A third author looked to the disagreements and decided which tag is correct. We discuss in the next section the Cohen's Kappa statistics for each component to compute the inter-rater reliability. It is relevant to mention that two of the authors have over 12-years of research and industrial experience on refactoring while the third author did many extensive similar experiments on large open-source/industry projects. Furthermore, the identification of the components is not a subjective task.

Component	Explanation
How	How developers document refactoring in practice. Answer to this question will help to understand whether developers mix functional and non-functional changes while documenting refactorings.
Why	The rationale of refactorings: The reasons behind the changes. For instance, the reason for changes could be improvements towards specific quality attributes or refactoring code smells
Where	Location of the applied refactorings i.e., at the package, class, or method level.
What	Refactoring Types: i.e., rename method, extract class, move method, etc...
When	Refactoring Timing: i.e., after fixing a bug, before releasing the code, during code review, or after introducing major functional changes
Who	The person(s) who made the changes.
Severity/Importance of the fixed problem	How critical, severe or important the problem was.
Benefits from the introduced changes	Estimated positive impact of the changes on the system.

Table 7.2: Refactoring documentation model



(a) Position and Years of experience.



(b) Level of Expertise.

Figure 7.3: Demographics of our participants.

7.3 Results

Figure 7.3 represents the demographic information of our participants. All the participants are very familiar with refactoring, software quality assurance, continuous integration, and code review.

RQ1: What are the refactoring documentation components in commit messages?

After the conducted interviews, none of the 14 developers removed any of the two preliminary components as described in Table 7.2. The participants reported a total of 9 components for refactoring documentation, where 7 of them were newly added during the interviews. After analyzing the distribution of our participants' feedback and categorizing the suggested components, we found that there was a strong consensus for five common components among all the interviewees. These components are: *when, why, where, what, and how*. However, we did not have enough evidence about the three other additional components (*severity of,* *benefits of, and committer (who)* of the applied refactoring) because they were suggested by a small number of participants (only 3 out of 14 participants). Furthermore, the ninth component consists of *"overall description of the code changes"*, which we removed from our analysis because it is too general and mentioned by only one participant. After aggregating and analyzing some of the suggested components, we obtained the final refactoring documentation model that we show in Table 7.2.

Our main goal from sketching this model is to define a rigorous set of components that practitioners can use to understand and document refactorings. Thus, we limited the online survey to the five main components to target a reasonable time frame to complete it (RQ2 and RQ3), but we considered the last three components of the model (refer to Table 7.2) with the least consensus in our commits analysis and model validation as discussed later (RQ4).

Key findings: The main refactoring documentation components reported by developers are *why, what, where, when, and how* code refactorings are applied. Some possible additional components with less consensus which are: *the committer, benefits, and severity* of applied refactoring, which can be considered but require further validation.

RQ2: How important is refactoring documentation *in general*, and each of its components *in particular*?

In this section, we investigated the practitioners' perspective about the importance and the challenges of refactoring documentation and the components of our proposed documentation model.

Figure 7.4 summarizes the responses collected from section 1 of our survey (step ❷ in Figure 7.2). The participants in our study rated the importance of refactoring documentation from *Extremely important*, to *Not at all important* as presented in part ❶ of Figure 7.4. In fact, 75% of the participants reported that refactoring documentation is important. In addition, one of the valuable results of our survey is presented in part ❷ of Figure 7.4 where the percentage of developers highlighting the need for an automated tool for refactoring documentation is 78.67%. This significant high percentage reflects the importance and the need for refactoring documentation in practice.

When asked about how often developers use commit messages and pull-request descriptions to understand their colleagues' code changes, 42.33% was the percentage for *Always* and 30.67% for *Most of the time* (see part ❸ in Figure 7.4). This information confirms the significant role of refactoring documentation in the process of comprehending the code. While our participants put emphasis on the importance of refactoring documentation, they reported the challenges associated with it (see part ❹ in Figure 7.4): time-consuming (37.18%), deadline pressure (26.28%), lack of tools to at least semi-automate the documentation (18.59%), and can be lost with other functional changes (14.74%). The participants also added other challenges including *"lack of common technical words to describe refactoring activities"*. The responses of the participants to the question *"Should refactoring documentation follow specific guidelines?"* are represented in part ❺ of Figure 7.4). 84% of the participants either *strongly agree* or *agree* that refactoring documentation should follow a guideline, which confirms the need to define these best practices.

Figure 7.5 and 7.6 show the importance of each component of our refactoring documentation model and whether developers wanted to keep/remove any of them. Interestingly, the majority of the participants reported that they would rather keep all 5 of the proposed components. More precisely, our participants provided positive comments describing the model as comprehensive. One participant said: *"I think your 5 components makes a lot of sense."* and another participant wrote: *"I'm not sure what there is to really add"*. The top two components that developers would highly agree to keep are the *why* and the *where* as shown in Figure 7.5. The high importance of the *why* and *where* highlights that developers, while seeking and documenting refactoring-related changes, focus more on the rationale for the changes and the location of the refactorings applied.

RQ3: ***What is the developers' experience with finding and documenting refactoring's components?***

Figure 7.7 shows the frequency and difficulty of finding and documenting for *why, where,*

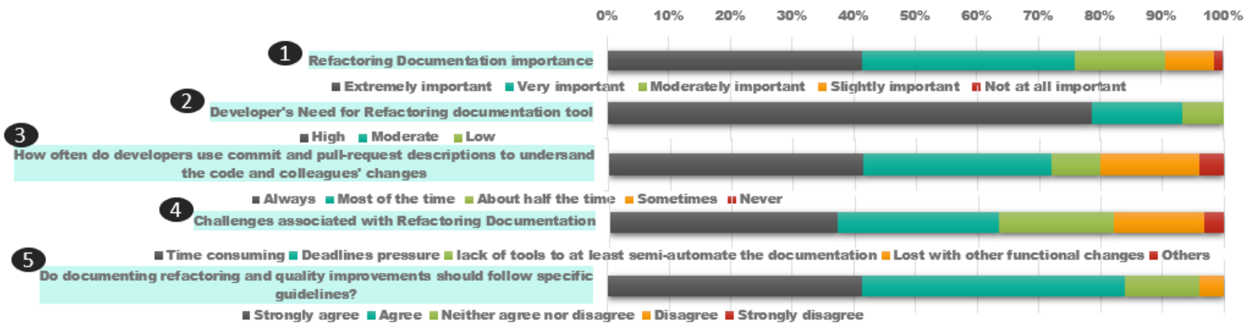


Figure 7.4: Information highlighting the importance of refactoring documentation (RQ1), according to the survey results.

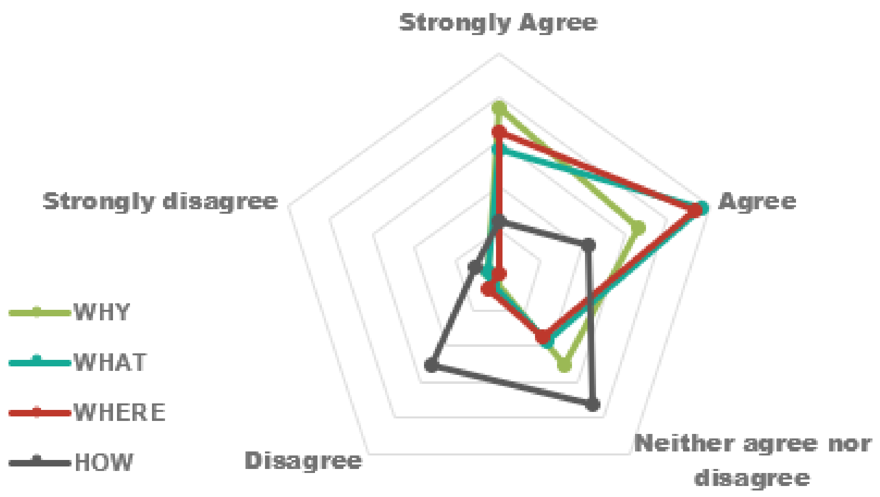


Figure 7.5: Importance of the refactoring documentation components.

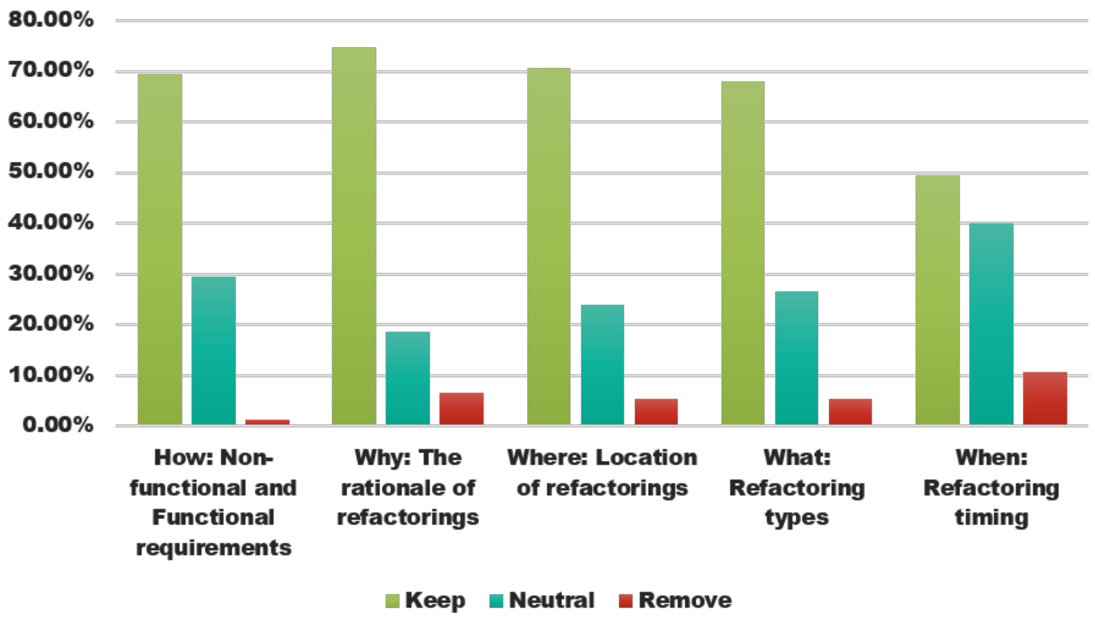


Figure 7.6: Participants modifications to the components (survey results).

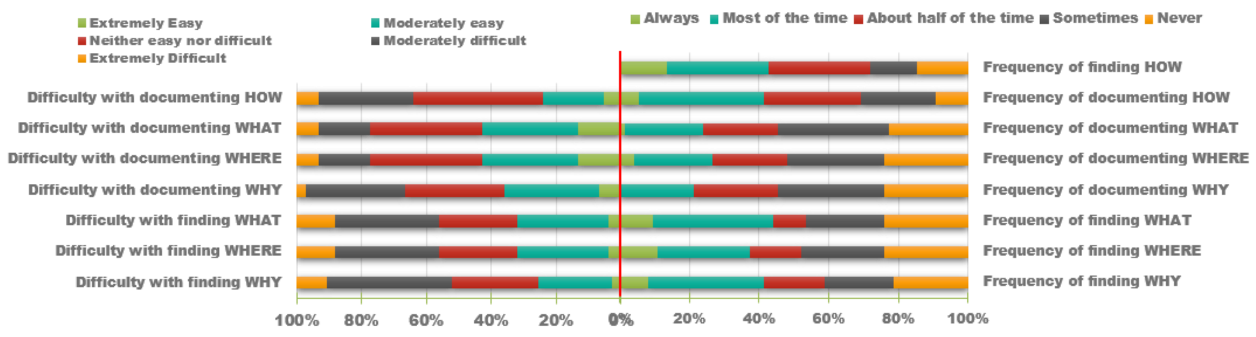


Figure 7.7: Experience of practitioners finding and documenting *why, where, what, and how* (survey results).

what, and how components; Figure 7.8 shows the experience of practitioners finding and documenting the *When* component as described below.

Frequency and difficulty of finding. The most frequent and easiest-to-find component is the *What* (type of applied refactoring), with 44% for "Always & Most of the time" answers, followed by *How* and *Why* components. This shows that developers easily recognize the applied refactoring types, which can be explained by the fact that there is a pre-defined and finite list for refactoring types.

Frequency and difficulty of documenting. Figure 7.7 shows that developers often mix functional and non-functional requirements in their commit messages. In fact, the most frequently documented and the easiest-to-document component is *How*, with 41.33% and 24% responses, respectively. Overall, the difficulty vs. frequency with which developers document different refactoring documentation components is almost similar for all components. For instance, while the *Where* (location of refactoring) represents the second most frequently documented component, it is the most difficult to document from the developers' perspective. These results highlight the contradictions between frequency and difficulty of documenting the different refactoring components. In fact, developers frequently document our model's components highlighting their importance, but they also continuously expressed a struggle to understand complex refactoring and their relevance; thus, they find them hard to document. This type of contradiction between *frequency and difficulty of documenting refactorings* could provide an explanation for why developers report that refactoring documentation is challenging, time-consuming, and difficult to write without guidelines, which can impact the consistency between the refactoring documentation and the actual code changes/refactorings.

Figure 7.8 investigates the experience of practitioners with documenting when refactorings are applied. While the left side of Figure 7.8 shows that developers most likely document refactoring applied *after introducing major functional changes (86.67%)* and *after fixing a bug (77.33%)*, the right side shows that the most challenging time to document refactoring is *after introducing major functional changes*. In fact, major functional changes require an extensive code cleaning and restructuring, which may result in a high number of different refactoring activities that can be difficult to document.

We were also interested in decomposing the rationale of refactoring, since it is the second most important component reported by practitioners (70.66% agreed about its importance, refer to Figure 7.5). The responses of the participants, reported in Figure 7.9, show that the documentation of the refactoring rationale can be decomposed further, mainly into *code smells* documentation (60% of responses) and QMOOD quality attributes [253] documenta-

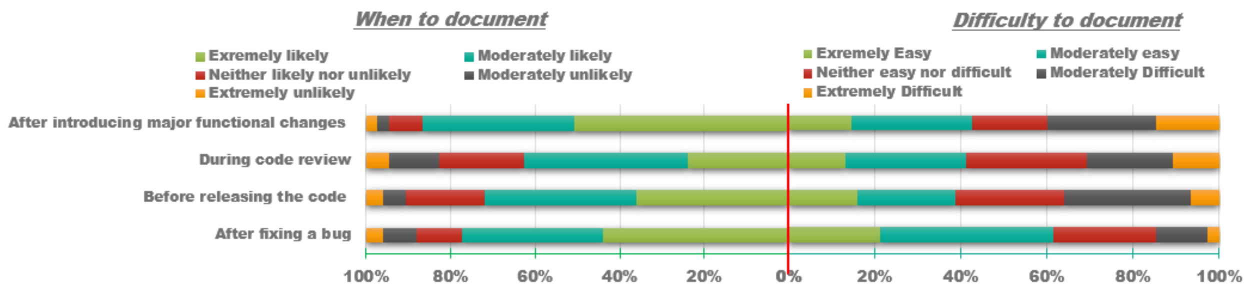


Figure 7.8: Experience of practitioners finding and documenting *When* (survey results).

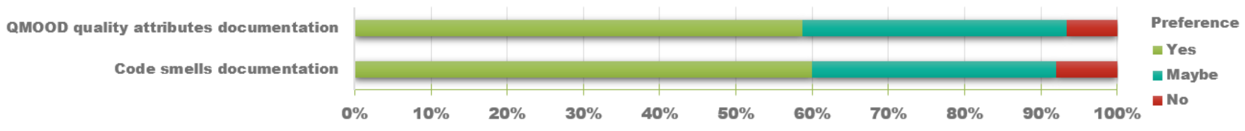


Figure 7.9: Decomposition of the rationale for refactoring (survey results).

tion (58.67% of responses). Participants did also report other possible reasons behind their applied refactorings. One participant said *"I also do refactoring to reduce impact of errors."* and other participants reported *preventing bugs* as a rationale for refactorings.

To get a better idea about the experiences of developers with documenting the rationale of refactorings, we asked the participants about their preference for documenting (1) each of the six QMOOD quality attributes [82] and (2) different code smells types [254]. Most of the practitioners are mainly interested in documenting *Understandability*, *Functionality,* and *Reusability* as a rationale for their applied refactorings. One participant said *"I found the QMOOD attributes very relevant since I mainly do refactorings to improve understandability and reusability of my code"*. The top 5 code smells types that were reported by our participants are: architecture, design, implementation, performance, and test smells. We have also found that developers focus more on documenting the refactoring related to restructuring the architecture of the system.

RQ4: ***Would comparing the experience of developers in requiring, finding, and evaluating the refactorings documentation with samples of actual refactoring commit messages reveal some inconsistencies?***

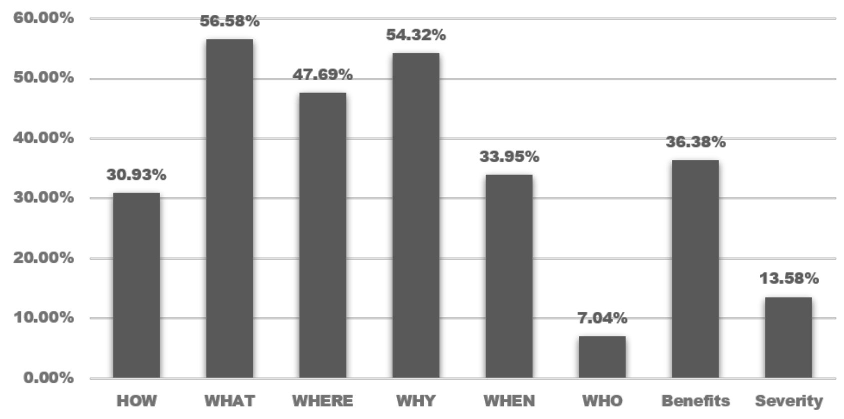


Figure 7.10: The percentage of each individual documented component in our commit messages sample.

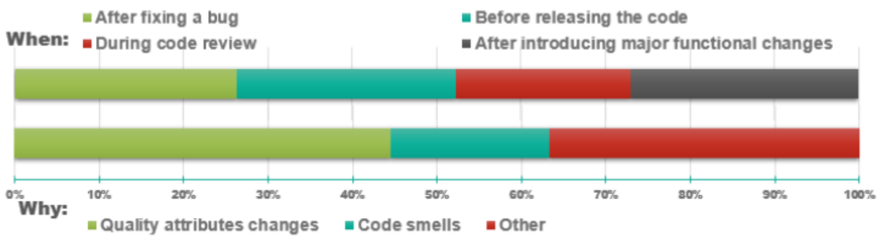


Figure 7.11: Distribution of sub-components of the *when* and *why* according to our quantitative analysis.

Importance of the components Vs. Actual documented components ratio. Figure 7.10 shows the frequency of finding/documenting each individual component of the model (Table 7.2) in our commit messages sample. Interestingly, the most documented components in practice are *why* (56.58%), *what* (54.32%), and *where* (47.69%). However, the order of the components in terms of the importance score reported by our participants is reversed, but they still have similar rates. In fact, Figure 7.5 shows that the most important component is *where*, followed by *what* and *why*. This difference shows that there is a gap between actions and opinions, and thus the *where* component should be better documented in existing commit messages.

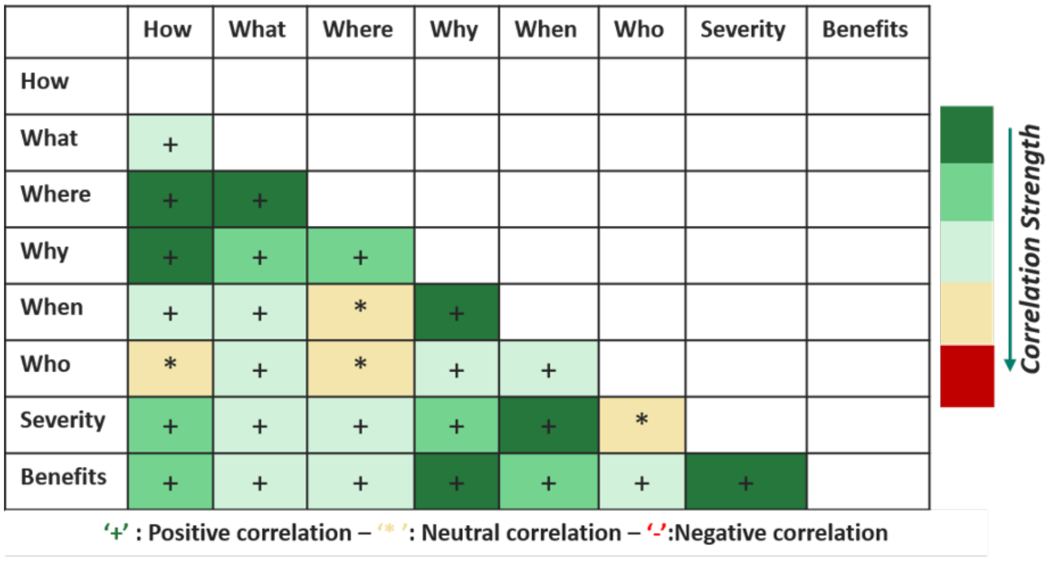
Despite the high importance of refactoring documentation and the positive feedback about the components of our documentation model, we found that in practice, a reasonable number of commit messages contain at most only one component documented. In fact, 14.50% of the inspected commit messages have none of the components documented, and 14.59% contained documentation for only one component. Figure 7.10 also shows that the *Benefits* component is highly documented by developers in practice. This information was not emphasized by our participants during the interview sessions. We considered this component as an optional component, but based on our quantitative analysis, it can be useful to add to our refactoring documentation model in the future after further investigation.

Regarding the documentation of *the rationale* of refactorings, our commits analysis findings (see Figure 7.11) confirm that it can be further decomposed mainly into *quality attributes changes* and others that include *fixing bugs* and *functional purposes*. Code smells was the least documented by developers in practice, which is in contradiction to our practitioners' responses (Figure 7.9). One of the reasons that could explain this contradiction is that many developers lack the knowledge of formal definitions of these code smells, such as *'feature envy'*, and may use different vocabulary for the same meaning.

Figure 7.9 also presents the decomposition of the *When* component in practice. The results show that developers often document refactorings *after introducing major functional changes* (12.74%), after fixing a bug (14.41%), and *before releasing the code* (12.31%). Both *after introducing major functional changes* and *before releasing the code* were also reported by our participants as the most frequently documented components.

We note that the disagreements between the authors are very rare when identifying the components in the refactoring commit messages. The Cohen's Kappa results show that min Kappa was 0.909 for the HOW component. In case of any disagreements between two authors who checked the messages independently, the opinion of the third author is considered.

How developers document the different components in practice. Our manual inspection of commit messages shows that while developers may document in isolation the different components over multiple commits, they rarely document all of them in a single commit. For instance, only 0.92% (11 commits) contained all the components documented. Moreover, one interesting result is that at least two components out of the 8 investigated components are documented by developers in practice. In addition to investigating how many components developers document in practice, we were interested in assign-



	How	What	Where	Why	When	Who	Severity	Benefits
How								
What	+							
Where	+	+						
Why	+	+	+					
When	+	+	*	+				
Who	*	+	*	+	+			
Severity	+	+	+	+	+	*		
Benefits	+	+	+	+	+	+	+	

'+' : Positive correlation – '* ': Neutral correlation – '-':Negative correlation

Correlation Strength

Figure 7.12: Correlation between the components.

ing weights to the components and capturing their pairwise correlations. Our correlation analysis results are represented in Figure 7.12, which shows that there is a strong correlation between *where-how, where-what, why-how, when-why, severity-when, benefits-why, and benefits-severity.* This correlation analysis will help in defining the context for documenting each component. Understanding how the components are linked together and investigating their pattern of appearance in the actual documentation can help developers in choosing a subset of components to document.

Validation of the proposed refactoring documentation model. All the commit messages were evaluated by the interviewees in terms of quality with an overlap of 207 random commit messages. Thus, each of the 14 participants evaluated between 14–15 overlapping-messages used to calculate the Cohen's-Kappa statistics which was higher than 0.85. This confirms that there is a clear consensus when evaluating the quality of commit messages using the Likert Scale.

We first checked the frequency of each component of the proposed model in the commit messages with different levels of quality (from very poor to very good). Figure 7.13 shows that all the five components (4W+H) were frequent/strongly-correlated with good/very good commit messages with almost equal-distribution, and they are much less frequent in

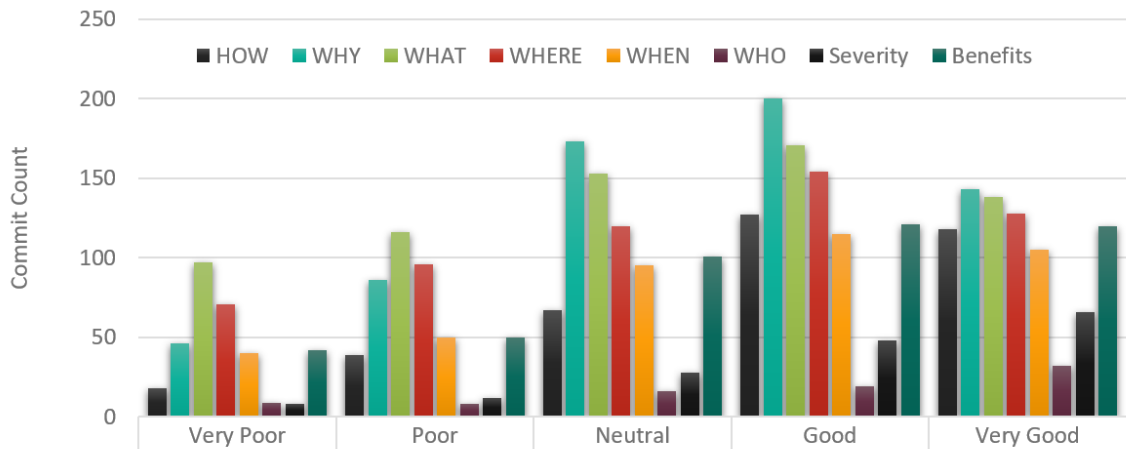


Figure 7.13: Distribution of the 5 components per quality score.

poor/very poor commit messages: when new components are added then the quality of commits message increased. It is clear that the WHY is the most important/frequent component in very good/good messages and the WHO/Severity are the less frequent ones which confirm our conclusion to remove WHO and Severity from our final documentation model. The benefits component is overlapping with the WHY which explains its frequency/correlation in good/very good messages thus we removed the benefits component from our final model.

To statistically study the correlation/association of the five components of our final model 4W+H and its variations with the quality of refactoring commit messages, we run two tests which are: GOODMAN AND KRUSKAL'S and Chi-Square test. Our aim is to investigate the correlation between the quality of commit messages and the different variations of the proposed taxonomy (different combinations of the components). The number of variations of 4W+H is 5 where each time we remove one component out of the five and we study again the correlation with the quality categories of commit messages. Thus, the two studied variables are the quality of the commit messages and the aggregation of the components of the studied model as a single variable.

The comparison between the different variants of our 4W+H model shows that the aggregation of all of them has the strongest correlation with the quality of commit messages. We have significant association between our model 4W+H and its 5 variations with the quality of documentation (all the p-values for the 6 tests are less than .0005). Based on the Cramer's V value (effect size), the 4W+H and Quality variables have the highest association strength (0.527) and 3W+H (no Why) Quality has the lowest. Thus, the complete model (all the 5 components) contributes better to the quality of the documentation. The detailed statistical test results can be found in the appendix [247].

Table 7.3: Statistical analysis (model variants and quality of messages)

Model variants	Chi-Square Value	ρ-value	Effect Size
4W-H	1323.343	< 0.001	0.527
4W	1163.564	< 0.001	0.494
(3W+H)' (no Why)	1178.095	< 0.001	0.497
(3W+H)" (no Where)	1178.095	< 0.001	0.497
(3W+H)"' (no What)	1235.875	< 0.001	0.509
(3W+H)"" (no When)	1110.499	< 0.001	0.482

7.4 Discussions and Implications

Better understanding of refactoring documentation. A guideline or a rigorous pre-defined template for refactoring documentation can be a starting point for improving how developers document their refactoring. Moreover, a pre-defined template makes the discussions between practitioners about refactoring and its impact on the code consistent and more understandable. However, we do not advocate that every identified component in this study should be documented by developers in all the cases. In fact, our participants mentioned different levels of importance and frequency to look for these components (refer to Figures 7.5 and 7.7), and our goal is to understand the pieces of information that developers may search for when understanding refactorings.

Refactoring vs code changes documentation. Some of the refactoring documentation components identified in this study can be used for documenting regular code changes as well. However, some others are specific to refactoring such as code smells, QMOOD quality metrics improvement, and refactoring types. We discussed existing studies in the related work section about documenting code changes, and this study can be extended to understand the differences in the importance and frequency of the documentation components of refactorings versus regular code changes.

Implications for educators and practitioners. Our results provide a common ground for documenting, discussing, and assessing refactoring and its impact on the system. This common ground will help educators to disseminate multiple dimensions of refactoring documentation in commit messages.

Though we do not expect practitioners to document all the refactoring documentation components independently from the context, we expect them to judge which components are more relevant and adequate for their specific context. Team-leads can work with developers to establish customized guidelines from this study for documenting refactorings.

These guidelines could trigger developers to capture the impact and the rationale of their code changes appropriately for each situation, developing beneficial habits and long-lasting documentation.

Implications for researchers and tool builders. We observed in Section 7.3 that, in practice, software developers fail to document refactoring components. Such practices make it harder for other software developers to comprehend the introduced refactorings. With this proposed empirical study, we described the scientific foundations required to generate refactoring documentation. For instance, a new checker and generator for CI platforms could be developed to generate refactoring documents. Specifically, the tool could support the following features: (1) check the consistency and the completeness of refactoring and non-functional requirements documentation, (2) offer suggestions to document and complete the missing important components, or (3) generate a complete refactoring documentation and present it for the developer's approval.

7.5 Threats to Validity

Construct validity. Some threats can be related to the construction of the proposed refactoring documentation model. To mitigate the threat, we made sure not to limit the interviews to a list of predefined questions. We also encouraged the participants to think openly without providing any implicit inputs and bias. We gave them enough time to formulate their ideas. We have included a free-form "Other" option in the answers of interviews and survey questions to enable them freely express their feedback and ideas.

Internal validity. Some online participants of the survey may have decided to take it because they had greater interest/enthusiasm in documentation than others, thus they could provide a "biased view" of the investigated phenomena on refactoring documentation. To mitigate this threat, we ensured that the selected population is composed of practitioners (e.g. Figure 7.3) with different roles, background, different organizations, and may have different views on the refactoring documentation issues. A typical co-factor in survey studies is the respondent's fatigue bias. We mitigated this threat by running a pilot study with four developers and four PhD students to make sure that the survey could be answered within 20 minutes. Another internal threat to validity in our study is drawing conclusions based on recollected memories. Thus, we allowed enough time for participants to remember their experiences with refactoring documentation.

External validity. The selected practitioners may not represent the very large population of developers. To mitigate this threat, our participants were chosen from a diverse population, with diverse expertise and years of experience (e.g. Figure 7.3).

7.6 Conclusion

We used a combination of interviews and a survey to understand refactoring documentation from practitioners' perspective. We started first with a set of interviews with practitioners to define a refactoring documentation model. Then, we performed a large online survey to gather the experiences of practitioners with the importance, frequency, and difficulty of refactoring documentation for the different components of our model. We found 5 main important refactoring documentation components for practitioners.

The outcomes of this empirical study can be used to improve the quality of refactoring documentation. Furthermore, researchers and tool builders can use the discovered components and the experiences of the developers to build refactoring documentation generation tools.

CHAPTER VIII

Commit Message Generation of Composite Changes

8.1 Introduction

In modern software development, developers collaborate together by submitting source code changes onto version control systems (VCS), e.g., Git [242]. These allow developers to store different versions of the system being developed and archive them as *commits*, i.e., individual changes to a file (or a set of files). In so doing, developers can document the newly committed code changes through *commit messages*, which are short natural language descriptions of the modifications performed. Previous work has shown that high-quality commit messages maintain the rationale of the code changes, in addition to favoring program understandability [78, 251] and developer's productivity [255].

Unfortunately, commit messages in reality are often empty, have very short strings or lack any semantic meaning [256]. For this reason, the research community has been actively working on automated techniques to support developers when describing code changes through commit messages [76, 257, 258, 69]. In this respect, the last few years have seen the rise of two main lines of research. On the one hand, researchers have proposed search-based approaches [64, 65, 66] that find the most similar commits in the project's history in order to reuse or adapt their messages. On the other hand, deep learning models have been investigated: most of them are based on neural machine translation (NMT) and generate short commit messages based on diff files [67, 68, 66]. Later studies have also proposed different variations of a vanilla NMT model to improve the quality of the generated commit messages. For example, a pointer-generator network is added to treat out-of-vocabulary words [66]. Xu et al. [69] modified the encoder to take two inputs: code semantics and code structures for commit message generation.

While these previous research studies reported promising results, with both search-based and NMT models being able to generate commit messages similar to those written by developers, most of the currently available techniques are particularly suitable only when dealing with commits composed of a few *atomic* changes, i.e., operations where developers apply a set of disjointed changes, like additions/deletion of lines of code, to multiple files. However, they might not be as accurate when facing what we define as *composite* changes, namely, a set of conceptually related modifications that are intended to implement a unique high-level code change.

To highlight the limitations of existing approaches, let us consider the example of a commit[1] of the OpenEngSB framework- a technical integration platform for software tools.[2] From a purely structural perspective, the commit modifies 13 files by adding 134 lines of code and deleting 41 of them. Analyzing it more deeply, the actions done in the commit had the main goal of performing a number of composite operations: the classes `EncryptionException`, `DecryptionException`, and `MarshalException` were subject to a *Move Class* [234] in order to place them in a more coherent package. The committer applied rename operations aimed at providing a better name to the class `MarshalException` and its methods. Finally, s/he also applied an *Introduce Parameter* refactoring to the `GenericSecurePortTest` class. The remaining modifications updated the references of the changed classes.

This example presents a key challenge for the existing approaches: the diff file contains more than 300 lines with 175 additions and deletions. Representing a similar diff is challenging and would require additional information to be properly exploited by existing approaches.

[1]github.com/openengsb/openengsb/, commit:99154fe7f14e1f7941e5a1bdda1b79a6d896f44c
[2]Link: http://openengsb.org/.

Indeed, it is rare to have other diffs that are similar in text. As a consequence, a search-based approach is likely to fail because it may not find similar commits in the change history, providing a poor solution. For this exemplary commit, indeed, the state-of-the-art search-based technique proposed by Liu et al. [65] would output the following commit message:

Commit message. *FFT convolution almost done Problem with rearranging of size is odd (e.g. 315).*

An improved representation of these composite changes that considers enhanced mechanisms to structure similar diff files might potentially lead to better outcomes. Similarly, an NMT model would not perform well if its training set does not contain enough of these cases. For the example above, the technique proposed by Jiang et al. [67] would deliver the following commit message:

Commit message. *Moved to class.*

This example clearly highlights that existing techniques do not properly treat these types of composite commits and as a consequence cannot generate meaningful commit messages. We argue that novel strategies able to provide additional information when training these models, e.g., the type of changes, could induce a boost in their performance to generate better commit messages that capture the logic behind applying a sequence of atomic changes.

In this chapter, we aim to overcome the limitations of existing techniques. We first mine over 7,000 open-source projects from GITHUB and build a new dataset of 53,066 composite code changes. Afterward, we devise a novel mechanism that improves the representation of composite code changes: we (1) add change types as input to enrich the description of the commits; (2) use a placeholder mechanism to standardize diff-format text; and (3) build a language model that is pre-trained on a larger corpus to embed rare words based on their context. We evaluate our approach using the human-written commits contained in the built dataset, finding improvement over the start-of-the-art techniques in terms of BLEU score. We have also conducted controlled experiments with 12 practitioners and professional developers to evaluate the semantic meaning of the generated commit messages.

Our results show that adding change types as a second input to an NMT model or a search-based method can increase the BLEU score by more than 3 points. Similarly, using the placeholder mechanism increases the BLEU score by more than 2 points for both the NMT and the search-based method. Finally, the BERT embeddings increase the BLEU score of the NMT model by 2 points. The manual evaluation results are in line with our findings of the BLEU scores.

Replication Package. All data, and the anonymized developers' answers are available in our replication package [259].

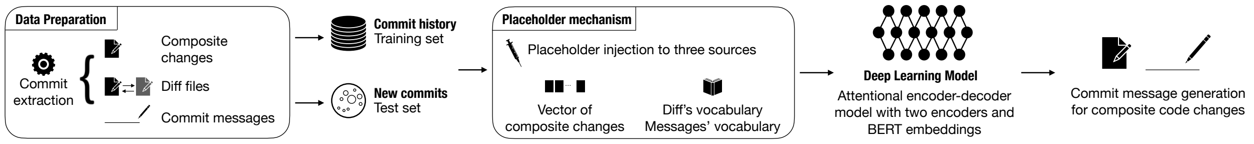


Figure 8.1: Overview of the methodology.

8.2 Methodological Overview

Figure 9.2 overviews the main steps of our methodology applied to address the problem of generating meaningful commit messages for composite code changes. As depicted, we designed our research with two major steps. At first, we required a large amount of data pertaining to composite code changes made on real software projects. In this regard, we devised a set of data collection, cleaning, and preparation steps aimed at retrieving as many composite code changes as possible, other than human-written commit messages, that allowed us to conduct large-scale experimentation. These pieces of data were split to create a training and a test set. Section 8.3 describes the dataset construction and preparation.

On the basis of the collected dataset, we implemented our novel strategy to generate commit messages for composite code changes. Section 8.4 reports the two components forming the technique, namely, (1) the placeholder mechanism and its injection within three sources like vector of composite changes, diff's vocabulary, and commit message's vocabulary, and (2) the definition of an attentional neural network with two encoders and a BERT embedding mechanism. These two components allow the generation of commit messages, whose effectiveness was later assessed through the empirical investigation presented and discussed in Section 9.4.

8.3 Dataset Construction

8.3.1 Data Collection

To collect a large amount of data that can be later exploited in the context of techniques for commit message generation, we mined a random set of 7,492 open-source projects from GitHub. This step provided us with a total of 330,101 commit messages. Given the amount of data mined, a manual validation of composite code changes would be excessively expensive; therefore, we had to identify an automated strategy to distinguish atomic from composite changes. After some manual analyses, we decided to collect composite changes by mining refactoring-related commits while relaxing the pre- and post-conditions to preserve the behavior. In fact, the composition of atomic changes led to a refactoring type such as extract class, extract method, etc. However, we relaxed the behavior preservation check, as many of these composite changes would have enabled the collection of data related to bug fixes and features change/integration rather than just structural changes. On the one hand, these commits are *by nature* composite, i.e., developers who refactor source code apply unique conceptually related changes to improve the internal structure of source code. On the other hand, collecting composite changes from different commits would have been a source of imprecision that would increase threats to internal validity. Thus, we found that the best way is to get inspiration from the definition of refactoring types, since they are comprehensive of possible composite code actions, while removing the conditions of the behavioral preservation so as to collect data beyond reasons related to changes in the structure. This choice is the best compromise in terms of accuracy of the information retrieved. Nonetheless, it is important to remark that this choice is not binding for the inner-working of commit message generation techniques. Our goal is indeed one of generating reasonable commit messages from conceptually-related changes independently from their final goal. From an operational perspective, we used the GitHub APIs [260] to download GitHub repositories. Then, we extracted the applied composite changes of every commit in all the downloaded repositories using RefactoringMiner [235] without considering the pre/post conditions of behavioral preservation. We selected RefactoringMiner because it has shown high precision and a recall score of more than 90% [235]. By applying the tool on the repositories, we collected a total of $1,208,970$ composite operations. Additionally, we also collected diff files for all the commits using the command `git diff`. The full list of composite types is online [259].

8.3.2 Data Cleaning and Structuring

Significant preparation was necessary to build a dataset suitable for commit message generation experiments. We perceive this non-trivial preparation as an important contribution to the research field, as a similar dataset of such size and complexity is not available for documentation of composite code changes documentation.

The first step in our pursuit is to have a cleaned and well-filtered dataset to avoid noisy data that may lead to misleading results [261].

We started by preprocessing our collected dataset (applied composite changes, commit messages, diff files, and information about committers and projects) by removing non-alphabetic characters and discarding empty or too-short commit messages. We also removed the merge, rollback, and bot-generated as well as trivial commits as suggested by other relevant studies [67, 66]. The full list of our cleaning steps is available in our online appendix [259]. To reduce vocabulary size, we removed commit IDs from diff files. After the above preprocessing steps, we had $269,252$ commits remaining.

8.3.3 Composite Changes Data Filtering

Starting from the output of the previous stage, our goal was to retrieve high-quality commit message from composite change commits. Hence, we devised a filtering procedure aimed at discarding those commits that were not accompanied by good commit messages. This procedure was based on:

Step 1. Get composite changes-related commit messages. We used the keywords provided by Rebai et al. [75] to identify commit messages explicitly referring to composite changes. We got around 40,000 commits.

Step 2. Compute the rank of the sentence structure. Since we want to have well-written commit messages similar to human-written ones, we extracted the sentence structure of each commit message. The sentence structure extraction has two main stages which are tokenization and tagging. We used Spacy and Stanza [262] (previously called CoreNLP) in the context of these two stages. To compute the rank of a sentence structure, we (1) extracted the sentence structures of all the commit messages, (2) computed the number of occurrences of each sentence structure, and (3) sorted the sentence structures by their occurrences. We hypothesize that if the structure is popular, then it is probably a good structure for a commit message.

Step 3. Count composite change operations in the commit. For each of the commit messages, we computed the number of composite change operations applied. We hypothesize that if a commit contains a higher number of composite changes, it has a higher chance to be more complex to document.

Step 4. Count the developer's composite changes. Our fourth filtering criterion is based on *developer profiling*. If a committer is applying a high number of composite actions and a high number of commits, then there is a high chance that the other commits by the developer are also include composite changes. Therefore, for every committer, we computed , the sum of the numbers of composite changes and commits.

Step 5. Computing the filtering score. After performing all four of the previous filtering steps separately, we computed a filtering score of each commit of our initial database. The filtering score of a commit is the arithmetic average of the rank of the sentence structure, the number of composite operations in the commit, and the number of the developers' composite changes and commits.

All the above-mentioned values were normalized using min-max scaling. After scoring the commits, we manually examined 488 commit messages belonging to different scores ranges. We observed that (1) the higher the score, the higher the correlation with composite changes and the better the commit message; and (2) most of the good commit messages have a score above 0.25, which we chose as a threshold for selecting more commit messages from the remaining data. Finally, the scoring of the commits helped us increase our 40,000 commit messages extracted from Step 1 with high-quality commit messages. After cleaning, filtering, and scoring the commits we were left with $53,066$ commits that we think are composite changes-related with well-written structures. Finally, we randomly divided our dataset into 3 sets (training, validation, and test sets) using 70%-15%-15% splits. The training set contains 37,146 commits. The validation and testing sets each contain both 7,960 commits.

8.3.4 Prepare the Dataset for the Models

As a final stage, we conducted further filtering and tokenization steps to get the data ready for our models. In particular, we filtered the commits by the size of diffs and removed the commits that had a diff file larger than 1MB since such large files are not suitable for NMT. After combining all the filtering steps done on our dataset, we had $269,252$ commits remaining. Finally, we tokenized the extracted commit messages, diffs, and performed composite changes. Since NMT needs predefined vocabularies for its input sources, we chose to work with 50,000 tokens for the diffs, as was often done in the previous studies [67] and other NMT models [263]. In the training set, the number of distinct composite change types was 18 (Encapsulate Field, Increase Field, Decrease Field, Pull up Field, Push Down Field, Move Field, Increase Method, Decrease Method, Pull Up Method, Push Down Method, Move Method, Extract Class/Method, Extract Superclass, Extract Subclass and Rename Method-/Class/Field) and our commit messages had 30,000 distinct tokens. Thus, we selected all 30,000 tokens to be the vocabulary of commit messages and 18 tokens as the vocabulary for composite changes.

8.4 Generating Commit Messages for Composite Changes

8.4.1 Component #1: Placeholder Mechanism

The placeholder mechanism consists of temporarily replacing the non-standard tokens including emoticons, brands, numbers, and other identifiers with special placeholder tokens during translation without actually translating them. Previous studies [264, 265, 266] reported that this mechanism improves translation accuracy even with noisy texts. To help NMT learn changes more efficiently, we also standardized our collected commit messages by replacing all the unique names such as class, method, and package names as well as committer names, bugs numbers, and versions, with their corresponding placeholders. Thus, these identifiers were not treated as individual words in this study. This preprocessing step of placeholders relies on Java naming conventions in order to distinguish between methods, classes, and packages names. To replace identifiers in a commit's change types, we extracted the impacted entities based on the rules of the composite operations' controlling parameters. For example, for change type like "extract class," the first parameter is the source class and the second parameter is the new extracted class, both of which should be replaced with placeholders.

Table 8.1 lists the placeholders used in our dataset. The unique names and identifiers are saved and categorized based on their types. Thus, for every commit we have the list of the replacement placeholders that can be used during the **refinement step**. The outcome of this step is a registry that maps placeholders to their actual names. This mechanism has two main advantages: (1) it reduces the vocabulary size and (2) it enhances the similarity between parts of code that otherwise would be very different because of the unique identifiers (e.g. different class names). The generated commit message gets postprocessed in the refinement step.

Table 8.1: List of placeholders.

Placeholders	**Description**
#class#, #package#, #method#, #attribute#, #interface#	Identifiers of source code entities
#version#	A specific release of a software program
#nbr#	Any number that can represent: bug, issue or number of new added features
#@person#	The name of the collaborators/committers

This refinement step consists of searching for placeholders in predicted messages and replacing them with their actual words from the registry created during the preprocessing step while checking the actual applied change types. One main challenge encountered is that sometimes there are different candidates for the same placeholder and the change types cannot always help in choosing the best one. Thus, the simplest solution to deal with this challenge is to randomly select from one mapping from the candidates found in the registry.

8.4.2 Component #2: Deep Leaning Model

This section describes our proposed neural model. The model assumes a typical NMT architecture, which is based on a slightly modified attentional encoder-decoder [267]. It has two encoders- one for composite changes and one for diffs data. The model takes our generated BERT embeddings as initial embeddings. In the rest of this section, we give more details about our enhanced model.

8.4.2.1 Adding Change Types as a Second Input to the NMT Model

While software composite changes can be categorized using different methodologies, refactoring operations are used here for labeling changes because (1) refactoring operations are abstract information hidden in the concrete changes that is more difficult for the models to learn, and (2) many code changes can be labeled as one of the refactoring operations, which can be a good starting point for our method.

In our first method, the types of refactoring operations are used to label composite change types. The complete list of refactoring operations retrieved by RefactoringMiner is in our online appendix, yet it includes most of the well-known refactoring types studied in the literature, e.g., *Extract Class*, *Rename Method*, and *Move Field*. We note that the collected data exclude the behavior preservation part to take inspiration from the refactoring types while making the scope broader to include both functional and non-functional changes.

More specifically, each composite change operation takes a list of controlling parameters [268]. For instance, the *Extract Method* operation is represented by the following tuple: *Extract Method (sourceClass, sourceMethod, targetClass, ExtractedMethod)*. The operation extracts a block of code in *sourceMethod* in *sourceClass* to *ExtractedMethod* in *targetClass*. For each commit, we obtained a list of composite change operations as its **change types**. The vector of change types, which may also be called the composite changes vector, is represented in the figure below—note that controlling parameters are omitted from the figure due to space limits.

MoveClass	**RenameClass**	**MoveMethod**	**.......**	**PullupAttribute**	**PullupMethod**

In the following sections, we describe our method of adding the composite changes to the NMT model.

Neural machine translation (NMT), also called the neural sequence-to-sequence model [269, 270, 271], consists of two recurrent neural networks, an encoder and a decoder. The encoder converts a sequence/sentence into a fixed length vector representation. The decoder takes the vector representation as an input and converts it to another sequence. The input sequence of the encoder and the output sequence of the decoder are in different languages. In our work, we exploited an NMT with two encoders, each encoder taking different inputs separately. Existing work [272, 273] combined different data sources for image captioning (e.g., merging convolution image output with a list of tags). Additionally, to generate subroutine summarization, LeClair et al. [274] proposed an attentional encoder-decoder with two encoders consuming two different data sources (code/text data and AST data). Our model architecture is inspired by the models used in these studies. The main differences between our Neural Machine Translations (NMTs) model and LeClair et al.'s are that (1) we adapted two attention mechanisms in our model and concatenated the vectors from each attention mechanism to create our context vector; (2) we used GRU instead of CuDNNGRU; and (3) for the model hyperparameters, the number of recurrent units was decreased from 256 to 128, the embedding dimension for the commit messages was increased from 100 to 128, and the embedding dimension for the diff text was increased from 256 to 768.

More specifically, the first encoder is to process the change types we collected, and the second one is dedicated to processing the diff files. We modified the first encoder presented by LeClair et al. [274] to work with our change types which are different from their input source, AST. We trained our model on 37,146 triples (diffs, composite changes, human-written commit messages) with a validation set of 7,960 (prepared in 8.3.1). We ran 20 epochs for training, using Adam with an initial learning rate of 0.01.

8.4.2.2 A BERT Embedding Technique

Pretrained language models provide significant improvements for a range of language understanding tasks [275, 276, 277, 278]. The main idea is to train a large generative model on a vast dataset and use the resulting representations on tasks for which only a limited number of labeled data are available. The BERT model architecture [278] is based on a multi-layer bidirectional transformer [279]. Instead of the traditional left-to-right language modeling objective, BERT is trained on two tasks: predicting randomly masked tokens (MLM) and predicting whether two sentences follow each other. To the best of our knowledge, there are rarely public pretrained BERT [278] models in the software engineering domain and there is no out-of-the-box model for diff text. The original BERT was pretrained on Wikipedia and BookCorpus [280] with a total size of 13 GB, which mainly focus on natural language and have a large gap to the diff-format texts, so it cannot be used in our task.

Dataset for Pretraining. As one of our main contributions, we pretrained a BERT model from scratch on 70k diff files randomly sampled from the dataset in Section 8.3.2. The dataset led to a size of 5 GB due to implementation and hardware limits. More specifically, we collected 70k diff files by the following three steps: (1) filtering out files containing non-Latin characters, such as Chinese, Japanese, and Korean; (2) filtering out extremely large diff files, whose size are over 1 MB; and (3) randomly sampling 70k samples from the remaining files (211k in total). The purpose of the first step was to build an efficient subword-level vocabulary for our BERT. According to machine learning convention, our pretraining dataset was independent of the test dataset. This setting was enforced to make sure that there was no information leakage from the test set to the training set.

Segments. We set the input sequence length limit at 512, which is a common setting for pretraining BERT. However, many diff files have more than 512 tokens, and for simplifying the process, instead of using a sliding window approach, we chose to split each diff file into segments and used segments as input sequences. The rationale for this processing method is that diff files are naturally composed of multiple blocks of continuous changes. Specifically, we split the diff files by "diff --git" and "@@."

Implementation and Training Details. Rather than using the original BERT training process, we exploited RoBERTa [281], which has the same architecture as BERT [278], but optimized the overall training scheme. One big difference between RoBERTa and the original BERT is that RoBERTa has only one task, MLM. We used the implementation provided from Huggingface.[3] We uset 12 layers of transformer decoder blocks, each block having 12 self-attention heads with 768 hidden dimensions, for 132M parameters in total. We use Adam with an initial learning rate of 5e-5, β_1 of 0.9 and β_2 of 0.999. The batch size was 32 (16 on each GPU). Our training stopped before one epoch finished, at step 73.5k, after 23 hours of training on an Nvidia Quadro P6000 GPU and an Nvidia Titan RTX. At the time of stopping, the curve of loss had flattened, so we choose to use this pretrained model instead of continuing the pretraining.

Get embeddings. The conventional way of using a pretrained language model is to finetune it toward a downstream task. Instead, to quickly test our hypothesis of the usefulness of BERT embeddings in our task, we used embeddings generated by BERT directly as the initial embeddings for the NMT model. BERT generates embeddings for a token dynamically, which means each embedding is informed by the context of a word. To get accurate embedding for each word, we needed to store every embedding for every appearance of the word in the training set. This would explode the storage and increase the complexity of the NMT model's training process. Therefore, as the first step toward using the diff-based BERT model, we applied a simple mechanism where we ran BERT on the training set (the training set for the NMT model) and for each token, we stored only one embedding. In the end, we had a matrix with a size of 30k x 768, which was used as the initial embeddings for the NMT model.

To examine the quality of the embeddings that BERT generates, we inspected the embeddings using t-SNE visualization [282]. Figure 8.2 shows the visualization of a partial set of the vocabulary. The green group includes the words *done, called, added, returned,and disabled*, which tend to be finished actions; the orange group: *ask, use, apply, need, and delegate* is more about requests; the red group: *swing, awt, Dimension, Border, and JPanel* is Java Swing related.

[3]https://huggingface.co/transformers/

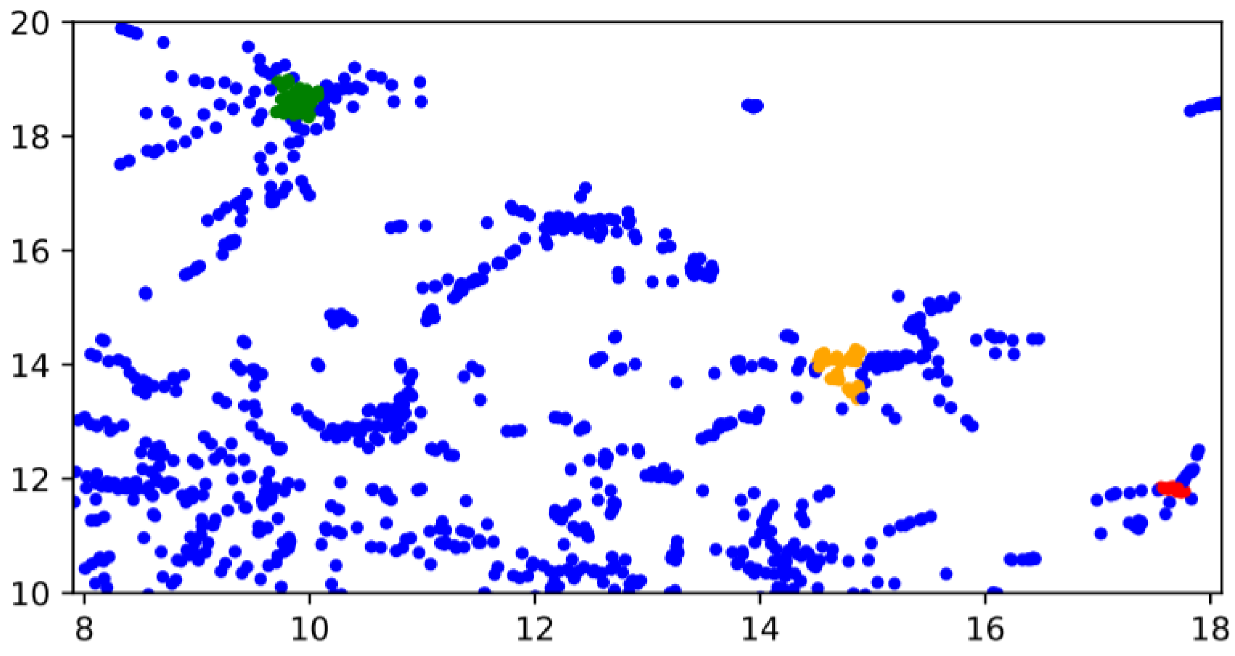


Figure 8.2: An embedding visualization of a partial set of the vocabulary.

8.5 Empirical Experiments

The *goal* of the empirical study was to assess the performance of our proposed approach for commit message generation of composite code changes, with the *purpose* of understanding how its different components work when compared to existing methods. The *perspective* is of both researchers and practitioners: the former are interested in understanding how to generalize existing commit message generators to more complex code changes such as the composite ones; the latter are interested in assessing the feasibility of the proposed approach in practice. More specifically, the empirical experiments revolve around three research questions.

- **RQ$_1$.** *How does the devised strategy based on the addition of change types as a second input work when compared with baseline models?*

- **RQ$_2$.** *How does the devised strategy based on placeholders work when compared with baseline models?*

- **RQ$_3$.** *How does the NMT model perform with BERT embeddings compared with the model without BERT embeddings?*

These three research questions aim at assessing the gain provided by the defined methods when generating commit messages for composite changes. In the next sections, we describe the methodology and results addressing our goals.

8.5.1 Research Methodology

To address our research questions, we proceeded with a mixed-method approach [283], combining quantitative results with qualitative insights coming from developers.

Table 8.2: The list of used baselines.

Baseline	Placeholder Mechanism	Composite changes as a second input	Technique SB vs. DL	Number of encoders	Research Questions
NMT_Two	Yes	Yes	Deep Learning	2	RQ1, RQ2,RQ3
NMT-emptySecondInput	Yes	No	Deep Learning	2	RQ1
NMT_One	Yes	No	Deep Learning	1	RQ1
NNGen+ChangeTypes	Yes	Yes	Search-Based		RQ1, RQ2
NNGen	Yes	No	Search-Based		RQ1
NMT_TwoV	No	Yes	Deep Learning	2	RQ2
NNgen+ChangeTypesV	No	Yes	Search-Based		RQ2

Automatic evaluation. From a quantitative standpoint, we assessed the performance of the three devised methods by computing the well-known BLEU index [284]. This is a metric that is able to establish the quality of text that has been machine-translated from one natural language to another. The general idea behind the metric is to assess how close the machine translation is with respect to a professional human translation. It analyzes n-grams and computes the precision of the machine-translation on blocks of text. The BLEU index scores a translation on a scale between 0 and 1, where 1 indicates a perfect translation in terms of adequacy and fluency of the output. In our work, the number of modified n-grams that the metric computes was equal to 4, as done in previous work [67]: as such, the computed BLEU-4 index measures the average modified 4-gram precision with a penalty for overly short sentences. Table 8.2 describes the baselines used.

In **RQ$_1$**, we included the information on composite changes to both NMT and NNGen models. As such, we compared our method with a series baselines (the first five lines of table

8.2). First, we considered NMT_Two, which is an NMT model similar to our approach but does not take BERT embeddings. Second, we considered the approach proposed by Jiang et al. [67] with our placeholder mechanism (called NMT_One in Table 8.2). This is based on a neural machine translation (NMT) model to generate commit messages from diffs. The source code diffs are considered as the source language and the commit messages as the target language. The model by Jiang et al. consists of two recurrent neural network layers, one for encoder and the other for decoder. They employed a standard Bahdanau attention mechanism [285]. We also considered an NMT with an empty second input. This is the same model as NMT_Two (two encoders), but we did not provide change types in the testing: this baseline allowed us to quantify more precisely the actual relevance of the information on change types. Moreover, we considered the basic NNGen which is a search-based method proposed by Liu et al. [74] as baseline: this is based on the nearest neighbor (NN) algorithm. NNGen [65] takes a new diff and a training set as inputs and outputs a one-sentence commit message for the new diff. It chooses the top-k similar diff files to the new diff file using cosine similarity. Then, it calculates BLEU score to choose the most similar out of the top-k diffs file. Finally, NNGen reuses the same message of the most similar diff from the training set as the new commit message. In our validation, NNGen uses the placeholder mechanism. Additionally, to investigate the importance of the change types and compare our approach with an advanced search-based method, we implemented NNGen+changeTypes. To add change types to the NNGen Model, we adapted the steps presented by Zhongxin et al. [65] to include our new input vector *composite changes* and to make it more specific to run on our dataset.

We used the similarity between change types (e.g. composite change operations) instead of textual similarity to guide the search algorithms. We developed a strategy to assess the similarity of composite changes by considering the types of different operations, the scales of the changes and the involved parameters. Our approach first extracted diffs and composite changes vector from the training set. Next, the training diffs with their training composite changes vector and the new diff with its composite changes vector were represented as vectors in the form of "bags of words" [286]. Then, it calculated the *cosine similarity* between the new diff vector and rest of all the training diff vectors individually; similarly between the new composite changes vector and rest of all the training composite changes vectors individually. The algorithm selected the top `k` training diffs with their composite changes vectors with combined highest similarity scores. After that, the BLEU-4 score between the new diff with its composite changes vector and each of the top-k training diffs with their composite changes vectors were computed. The training commit with the highest BLEU-4 score after combining the results from its training diff and composite changes vector, was regarded as the nearest neighbor of the new submitted commit.

In **RQ**$_2$, we assessed the gain provided by the placeholder mechanism to both the NMT and NNGen models that include the change types information. The baselines used to answer this research question were NMT_Two, NMT_TwoV, NNGen+ChangesTypes, and Nngen+ChangesTypesV. The only difference between NMT_Two and NMT_TwoV is that the latter does not take into consideration the placeholder mechanism. Similarly, NNGen+ChangesTypesV is a variant of NNGen+ChangesTypes that does not include the placeholder mechanism. This can be seen as an iterative evaluation where we quantified how much the performance of our models can further improve by means of the placeholder mechanism.

In **RQ**$_3$ we compared our BERT embeddings method to the NMT_Two model that does not include embeddings.

Manual evaluation. We complemented the automatic evaluation with a survey conducted with practitioners aimed at evaluating the commit messages generated by our models NMT_Two (2 encoders) and NNGen+ChangeTypes. While BLEU computed the textual similarity between the generated and the reference messages, the human study aimed to evaluate the semantic similarity. More particularly, we selected 12 participants for 40 minutes each to evaluate the generated commit messages. The detailed demographics of the participants are available in our replication package; in short, the are all industrial developers with a development experience of more than 5 years. The survey started with a welcome section where we explained the goal of the study and the approximate completion time. The welcome section also had information about the scoring system that would be used in the study. In fact, we used the same scoring system as was provided by Zhongxin et al. [65]. The survey had two sections:

Section I: Demographics. This section presents basic demographic questions to the participants, such as their software development experience. In particular, we asked participants about their current position and total experience in terms of years related to software development, as well as their level of expertise in composite changes/refactoring, software quality assurance, continuous integration, software development, and code review. The collected information helped portray a clear and professional profile of our participants.

Section II: Generated Message Evaluation. This section contained a description of the study and a link to two Excel files. The first contained 101 rows and had six columns which are the commit ID, three columns for commit messages (reference message, NMT_Two (2 encoders message), and NNGen+ChangeTypes message in random order), the applied composite changes, and a column to answer the question "which commit message better describes the changes in composite operations and diff files?" and optionally to justify their choice in an open-ended question. Participants were also given a link where they could find the diff files for every studied commit. The three commit messages were presented anonymously. Thus, participants could not distinguish between reference and generated messages. In the second file, we revealed only which commit message was the reference commit and asked participants to select a score between 0 to 4 for the remaining two messages, where 0 meant that there was no similarity between the message and the reference and 4 meant that the message had the same meaning.

We randomly selected 606 commits from our dataset and divided them evenly into 6 groups. Each group had its specific survey, and it was distributed to 2 different participants.

8.5.2 Analysis of the Results

The results of the study are analyzed in the following.

Automatic evaluation. Table 8.3 describes the BLEU-4 scores for each of the approaches compared in our experiments. From the table, we observe that NMT_Two (2 encoders) has the highest BLEU score with 16.83 BLEU and NNGen has the lowest with 9.87 BLEU.

Table 8.3: Method comparison in terms of BlEU-4 scores.

Dataset	Approach	BLEU-4
With placeholders	NMT_Two (2 encoders)	**16.83%**
	NMT-emptySecondInput (2 encoders)	13.04%
	NMT_One (1 encoder)	11.72%
	NNGen+ChangeTypes	**12.79%**
	NNGen	9.87%
	NMT_BERT (2 encoders)	**18.91%**
Without placeholders	NMT_TwoV (2 encoders)	**14.07%**
	NNGen+ChangeTypesV	**10.75**

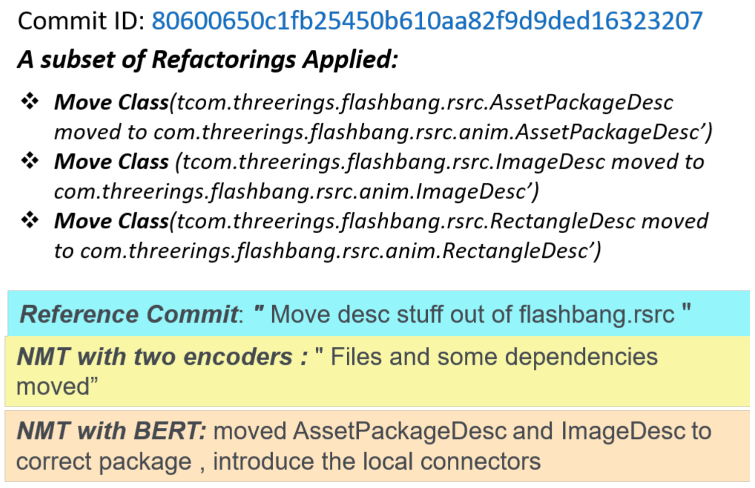
Table 8.3 clearly shows that both NMT_Two (2 encoders) (16.83 BLEU) and NNGen+ChangeTypes (12.79) have significantly higher BLEU scores than the considered baselines—NMT_One (1 encoder) (11.72 BLEU) and NNGen (9.87 BLEU). Additionally, NMT_Two (2 encoders) outperforms NNGen+ChangeTypes by almost 4 full BLEU points. This significant difference in performance can be explained by the nature of composite changes and the type of commits that we have in our dataset. In fact, we manually looked into a random subset of our dataset (~400 commits) and confirmed that it is rare to find similar diffs in our dataset. While an information retrieval can be useful to some extent in documenting code changes, it cannot be used to generate commit messages for either composite changes or partially new code changes.

One major observation is that NMT_Two (2 encoders) outperforms all the other different techniques. One important question to be addressed at this stage is "Will NMT_Two (2 encoders) also work on commits when change types are not available?" To answer this question, we tested NMT_Two (2 encoders) without change types as inputs. In other words, the first encoder's input is set to empty. The BLEU score of this test, NMT-emptySecondInput (2 encoders), is 13.04%. The reported BLEU score clearly shows that NMT_Two (2 encoders) works better than the baselines even though there are additional inputs of change types. Furthermore, the deterioration of the scores highlighted in red in Table 8.3 sheds light on the importance of our placeholder mechanism. For NMT_Two (2 encoders) and NNGen+ChangeTypes, BLEU scores were decreased by more than two points when the placeholder mechanism was not used.

Turning attention to the performance of the BERT embeddings, Table 8.3 reports the BLEU-4 score for NMT_Two (2 encoders) combined with BERT embeddings. The BLEU score for this enhanced approach is 18.91% which represents an improvement of two full BLEU point over our model NMT_Two (2 encoders) that uses the default embeddings. This outperformance shows that pretraining a BERT model and using its embeddings in the model to generate commit messages helps NMT to better capture the changes and thus improve the overall model performance.

An example of the commit messages generated by NMT_Two, NMT-BERT and the reference message is represented in Figure 8.3. This commit[4] has 7 changed files, with 5 additions and 12 deletions. A subset of the applied composite changes is represented in Figure 8.3. The commit generated by NMT-BERT captures specifically the composite changes "moving classes" better than NMT_Two (2 encoders). The generated message by NMT-BERT has clearer information about the changes made in the commit than the reference message and the message generated by NMT_Two.

[4]https://github.com/sudheer307/flashbang-playn,
commit:80600650c1fb25450b610aa82f9d9ded16323207


Commit ID: 80600650c1fb25450b610aa82f9d9ded16323207

A subset of Refactorings Applied:

- ***Move Class****(tcom.threerings.flashbang.rsrc.AssetPackageDesc moved to com.threerings.flashbang.rsrc.anim.AssetPackageDesc')*
- ***Move Class*** *(tcom.threerings.flashbang.rsrc.ImageDesc moved to com.threerings.flashbang.rsrc.anim.ImageDesc')*
- ***Move Class****(tcom.threerings.flashbang.rsrc.RectangleDesc moved to com.threerings.flashbang.rsrc.anim.RectangleDesc')*

Reference Commit: **"** Move desc stuff out of flashbang.rsrc **"**

NMT with two encoders : " Files and some dependencies moved"

NMT with BERT: moved AssetPackageDesc and ImageDesc to correct package , introduce the local connectors

Figure 8.3: A commit from flashbang-playn github repository and its three associated commit messages.

Manual evaluation We obtained 1212 pairs of responses from our manual evaluation. Each pair contained a choice for the message that better describes the composite changes, a score for the message generated by NMT_Two (2 encoders) and a score for a message generated by the NNGen+ChangeTypes model. We found a low disagreement rate between the two evaluators with a Cohen's Kappa coefficient [287] equal to 0.95 for the scoring of the messages and a Cohen's Kappa coefficient of 0.92 for choosing the best commit message. Both Cohen's Kappa statistics showed that there was a considerable agreement between different evaluators. For each group, the two assigned evaluators were able to contact each other to look into the disagreements and decided the final score/choice. Following the same scoring scale described by Zhongxin et al. [65], we categorized the generated messages into three categories: low-quality (score of 0 and 1), a score of 2 as medium-quality, and a score of 3 and 4 as high-quality. Table 8.4 presents the results. We can see that the proportion of high-quality of NMT_Two (2 encoders) messages is significantly higher than that of NNGen+ChangeTypes commit messages. While the percentage of the medium-quality messages for NMT_Two (2 encoders) is 17%, NNGen has a comparable proportion of 15.22%.

As for the question of which commit message better describes the changes, more than one-third of the time (34.32%) the developer chose the message generated by our model

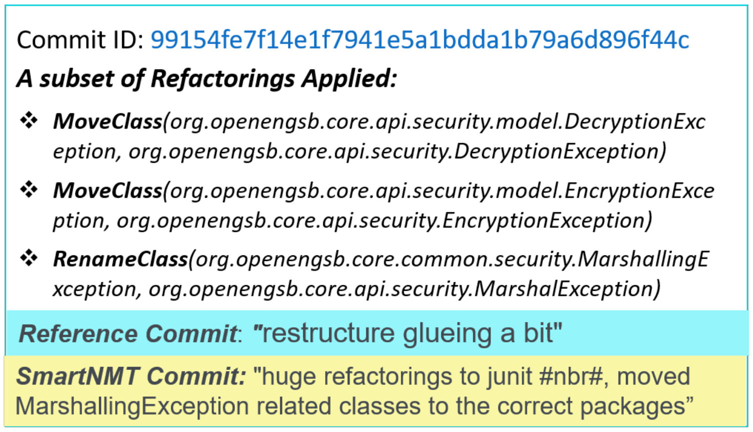
Table 8.4: The results of our user study.

Approach	Low	Medium	High
NMT (2 encoders)	**64%**	**17%**	**19%**
NNGen+ChangeTypes	79%	15.22%	5.78%

NMT_Two (2 encoders) as the message that better described the changes. Only 14% of the developer's choices went to the messages generated by NNGen+ChangeTypes. Comparing the effectiveness of NMT_Two (2 encoders) and NNGen+ChangeTypes in generating composite changes-related commit messages, we found that the deep learning model is more than *twice* as effective as the information retrieval approach—this is consistent with what was discovered in our automatic evaluation.

We were also interested in understanding the reasons and the situations where the evaluator chose the reference message over the generated messages. After looking at all the survey results, we found that 20% of the time the reference message and the message generated by NMT_Two (2 encoders) were "similar in meaning but have different information" (score 3). This means that both of them were describing the changes in different ways and that each message had different information to include. The developer left a comment when they found it hard to choose between them, saying *"Both Message#2 and Message#3 look good as a commit message and they can be complementary"*. Figure 8.5 shows one such example, where the evaluator chose the first commit but both were similar and seemed to serve as a good message for the changes. This observation may put an emphasis on the usefulness of building a tool not only to generate a completely new commit message but also to check for the completeness of the documentation, thus assisting the developer in writing a more comprehensive description for commits with composite code changes.

To get a better explanation for the relatively high percentage of low-quality commit messages (Table 8.4) for NMT_Two (2 encoders) while they were frequently chosen as the best commits to describe the changes, we manually examined some specific examples where the developers chose them as the best commit message but assigned them a low score. In fact,


Commit ID: 99154fe7f14e1f7941e5a1bdda1b79a6d896f44c

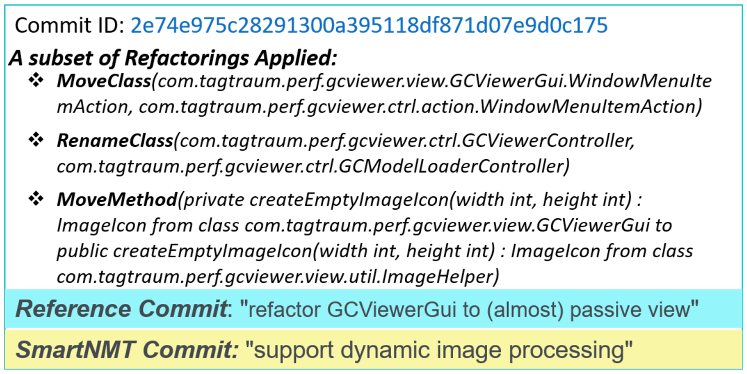
A subset of Refactorings Applied:

- ***MoveClass****(org.openengsb.core.api.security.model.DecryptionException, org.openengsb.core.api.security.DecryptionException)*
- ***MoveClass****(org.openengsb.core.api.security.model.EncryptionException, org.openengsb.core.api.security.EncryptionException)*
- ***RenameClass****(org.openengsb.core.common.security.MarshallingException, org.openengsb.core.api.security.MarshalException)*

Reference Commit: "restructure glueing a bit"

SmartNMT Commit: "huge refactorings to junit #nbr#, moved MarshallingException related classes to the correct packages"

Figure 8.4: An example where NMT (2 encoders) performs better than the reference message.


Commit ID: 2e74e975c28291300a395118df871d07e9d0c175

A subset of Refactorings Applied:

- ***MoveClass****(com.tagtraum.perf.gcviewer.view.GCViewerGui.WindowMenuItemAction, com.tagtraum.perf.gcviewer.ctrl.action.WindowMenuItemAction)*
- ***RenameClass****(com.tagtraum.perf.gcviewer.ctrl.GCViewerController, com.tagtraum.perf.gcviewer.ctrl.GCModelLoaderController)*
- ***MoveMethod****(private createEmptyImageIcon(width int, height int) : ImageIcon from class com.tagtraum.perf.gcviewer.view.GCViewerGui to public createEmptyImageIcon(width int, height int) : ImageIcon from class com.tagtraum.perf.gcviewer.view.util.ImageHelper)*

Reference Commit: "refactor GCViewerGui to (almost) passive view"

SmartNMT Commit: "support dynamic image processing"

Figure 8.5: An example of good messages deemed complementary by the evaluators.

the scoring system relies only on the similarity of the commit messages, with the assumption that the reference message is of a high-quality. However, our manual examination revealed that the reference messages can be poorly-written, which aligns with the findings of Cortes et al. [76]. Therefore, the generated commit message can be in fact of a higher quality, while not being similar to the reference message. Figure 8.4 shows one of these examples (more examples in our appendix [259]).

Additional manual analysis. In an effort to provide further findings into the usefulness of change types for commit message generation, we manually investigated the types of composite change operations that were more useful in our dataset for the generation. This analysis aims to explain the rationale behind the performance improvement of the model NMT_Two (2 encoders) when adding the change types as an additional input. To investigate the impact of different change types such as composite change operations, we performed correlation analysis between the quality of the commit messages and the frequency of the different applied composite change operations of the specific commit.

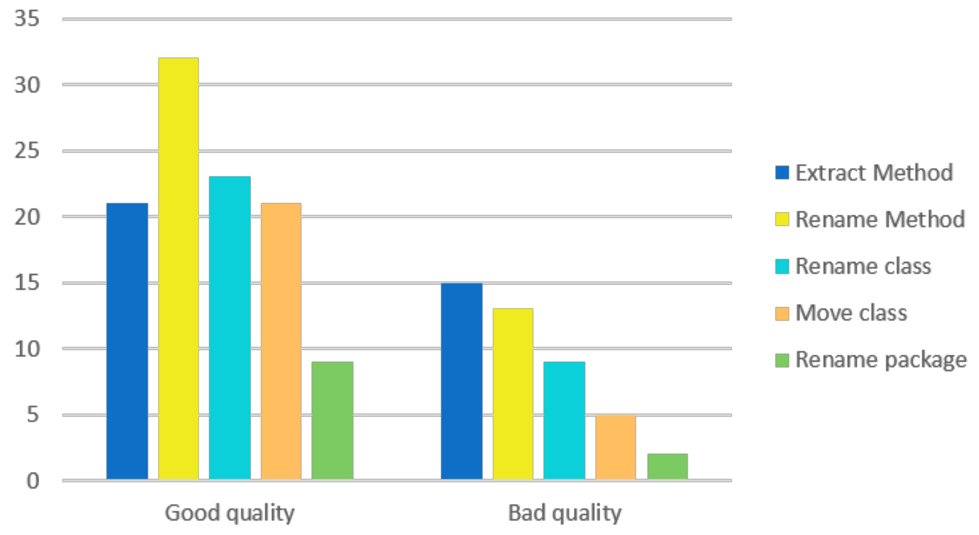


Figure 8.6: Distribution of frequent Composite change types.

The five most frequent composite change operations in our dataset are: *Extract Method, Rename Method, Rename Class, Move Class, and Rename Package.* Figure 8.6 shows the distribution of these five composite change operations in bad and good quality commit messages. *"Rename Method"* has the highest impact on the generation of good quality commits followed by *"Rename Class"*, and then *"Move Class"* and *"Extract Method"*. On the other hand, *"Extract Method"* has the highest impact on the generation of bad quality commits. For both quality levels (bad and good), we could see that *"Rename Package"* has the lowest impact on the model performance, out of the top five change types. One most probable reason for this low impact is that "rename package" changes are not well documented or are not well captured from diffs, and thus their contributions to the learning process are not significant.

Table 8.5 shows the correlation table of the most frequent change types in good-quality commit messages. All five composite change operations are strongly correlated (correlation coefficient above 0.92). This implies that the model generates good messages when there

Table 8.5: Good quality generated commit messages.

	ExtractMethod	RenameMethod	RenameClass	MoveClass	RenamePackage
ExtractMethod		0.97	0.95	0.95	0.92
RenameMethod			0.96	0.96	0.93
RenameClass				0.96	0.92
MoveClass					0.94
RenamePackage					

are more correlated change types. In other words, if a commit has many change types that are somehow related to each other (e.g., changing the same code fragment or improving the same quality aspect), the better the performance of the model would be.

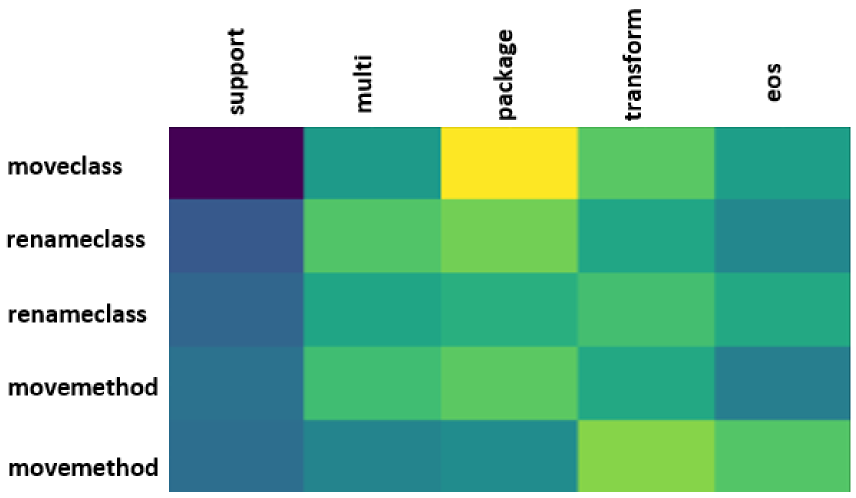


Figure 8.7: Heatmap of the attention layer in NMT (2 encoders) for the Change Types input for a good commit.

To further understand how composite change operations impact model performance, we performed attention visualization [288, 289] for a good example of where composite changes input gets the attention and plays an important role in generating the commit message, as shown in Figure 8.7. The x-axis in Figure 8.7 is the commit message input and the y-axis is the change types input vector (applied composite changes). High activation (more yellow) indicates more attention paid. For instance, we can notice in Figure 8.7 that the model gave attention to "package" and associated it with the "move class" change type. This can be explained by the fact that the model learned to give attention to "package" when there was a "move class" since classes are usually moved within packages.

8.6 Threats to Validity

One potential threat to validity could be related to our dataset and the quality of the collected commit messages. We have carefully extracted actual human-written commit messages from 7,492 GitHub repositories, and used several cleaning, standardizing, and filtering steps, including the extraction of the structure of the commits, and then selected a good set of relatively high-quality composite changes-related commit messages. Another threat to validity is about the limited number of evaluators participating in our human evaluation. We cannot guarantee that the choice of a commit message or its assigned score is fair. However, we tried to mitigate the human bias by selecting as many professional participants as possible. We also assigned two evaluators for each set of 101 commit messages, and the disagreements were rare (Cohen's Kappa score of 0.95).

Finally, another threat is related to the implementation. NMT (1 encoder) and NMT (2 encoders) have different training setups and model parameters. The former uses the setup that Jiang et al. [67] have reported. Thus, the comparison of the results may be affected. However, both models are seq-2-seq models with almost the same architecture. The data preparation is the same for both models.

8.7 Conclusion

We have shown how NMT techniques can be enhanced to address the problem of commit messages generation for composite code changes. Our NMT-BERT model outperforms the state-of-the art techniques, shedding light on the importance of working with a pretrained model as the source for embeddings. Diff files can be very lengthy, which remains a problem that we are planning to address in the future. Lastly, we consider this presented work as a step forward, suggesting solutions and insights on how to advance the state-of-the-art techniques for commit messages generation.

CHAPTER IX

Multi-Objective Code Reviewer Recommendations: Balancing Expertise, Availability and Collaborations

9.1 Introduction

The source code review process has always been one of the most important software maintenance and evolution activities [70]. Several studies show that a careful code inspection can significantly reduce defects and improve the quality of software systems. Recently this process has become informal, asynchronous, light-weight and facilitated by tools [71] [72]. A survey with practitioners, performed by Bacchelli et al. [73], show that code review nowadays is expanding beyond just looking for defects but to also provide alternatives to improve the code and transfer knowledge among developers.

Despite recent progress [79, 80] code reviews are still time-consuming, expensive, and complex involving a large amount of effort by managers, developers and reviewers. Thongtanunam et al. [81] found on four open source projects with 12 days as the average to approve a code change. The automated recommendation of peer code reviewers may help to reduce delays by finding the best reviewers who will then spend less time in reviewing the assigned files.

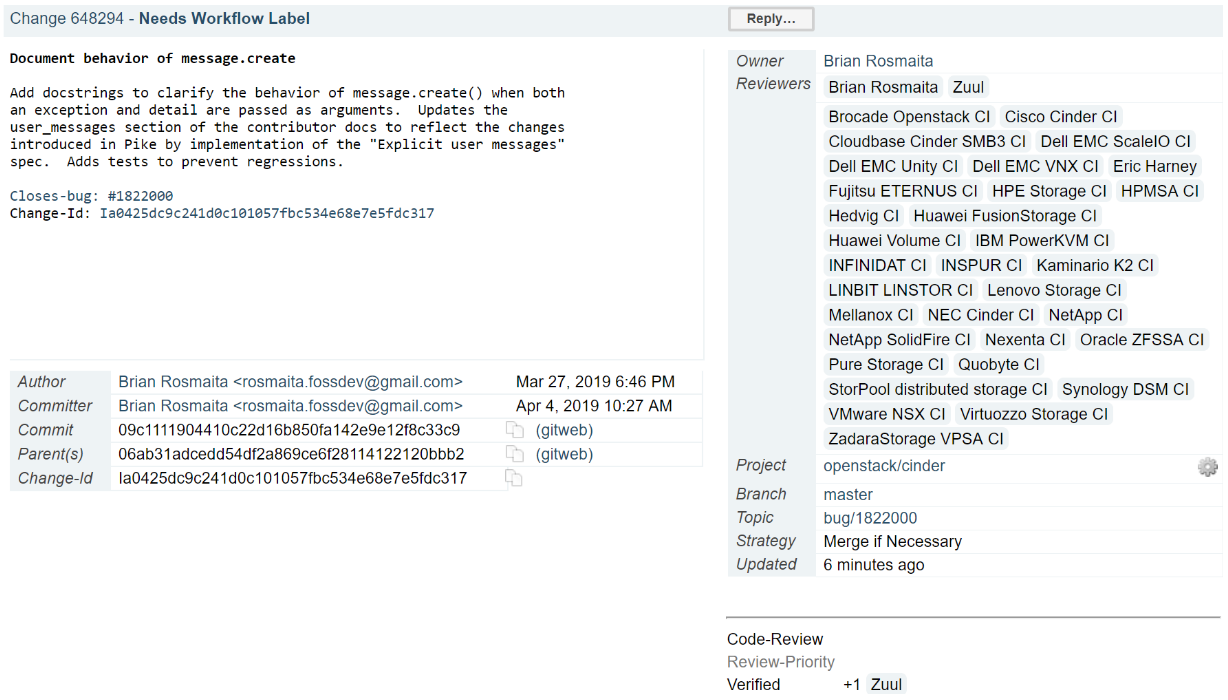
The majority of existing tools and techniques for automated recommendation of code reviewers are based on the level of reviewer expertise [80, 71, 142, 81]. Expertise is mainly defined as the prior knowledge of the changes under review. For instance, a selected peer reviewer with high expertise should have reviewed the same files [81, 142], or even the same lines of code in the files [71]. An empirical study at Microsoft found that selected reviewers with high expertise can provide valuable and rapid feedback to the author of the code under review [73]. However, reviewers with high expertise may not be always available in practice, or at least assigning them may create delays.

To address the above challenges we propose to formulate the selection of peer code reviewers as a multi-objective problem. The goal is to balance the conflicting objectives of expertise, availability and history of collaborations. The multi-objective approach tries to find a trade-off between multiple objectives and minimizing the former collaborations on reviewing the same files is just one component between many objectives. We adopted one of the widely used multi-objective search algorithms, NSGA-II [231], to find a trade-off depending on current context and available resources. For instance, our formulation can slightly sacrifice expertise to avoid a delay caused by limited resources (e.g. low availability of peer reviewers). In another context, the reviewer(s) with the highest expertise can be selected when the goal is to inspect high priority code changes such as critical buggy files. Thus, our approach enables navigation between the three different dimensions by generating multiple non-dominated peer reviewer recommendations instead of one solution as is done in existing work.

Our validation on 9 open source confirms the effectiveness of our multi-objective approach by making better recommendations than the state of the art.

9.2 Preliminary Study

As part of preliminary work of this contribution, we performed unstructured survey with 6 senior managers and 11 senior developers actively involved in code reviews to assign reviewers or/and review pull-requests. We decided to perform an unstructured survey to encourage the participants to think-aloud and avoid biasing them with our opinions. Furthermore, the goal of our surveys is get insights about the current challenges in code reviews rather than a large



```
Document behavior of message.create

Add docstrings to clarify the behavior of message.create() when both
an exception and detail are passed as arguments.  Updates the
user_messages section of the contributor docs to reflect the changes
introduced in Pike by implementation of the "Explicit user messages"
spec.  Adds tests to prevent regressions.

Closes-bug: #1822000
Change-Id: Ia0425dc9c241d0c101057fbc534e68e7e5fdc317
```

Author	Brian Rosmaita <rosmaita.fossdev@gmail.com>	Mar 27, 2019 6:46 PM
Committer	Brian Rosmaita <rosmaita.fossdev@gmail.com>	Apr 4, 2019 10:27 AM
Commit	09c1111904410c22d16b850fa142e9e12f8c33c9	(gitweb)
Parent(s)	06ab31adcedd54df2a869ce6f28114122120bbb2	(gitweb)
Change-Id	Ia0425dc9c241d0c101057fbc534e68e7e5fdc317	

Owner	Brian Rosmaita
Reviewers	Brian Rosmaita, Zuul, Brocade Openstack CI, Cisco Cinder CI, Cloudbase Cinder SMB3 CI, Dell EMC ScaleIO CI, Dell EMC Unity CI, Dell EMC VNX CI, Eric Harney, Fujitsu ETERNUS CI, HPE Storage CI, HPMSA CI, Hedvig CI, Huawei FusionStorage CI, Huawei Volume CI, IBM PowerKVM CI, INFINIDAT CI, INSPUR CI, Kaminario K2 CI, LINBIT LINSTOR CI, Lenovo Storage CI, Mellanox CI, NEC Cinder CI, NetApp CI, NetApp SolidFire CI, Nexenta CI, Oracle ZFSSA CI, Pure Storage CI, Quobyte CI, StorPool distributed storage CI, Synology DSM CI, VMware NSX CI, Virtuozzo Storage CI, ZadaraStorage VPSA CI
Project	openstack/cinder
Branch	master
Topic	bug/1822000
Strategy	Merge if Necessary
Updated	6 minutes ago

Code-Review		
Review-Priority		
Verified	+1	Zuul

Figure 9.1: An example of a code review extracted from OpenStack

empirical study. We found that 10 days is the average to approve a code change at eBay. The main reason based on the surveys for the delay is the challenging task of identifying the right reviewers which is aligned with existing studies[290, 291].

A senior manager confirmed that ”We don’t actually need more tools to just suggest reviewers based on expertise. We need better support to manage code reviews especially with short deadlines and limited resources while not sacrificing a lot of expertise. It is a complex problem.” In addition, the participants highlighted that it is critical to consider the priority of the files to be inspected as part of the management of the code review process. Furthermore, we found in our interviews that the social interactions between code authors and reviewers is another critical aspect to consider to ensure high quality reviews.

Existing studies assume that peer reviewers with high interactions with authors/owners of the code under review are the best to select [79]. However, this aspect may be considered negative with extensive mutual peer reviews and/or quick approval of code changes as suggested by the participants. The diversity of peer code reviews is important, as pointed out by the eBay senior managers and peer reviewers, especially when frequent patterns of code authors/reviewers are observed.

9.3 Approach

In this section, we describe our proposed approach for recommending the most appropriate set of reviewers for pull-requests to be reviewed using multi-objective search.

9.3.1 Multi-Objective Optimization

Multi-Objective search considers more than one objective function to be optimized simultaneously. It is hard to find an optimal solution that solves such problems because the objectives to be optimized are conflicting. For this reason, a multi-objective search-based algorithm could be suitable to solve this problem because it finds a set of alternative solutions, rather than a single solution as result. One of the widely used multi-objective search techniques is NSGA-II [231, 292, 293] that has shown good performance in solving several software engineering problems [183].

A high-level view of NSGA-II is depicted in Algorithm 5. The algorithm starts by randomly creating an initial population P_0 of individuals encoded using a specific representation (line 1). Then, a child population Q_0 is generated from the population of parents P_0 (line 2) using genetic operators (crossover and mutation). Both populations are merged into an initial population R_0 of size N (line 5). *Fast-non-dominated-sort* [231] is the technique used by NSGA-II to classify individual solutions into different dominance levels (line 6). Indeed, the concept of non-dominance consists of comparing each solution x with every other solution in the population until it is dominated (or not) by one of them. According to Pareto optimality: “A solution x_1 is said to dominate another solution x_2, if x_1 is no worse than x_2 in all objectives and x_1 is strictly better than x_2 in at least one objective”. Formally, if we consider a set of objectives f_i , $i \in 1..n$, to maximize, a solution x_1 dominates x_2 :

$$\textit{iff}\ \forall i,\ f_i(x_2) \leqslant f_i(x_1) \text{ and } \exists j \mid f_j(x_2) < f_j(x_1)$$

Algorithm 5 High level pseudo code for NSGA-II

```
1: Create an initial population $P_0$
2: Create an offspring population $Q_0$
3: $t = 0$
4: while stopping criteria not reached do
5:     $R_t = P_t \cup Q_t$
6:     F = fast-non-dominated-sort($R_t$)
7:     $P_{t+1} = \emptyset \ and \ i = 1$
8:     while $\mid P_{t+1} \mid + \mid F_i \mid \leqslant N$ do
9:         Apply crowding-distance-assignment($F_i$)
10:        $P_{t+1} = P_{t+1} \cup F_i$
11:        $i = i + 1$
12:    end while
13:    $Sort(F_i, \prec n)$
14:    $P_{t+1} = P_{t+1} \cup F_i[N- \mid P_{t+1} \mid]$
15:    $Q_{t+1}$ = create-new-pop($P_{t+1}$)
16:    t = t+1
17: end while
```

The whole population that contains N individuals (solutions) is sorted using the dominance principle into several fronts (line 6). Solutions on the first Pareto-front F_0 get assigned dominance level of 0. Then, after taking these solutions out, *fast-non-dominated-sort* calculates the Pareto-front F_1 of the remaining population; solutions on this second front get assigned dominance level of 1, and so on. The dominance level becomes the basis of selection of individual solutions for the next generation. Fronts are added successively until the parent population P_{t+1} is filled with N solutions (line 8). When NSGA-II has to cut off a front F_i and select a subset of individual solutions with the same dominance level, it relies on the crowding distance [231] to make the selection (line 9). This parameter is used to promote diversity within the population. This front F_i to be split, is sorted in descending order (line 13), and the first (N- $|P_{t+1}|$) elements of F_i are chosen (line 14). Then a new population Q_{t+1} is created using selection, crossover and mutation (line 15). This process will be repeated until reaching the last iteration according to stop criteria (line 4).

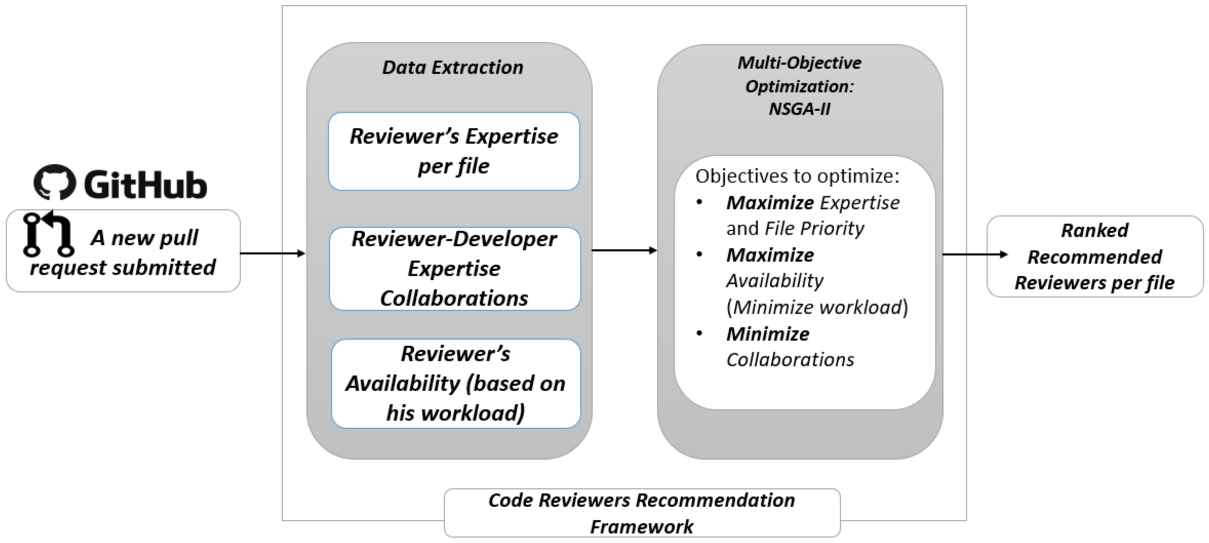

GitHub
A new pull request submitted
Data Extraction
Reviewer's Expertise per file
Reviewer-Developer Expertise Collaborations
Reviewer's Availability (based on his workload)
Multi-Objective Optimization: NSGA-II
Objectives to optimize:
Maximize Expertise and File Priority
Maximize Availability (Minimize workload)
Minimize Collaborations
Ranked Recommended Reviewers per file
Code Reviewers Recommendation Framework

Figure 9.2: Overview of our multi-objective search-based approach for code reviewer recommendation.

9.3.2 Approach Overview: A Multi-objective Code Reviewer Recommendation Framework

The ultimate goal of our Code Reviewer Recommendation framework is to automatically assign the most appropriate reviewers to newly opened pull-requests. The assignment is performed by balancing three important competing criteria: the expertise of the reviewers, their availability (considering their current workload) and their social connections (collaborations) with the submitter of the open pull request(s). Thus, we propose to use multi-objective search, based on NSGA-II [231], to find a tradeoff between the different competing objectives. An overview of the approach is illustrated in Figure 9.2.

Our approach takes as input: 1) the pull-request(s) to be reviewed; 2) the pull-request(s) under review and the involved reviewers; and 3) the detailed history of closed pull-requests. The extraction of these 3 required inputs is easy and straightforward by simply providing the GitHub link of the project to our tool. Using our integrated parser, we automatically analyze the GitHub repository to collect the code review history, commit messages and source code. Next, from the collected data, we extract three clusters of interaction information: a File-Reviewer interaction matrix (FR), a Developer-Reviewer interaction matrix (DR) and a File-Developer interaction matrix (FD). From the open pull-requests to be reviewed we can automatically extract the files that need to be reviewed and evaluate the expertise of assigned reviewers in our solution representation, as detailed later.

As an output, our multi-objective algorithm generates a set of trade-off solutions where each solution consists of assigning one or more reviewers per pull-request. Thus, the solution can be represented as a matrix matching reviewers to the files of the pull-request(s). For each file, the reviewers are ranked based on their level of expertise to review the file, their availability, and their past collaboration with the developer of that file, all while reducing the number of reviewers per pull-request as much as possible.

To find a trade-off between the different objectives, we used NSGA-II [231] since it was used for similar discrete problems in software engineering and performed well. The use of a metaheuristic algorithm to deal with conflicting objectives is justified by the large search space to explore. Let M be the number of total reviewers and P number of total files submitted to be reviewed for code changes. The size of the search space to explore in order to find the best subset of m reviewers among a set of M reviewers to review p files is of $\begin{pmatrix} m \\ M \end{pmatrix} \times p = \frac{m!}{m!(M-m)!} \times p$. This is a very fast growing function and as M grows the search space becomes prohibitively large to the point where exhaustive search is not practical. We propose the use of metaheuristic search to explore this combinatorial search space to find near-optimum reviewer recommendations.

The multi-objective approach proposed in this research work generates as output a set of non-dominated solutions (Pareto front). It is up to the team manager to select the reviewers assignment solution based on their preferences. Thus, the final output of the algorithm is a set of solutions (Pareto front) representing trade-offs between the three objectives. It is up to the manager to select the reviewers assignment (choose a solution) based on their preferences. In general, the preferences are defined based on the current context: urgency to release code quickly, available resources, speedy growth phase of the project, etc. These different contexts are not changing daily and they are not related to only one or few pull-requests but more related to the situation of the whole project. The preferred solution can be quickly selected by looking at the distribution of the solutions in the Pareto front or ranking the solutions based on the most preferred fitness function based on the current context. The two common ways to extract a solution from the Pareto front are the use of the reference point and the knee point [294, 295, 296, 297]. The knee point corresponds to the solution with the maximal trade-off between all fitness functions, i.e., a vector of the best objective values for all solutions. In order to find the maximal trade-off, we use the trade-off worthiness metric proposed by [297] to evaluate the worthiness of each solution in terms of objective value compromise. While the knee point selection may not be the perfect way, it is the only strategy to ensure a fair comparison with the mono-objective and deterministic approaches since they generate only one solution as output.

The manager may select a reference point with high expertise, if s(he) cares about finding knowledgeable reviewers of the files while accepting some delays in the review process. Thus, the selected solution will be the closest one to the specified reference point. This scenario happens, for example, when a pull-request is modifying some security critical files. However, it is not required that the managers specify the reference point for each pull-request since the preferences usually depend on the context of the whole project and they do not change daily. Moreover, the knee point can be automatically calculated based on the distribution of the solutions in the Pareto front [294] and it represents the maximum trade-off between the objectives.

9.3.3 Main Components of the Approach

9.3.3.1 Reviewer's Expertise Model

This model aims at exploring reviewer-file connections: Who are the peer reviewers who worked on the same file? From the previous commits and closed pull-requests, we can automatically extract a matrix that represents the expertise of reviewers. Expertise value is defined as the number of times that the reviewer reviewed the same file. In fact, for every file, the matrix keeps track of reviewers who reviewed that specific file and how many times every reviewer reviewed that particular file.

FR is a $P \times M$ matrix where each entry $fr_{k,i}$ represents the number of times reviewer r_i reviewed or modified file f_k where $i \in \{1, 2, \ldots, M\}$, $k \in \{1, 2, \ldots, P\}$, P is total number of files requested to be reviewed and M total number of reviewers working on the project. This matrix represents how familiar is each reviewer with each file, which is used as a proxy measure for expertise.

$$FR = (fr_{(k,i)})\varepsilon^{P \times M} \tag{9.1}$$

9.3.3.2 Reviewer-Developer Collaboration Model

To take the socio-technical factor into account when searching for the best reviewers to review a code change, we extracted the collaborations between reviewers and developers from the history of closed pull-requests. In fact, for every potential recommended reviewer, we extract both the list of developers and the files per pull-request that he/she reviewed or modified in the past. Then, we calculated for each pair (reviewer,developer) the total number of commonly modified files. Note that the reviewer can be found in the comments of the pull-requests of the submitter (developer). Thus, a "Collaborations" matrix DR is automatically created.

To sum up, DR is a $N \times M$ matrix where each entry $dr_{j,i}$ represents the number of times reviewer r_i reviewed a file changed by developer d_j where $i \in \{1, 2, \ldots, M\}$, $j \in \{1, 2, \ldots, N\}$, N is total number of developers working on the project and M total number of reviewers working on the project. In fact, $dr_{j,i}$ is defined as the number of files that the reviewer and the developer collaborated together (reviewed or modified) in the past. This matrix represents the social connections between reviewers and developers.

$$DR = (dr_{(j,i)})\varepsilon^{N\times M} \tag{9.2}$$

9.3.3.3 Availability Model

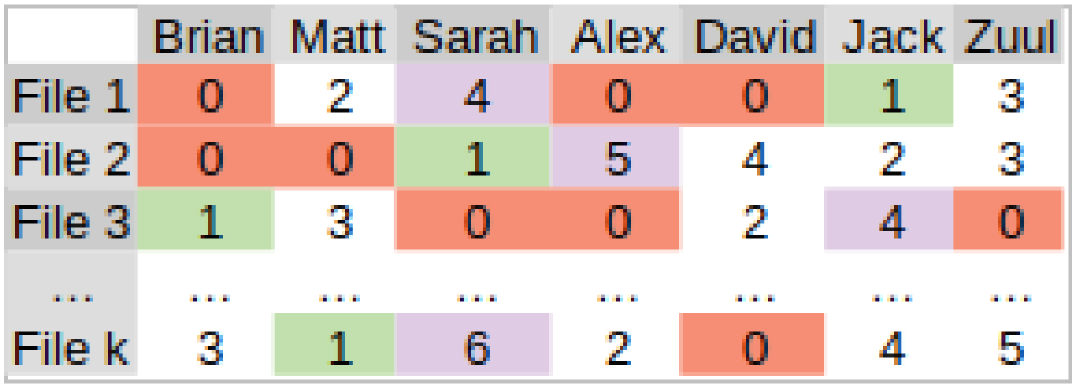
To estimate the availability of peer reviewers, we considered of the number of files per open pull-requests and numbers of commits where they are currently involved. We represented the availability (workload) in a vector $A = [a_1, a_2, \dots, a_M]$ where a_i represents the total number of files of open pull requests and commits for a reviewer r_i.

Data. For *expertise* and *collaborations*, we considered all the data since the start of the project because we believe that more information about the expertise and collaborations of the developers is useful in assigning the appropriate reviewer. Regarding the *availability model*, we considered the last 7 days of open pull requests because we wanted to have an estimate of the current workload of the reviewers.

9.3.4 Problem Formulation

9.3.4.1 Solution Representation

The solution of the optimization problem is a matrix S that contains an integer value $o \in \{0, 1, 2, \dots, M\}$ for entry $s_{k,i}$ denoting the recommended order (rank) for the reviewer r_i to review file f_k. This matrix contains P rows and M columns. P is the number of files that contains code changes to be reviewed and M is the number of potential reviewers. To initialize the matrix S, we first extract the number M because it represents the number of candidate reviewers for the files to be reviewed in the submitted pull-request. Second, we extract the files to be reviewed in the pull-request to review. Then, initially, each S[k,i] will take a distinct random number. Assigning 0 to S[k,i] means that the kth developer is not assigned to review the ith file and assigning an integer 0¡o¡=M means that the developer is assigned to review the ith changed file and his rank is o within the list of appropriate reviewers.



	Brian	Matt	Sarah	Alex	David	Jack	Zuul
File 1	0	2	4	0	0	1	3
File 2	0	0	1	5	4	2	3
File 3	1	3	0	0	2	4	0
...	...	...	...	...	...	...	...
File k	3	1	6	2	0	4	5

Figure 9.3: An example of our solution representation. Red: this reviewer is not recommended to review the file; green: the most appropriate reviewer for the file; and purple: recommended, but the least appropriate reviewer for the file.

After each iteration, the genetic algorithm decides if a reviewer is suitable for a review assignment for a specific file or not. If yes, it will decide the rank of that reviewer, compared to other candidate reviewers for the same file, based on our three objectives (defined in the section 9.3.4.2).

An example of a two-dimensional solution representation is illustrated in Figure 9.3. Let say we have seven reviewers who are working on the project: Brian, Matt, John, Alex, David, Jack and Zuul, and there are k files with code changes. Based on our solution representation, we suggest which reviewers are appropriate for reviewing which file(s) and in what order. In this example, Brian is not recommended to review $file1$ and $file2$, but he is the most appropriate reviewer to review the changes in $filek$. To review $file1$, Matt is the second best reviewer and Zuul is the third best one. To sum up, our multi-objective algorithm outputs reviewer-file matrix (as shown in Figure 9.3) which assigns reviewers to all the files changed in the submitted pull-request. Thus, for each pull-request (PR) we rank the reviewers based on how many files in that PR he/she is able to review taking into consideration the different fitness functions.

9.3.4.2 Fitness Functions

In our approach, we aim to optimize three fitness functions. The first and the second ones are formulated to maximize the *expertise* and *the availability* of the reviewers. While the third fitness function is formulated to minimize the social connections between reviewers and developers in the hope of reducing human bias. The motivation of our multi-objective approach is aligned with the observation of a recent study at Microsoft [298] highlighting that promoting diversity depends on the norms of the team, i.e., some teams prefer diverse, some teams prefer close connections. While previous collaborations between developers and reviewers could reduce the tension around the review task, the extensive former interactions/collaborations can be an indication of light/weak review to approve code quickly to meet release deadlines especially when associated with low expertise. The multi-objective approach proposed in this research work generates as an output a set of non-dominated solutions (Pareto front). It is up to the team manager to select the reviewers assignment solution based on their preferences. If the team prefers close connection then the selected/preferred solution from the Pareto front will be in the region of interest where the objective of collaborations is high otherwise the selected solution will be in the area of the Pareto front where the value of collaboration is low. Our goal is to provide a diverse set of good reviewers assignment solutions rather than only one solution then the user can select the preferred one based on his/her preferences.

We present in the following our three fitness functions: availability, expertise and collaborations.

Availability. The availability is the inverse of the estimated wait until reviewers that are selected to work on a selected set of file S become available. In our case, the waiting period is deducted from the workload that the reviewer has. We considered the workload as the combination of the number of commits submitted recently (during the last 7 days) and the total number of files for all open pull-requests.

$$Availability = \frac{1}{\sum_{k=1}^{P} \sum_{i=1}^{M} a_i * S[k,i]}, s_{k,i} > 0 \tag{9.3}$$

where $a = \{a_1, a_2, \ldots, a_M\}$ is an array that contains the tasks queued for a reviewer. a_i represents the number of tasks in the queue for the reviewer r_i. P is total number of files requested to be reviewed and M is total number of reviewers working on the project.

Expertise Considering File Priority. PR is a vector of weights that defines how urgently a file needs to be reviewed. For a file f_k, the priority score will take 1 if the tag "priority" is used in the pull-request, otherwise, the priority will be 0. We used both FR and PR to formulate the reviewer expertise as an objective.

$$Expertise = \sum_{k=1}^{P} \sum_{i=1}^{M} \frac{FR[k,i] + PR[k]}{S[k,i]}, s_{k,i} > 0 \tag{9.4}$$

where M is total number of reviewers working on the project and P is total number of developers working on the project. FR is a File-Reviewer matrix and $S[k,i]$ represents the rank of the reviewers in the solution S. In fact, We are ranking the reviewers from 0 to P. For instance, if we have $P = 7$ developers (potential reviewers), a reviewer with rank 2 would be more appropriate than a reviewer with rank 4 to review the assigned file.

Both fitness functions "availability" and "expertise" are to be maximized. Thus, a lower rank (more suitable reviewer) would result in a higher fitness function (availability or expertise) since the rank ($S[k,i]$) is in the denominator. Therefore, the top ranked developers with high expertise/availability would be more likely to survive for the next evaluations of the multi-objective algorithm.

Collaboration. It is computed as the sum of all connections between recommended reviewers selected to work with a selected set of developers:

$$Collaboration = \sum_{k=1}^{N} \sum_{j=1}^{P} \sum_{i=1}^{M} DR[j,i] * FD[k,j] * (S[k,j] > 0) \tag{9.5}$$

Where $(s[k,j] > 0)$ is a binary mask for $S[k,j]$, meaning each entry with value 0 will remain 0 and each entry with value greater than 0 will become 1. P is total number of files requested to be reviewed, M is total number of reviewers working on the project and N is total number of developers working on the project. DR is a Developer-Reviewer matrix and FD is a File-Developer matrix where $FD[i, j]$ represents the number of times that the developer i worked on the file j. Therefore, the developer who changed the file under review (one or many times) can be assigned as a reviewer. The two matrix DR and FD are created during the data extraction step.

9.3.4.3 Change Operators

We applied single point crossover and swap mutation to explore and exploit the search space. Regarding crossover, we deploy a single random cut-point crossover. This operator is performed by generating a random crossover point. The cut-point is a binary block from crossover point K, which is a row-index and a column- index of a solution, to the end of the solution is copied from one parent, the rest is copied from the second parent. Then, it exchanges the sub-sequences before and after K between two parent individuals to create two offspring. In case we generate any infeasible offspring we apply a repair mechanism.

Our mutation—bit inversion changes the new offspring by swapping two rows in the matrix of the solution. Mutation can occur at each row in the matrix with some probability. The purpose of mutation is to prevent all solutions in the population falling into a local optimum.

9.4 Experiment and results

To evaluate our approach for recommending relevant peer reviewers, we conducted a set of experiments based on different versions of 9 open source systems. Due the stochastic nature of search algorithms, each experiment was repeated 30 times and the results were subsequently and statistically analyzed with the aim of comparing our multi-objective approach with both a mono-objective search technique based on an aggregation of expertise and collaborations [79] and also all the three objectives (AEC GA), and existing tools not based on heuristic search cHRev[299], REVFINDER[81], and ReviewBot[71] that only use expertise models without considering collaborations and availability of peer reviewers. Furthermore, we conducted an ablation study to compare our approach with three multi-objective variants considering two out of the three objectives (AC NSAG-II, AE NSGA-II and EC NSGA-II). All these existing studies were already evaluated in the literature on the same projects considered in this validation and the associated data is available thus we did not find a need to re-implement them. In this section, we present our research questions followed by experimental settings and parameters. Finally, we discuss our results for each of those research questions.

9.4.1 Research Questions

We focused on the following three research questions to evaluate the efficiency of our approach:

- **RQ1.** (Efficiency) Can the proposed approach precisely identify relevant peer reviewers?
- **RQ2.** (Comparison to search-based techniques) Does the proposed multi-objective approach perform significantly better than an existing mono-objective formulation aggregating expertise and collaboration [79], a mono-objective aggregation of all the three objectives (AEC GA) and variants of our multi-objective search considering two out of the three objectives (NSGA-II, AE NSGA-II and EC NSGA-II)?
- **RQ3.** (Comparison to state-of-the-art) Does our approach perform significantly better than existing peer reviewer recommendation techniques not based on heuristic search?

To answer RQ1, we validated the proposed multi-objective technique on 9 medium to large-size open-source systems, as detailed in the next section, to evaluate the correctness of our code-reviewer recommendation framework. To ensure a fair comparison with existing techniques, we followed a similar evaluation procedure by taking the most recent 1000 reviews and the reviewers assigned to these pull-requests as the ground truth. We built the different expertise, availability and collaborations models based on the review data just before the pull-request to evaluate in order to assign peer reviewers. We used GitHub API to extract the information about the pull request. From the information extracted, there is a tag 'reviewer' which contains the name of the reviewer. The name of the reviewer is also extracted from the comments under the pull request and this information is also provided by GitHub API.To this end, we used the following evaluation metrics:

- **Precision@k** denotes the number of correct recommended peer reviewers in the top k of recommended ones by the solution divided by the total number of peer reviewer recommendations to inspect.
- **Recall@k** denotes the number of correct recommended peer reviewers in the top k of recommended ones by the solution divided by the total number of expected reviewers to be recommended based on the ground truth.
- **MMR@k** measures the mean reciprocal rank which is an average rank of correct reviewers in the recommendation list. The higher the value the better.

Since the number of involved reviewers in each pull-request evaluation is limited in general to a few developers, we calculate these precision and recall metrics with different k values, 1, 3, 5 and 10.

To answer RQ2, we compared, using the above metrics, the performance of our multi-objective approach with an existing mono-objective formulation, based on a Genetic Algorithm, aggregating the two objectives of expertise and collaboration into one objective as the sum of them with equal weight [79]. We selected that mono-objective approach since it is the closest one to our work and already outperformed random search and other metaheuristic algorithms (simulated annealing and Particle Swarm Optimization) based on the results presented in [79]. Furthermore, we implemented a mono-objective approach aggregating all the three objectives (AEC GA) in one fitness function to evaluate the impact of adding the availability objective on the quality of the results by comparing with [79]. In addition, we compared different variants of our multi-objective approach including only two out of the three objectives (NSGA-II AE, AC and EC) to evaluate the contribution of each objective to the quality of the assignment results. The comparison between NSGA-II EC and the mono-objective search using only expertise and collaboration [79] can confirm the impact of the conflicting nature of the two objectives on the quality of the results.

To answer RQ3, we compared our multi-objective approach to different existing techniques not based on heuristic search:

- REVFINDER [81] uses the paths of the files to be reviewed to find reviewers who evaluated files in the same location.
- cHRev [299] is a hybrid approach using the frequency and recency of the history of the reviews to find relevant peer reviewers.
- ReviewBot [71] uses static analysis tools to find experienced reviewers

We limited the evaluation in RQ2 and RQ3 to Android, OpenStack, and Qt to ensure a fair comparison based on an existing benchmark [81, 300, 79]. More details about these projects will be presented in the next section.

9.4.2 Studied Projects

As described in Table 9.1, we used a data set of 9 open-source systems including 3 projects (OpenStack, Android and Qt) from existing code review benchmarks [300, 81, 79]. We used our tool to collect the data about Atomix, Tablesaw, Vavr, Takes, Dkpro-core, and Pac4j. In fact, our tool is implemented in a way that it takes a link to the project repository on GitHub and extracts all the needed data automatically similar to the existing public dataset for OpenStack, Android and Qt. To collect the data, we used GitHub API to send multiple queries to GitHub to get the needed information about the project under study. Actually, GitHub API provides different queries to extract the information about the pull requests, its reviewers, its changed files and all the committer names. The response to each query is a JSON file. Thus, we had to perform some cleaning and extracting steps to keep only the needed pieces of information.

- **Atomix:** A fault-tolerant distributed coordination framework.
- **Tablesaw:** A data science platform that includes a data-frame, an embedded column store, and hundreds of methods to transform, summarize, or filter data.
- **Vavr:** A functional component library that provides persistent data types and functional control structures.
- **Takes:** Opinionated web framework which is built around the concepts of True Object-Oriented Programming and immutability.
- **Dkpro-core:** A collection of reusable NLP tools for linguistic pre-processing, machine learning, lexical resources, etc.
- **Pac4j:** A security engine.
- **Android:** A software stack for mobile devices developed by Google.
- **OpenStack:** A large platform for cloud computing to manage a data-center.
- **Qt:** A widget toolkit for creating graphical user interfaces.

Table 9.1 shows statistics for the analyzed systems including the number of reviewers, the number of reviews in a project, the size, etc. All collected reviews are from closed pull-requests and contain at least one file. We selected these open source projects for our experiments since they contain a large number of code reviews and they have been studied in the software review literature [299, 81, 71] to ensure a fair comparison with the current state of the art.

Table 9.1: Summary of Studied Systems

Project (Studied Period)	# of classes	# of reviewers	# of files	# of reviews
Atomix (04/2017-11/2018)	1459	136	182280	4237
Tablesaw (06/2016-03/2018)	224	12	52837	1930
Vavr (04/2016-08/2018)	301	123	126683	4188
Takes (07/2015-05/2018)	472	264	50369	2687
Dkpro-core (03/2015-08/2018)	376	411	54695	4564
Pac4j (08/2014-10/2017)	302	29	31916	2282
Android (10/2008-01/2012)	563	94	26840	5126
OpenStack (07/2011-05/2012)	539	82	16953	6586
Qt (05/2011-05/2012)	782	202	78401	23810

9.4.3 Parameter Tuning and Statistical Tests

Since metaheuristic algorithms are stochastic optimizers, they can provide different results for the same problem instance from one run to another. For this reason, our experimental study was performed based on 30 independent simulation runs for each problem instance and the obtained results were statistically analyzed using the Friedman test with a 95% confidence level ($\alpha = 5\%$). Since the Friedman test results were significant, we used the Wilcoxon rank sum test [184] in a pairwise fashion (AEC NSGA-II versus each of the competitor approaches) in order to detect significant performance differences between the algorithms under comparison based on 30 independent runs. For deterministic techniques, we did not perform 30 independent runs. The Wilcoxon test allows testing the null hypothesis H0 that states that both algorithms medians' values for a particular metric are not statistically different against H1 which states the opposite. The Wilcoxon test does not require that the data sets follow a normal distribution since it operates on values' ranks instead of operating on the values themselves. Since we are comparing more than two different algorithms, we performed several pairwise comparisons based on Wilcoxon test to detect the statistical difference in terms of performance. To compare two algorithms based on a particular metric, we record the obtained metric's values for both algorithms over 30 runs. For deterministic techniques, we considered one value of each metric on each system. After that, we compute the metric's median value for each algorithm. Besides, we executed the Wilcoxon test with a 95% confidence level ($\alpha = 5\%$) on the recorded metric's values using the Wilcoxon MATLAB routine. If the returned p-value is less than 0.05 then we reject H0 and we can state that one algorithm outperforms the other, otherwise we cannot say anything in terms of performance difference between the two algorithms.

The above tests allow verifying whether the results are statistically different or not. However, it does not give any idea about the difference in magnitude. To this end, we used the Vargha and Delaney's A statistics which are non-parametric effect size measures. In our context, given the different performance metrics (such as Precision@k and Recall@k), the A statistics measure the probability that running an algorithm B1 (NSGA-II) yields better performance than running another algorithm B2 (such as GA). If the two algorithms are equivalent, then A = 0.5.

An often-omitted aspect in metaheuristic search is the tuning of algorithm parameters. In fact, parameter setting significantly influences the performance of a search algorithm on a particular problem. For this reason, for each search algorithm and each system, we performed a set of experiments using several population sizes: 10, 20, 30, 40 and 50. The stopping criterion was set to 100,000 fitness evaluations for all search algorithms to ensure fairness of comparison. We used a high number of evaluations as a stopping criterion since our approach requires multiple objectives. Each algorithm was executed 30 times with each configuration and then the comparison between the configurations was performed based on different metrics described previously using the Friedman test. The other parameter values were fixed by trial and error and are as follows: (1) crossover probability = 0.5; mutation probability = 0.4 where the probability of gene modification is 0.2. We used the same parameters of the existing work of Ouni et al., called RevRec, [79] for a fair comparison.

9.4.4 Results

Results for RQ1. The results of Tables 9.2-9.3 and Figure 9.4 confirm the efficiency of our multi-objective approach, based on NSGA-II, to identify relevant peer reviewers for pull-requests from all the 9 open source systems. Tables 9.2 and 9.3 show the average precision@k and recall@k results of our NSGA-II AEC technique on the various systems, with k equal to 1, 3, 5 and 10. For example, most of the recommended peer reviewers in the top 3 (k=3) are relevant (compared to the expected results) with precision over 60% on all the 9 systems. The lowest precision is around 47% for k=10 which still could be considered acceptable due to a large number of possible reviewers in the selected systems.

In terms of recall, Table 9.3 confirms that the majority of the expected peer reviewers to recommend are located in the top 10 (k=10) with a recall score over 53%. The highest recall is 78% for k=10 (Qt project). Since several pull-requests may require more than one peer reviewer, most of the highest recall scores are obtained for k=5 and k =10.

Figure 9.4 shows that NSGA-II was able to efficiently rank the recommended peer-reviewers. In fact, the median MMR on the different systems is higher than 68% with the highest score of 79% for the Open Stack project. This outcome is important since the efficient ranking of the recommended peer reviewer is one of the main motivations of our approach that consider not only the expertise but also the availability and the collaborations among reviewers. The availability in our case is considered based on the number of commits and files that a programmer is working on in the time period closest to the evaluated pull-request. We noticed that our technique does not have a bias toward the evaluated system. We had almost consistent average scores of precision, recall and the mean reciprocal rank.

Results for RQ2. Tables 9.2-9.3 and Figure 9.4 confirm that our multi-objective approach (AEC NSGA-II) is better, on average, than the existing mono-objective technique, RevRec [79], based on the 3 metrics of precision, recall and MMR on all the 9 systems. The median precision and recall values of the RevRec tool on the 9 systems are lower than 56% as described in Table 9.2 for all values of k (1, 3, 5 and 10). Furthermore, the EC

Table 9.2: Median Precision@k results for the search algorithms (multi-objective variants) including RevRec (mono-objective search) on all the systems based on 30 runs. All the results are statistically significant using the Friedman test and Wilcoxon with a 95% confidence level ($\alpha = 5\%$)

Project	k	**Precision@k**					
		AEC (NSGA-II)	RevRec (GA)	AEC (GA)	AC (NSGA-II)	AE (NSGA-II)	EC (NSGA-II)
Atomix	1	0.62	0.56	0.60	0.52	0.58	0.60
	3	0.58	0.44	0.47	0.41	0.44	0.51
	5	0.52	0.38	0.43	0.36	0.40	0.47
	10	0.47	0.41	0.41	0.38	0.41	0.45
Tablesaw	1	0.57	0.49	0.54	0.44	0.52	0.54
	3	0.64	0.52	0.56	0.41	0.52	0.60
	5	0.61	0.44	0.51	0.38	0.48	0.56
	10	0.55	0.41	0.46	0.40	0.44	0.50
Vavr	1	0.62	0.53	0.56	0.46	0.53	0.58
	3	0.58	0.47	0.52	0.41	0.44	0.54
	5	0.64	0.56	0.59	0.47	0.52	0.61
	10	0.66	0.51	0.56	0.44	0.53	0.60
Takes	1	0.57	0.48	0.52	0.42	0.50	0.52
	3	0.62	0.56	0.59	0.48	0.52	0.59
	5	0.55	0.46	0.50	0.40	0.43	0.52
	10	0.53	0.44	0.47	0.37	0.44	0.50
Dkpro-core	1	0.63	0.52	0.56	0.41	0.50	0.59
	3	0.57	0.47	0.51	0.34	0.43	0.54
	5	0.66	0.55	0.59	0.42	0.55	0.61
	10	0.59	0.43	0.49	0.37	0.47	0.52
Pac4j	1	0.61	0.52	0.56	0.41	0.54	0.58
	3	0.56	0.43	0.47	0.38	0.45	0.49
	5	0.59	0.39	0.46	0.33	0.42	0.51
	10	0.54	0.42	0.46	0.36	0.40	0.49
Android	1	0.68	0.58	0.62	0.51	0.60	0.64
	3	0.62	0.47	0.53	0.44	0.51	0.56
	5	0.53	0.39	0.43	0.37	0.41	0.45
	10	0.47	0.34	0.39	0.31	0.36	0.41
OpenStack	1	0.72	0.59	0.64	0.52	0.61	0.64
	3	0.61	0.51	0.54	0.46	0.52	0.56
	5	0.64	0.43	0.5	0.39	0.48	0.52
	10	0.54	0.36	0.39	0.33	0.36	0.43
Qt	1	0.58	0.49	0.51	0.46	0.47	0.53
	3	0.61	0.45	0.50	0.43	0.43	0.55
	5	0.54	0.41	0.45	0.39	0.38	0.48
	10	0.46	0.34	0.39	0.31	0.32	0.39

NSGA-II variant of our approach outperformed the mono-objective search aggregating the same objectives [79] based on the metrics on almost all the systems. Thus, an interesting observation is the clear conflicting objectives of expertise and collaborations which confirms

Table 9.3: Median Recall@k results for the search algorithms (multi-objective variants) including RevRec (mono-objective search) on all the systems based on 30 runs. All the results are statistically significant using the Friedman test and Wilcoxon test with a 95% confidence level ($\alpha = 5\%$)

Project	k	**Recall@k**					
		AEC (NSGA-II)	RevRec (GA)	AEC (GA)	AC (NSGA-II)	AE (NSGA-II)	EC (NSGA-II)
Atomix	1	0.56	0.43	0.48	0.39	0.46	0.51
	3	0.52	0.39	0.44	0.36	0.44	0.47
	5	0.61	0.46	0.53	0.41	0.50	0.58
	10	0.58	0.34	0.45	0.39	0.43	0.56
Tablesaw	1	0.51	0.43	0.48	0.37	0.46	0.48
	3	0.55	0.41	0.46	0.36	0.43	0.52
	5	0.52	0.38	0.44	0.35	0.40	0.50
	10	0.59	0.33	0.50	0.36	0.42	0.56
Vavr	1	0.53	0.41	0.48	0.38	0.43	0.50
	3	0.62	0.39	0.52	0.35	0.46	0.59
	5	0.55	0.42	0.50	0.40	0.44	0.52
	10	0.59	0.38	0.46	0.34	0.41	0.54
Takes	1	0.49	0.41	0.46	0.38	0.44	0.46
	3	0.53	0.44	0.47	0.39	0.42	0.50
	5	0.62	0.37	0.43	0.31	0.40	0.59
	10	0.66	0.34	0.51	0.32	0.39	0.62
Dkpro-core	1	0.54	0.47	0.44	0.40	0.42	0.51
	3	0.51	0.41	0.46	0.39	0.43	0.48
	5	0.58	0.39	0.49	0.36	0.46	0.53
	10	0.67	0.35	0.59	0.31	0.56	0.63
Pac4j	1	0.56	0.41	0.49	0.38	0.44	0.53
	3	0.62	0.36	0.53	0.31	0.50	0.58
	5	0.51	0.31	0.39	0.28	0.35	0.47
	10	0.63	0.38	0.49	0.31	0.47	0.60
Android	1	0.57	0.38	0.51	0.36	0.48	0.54
	3	0.72	0.51	0.63	0.48	0.60	0.67
	5	0.76	0.61	0.66	0.53	0.63	0.71
	10	0.79	0.71	0.77	0.66	0.71	0.77
OpenStack	1	0.59	0.41	0.49	0.38	0.45	0.56
	3	0.68	0.54	0.62	0.51	0.60	0.65
	5	0.76	0.61	0.68	0.53	0.64	0.72
	10	0.81	0.74	0.77	0.68	0.69	0.77
Qt	1	0.56	0.41	0.48	0.38	0.43	0.50
	3	0.66	0.50	0.58	0.47	0.50	0.61
	5	0.68	0.59	0.63	0.53	0.61	0.63
	10	0.76	0.65	0.68	0.57	0.65	0.71

our observation in the eBay survey that collaborations does not mean qualified reviewers (with high expertise) are assigned to review the pull-requests. The same observation is valid for the ranking of recommended peer reviewers based on the MMR measure as described in

Figure 9.4. For instance, the MMR score for AEC NSGA-II is 78% on the Takes project while it is limited to 61% for RevRec.

The outperformance of NSGA-II can be explained as well by the consideration of the new objective of availability which may reflect the reality of how peer reviewers are manually assigned to reduce delays. In fact, the aggregation of all the three objectives in a mono-objective search (AEC GA) is performing better than [79] which confirms the positive contribution of the availibility objective on the quality of the results. The least performance of our multi-objective approach in terms of MMR (slightly less than RevRec) was observed for the Dkpro-core and pac4j projects. While investigating the reasons behind this decreased performance, we found out that the main reason is that these projects have a large enough number of contributors comparing to their sizes(in terms of files, commits and pull-request). In fact, the ratio *'contributors to size'* is larger than the other projects. Thus, the availability objective may not represent a big concern for these projects unlike the others since they have enough contributors to review the changed files/pull-requests.

All these results were statistically significant on 30 independent runs using the Friedman test and Wilcoxon test (pairwise comparison) with a 95% confidence level ($\alpha < 5\%$). We also found the results of the Vargha Delaney A_{12} statistic are higher than 0.8 (large) on all the systems which confirms the significant outperformance of AEC NSGA-II comparing to the mono-objective formulation. The detailed effect size results can be found in Tables 9.4 and 9.5.

Table 9.4: The effect size for Precision based on 30 runs when comparing AEC NSGA-II versus each of the search algorithms.

Project	Effect Size-RevRec (GA)	Effect Size-AEC (GA)	Effect Size-AC (NSGA-II)	Effect Size-AE (NSGA-II)	Effect Size-EC (NSGA-II)
Atomix	0.52	0.61	0.82	0.76	0.58
Tablesaw	0.39	0.72	0.79	0.73	0.63
Vavr	0.87	0.63	0.86	0.78	0.71
Takes	0.64	0.68	0.91	0.83	0.68
Dkpro-core	0.92	0.77	0.83	0.71	0.72
Pac4j	0.86	0.72	0.72	0.84	0.66
Android	0.52	0.64	0.77	0.92	0.74
OpenStack	0.76	0.68	0.84	0.81	0.63
Qt	0.94	0.71	0.92	0.83	0.61

Results for RQ3. Since it is not sufficient to compare our approach with just search-

Table 9.5: The effect size for Recall based on 30 runs when comparing AEC NSGA-II with each of the search algorithms.

Project	Effect Size-RevRec (GA)	Effect Size-AEC (GA)	Effect Size-AC (NSGA-II)	Effect Size-AE (NSGA-II)	Effect Size-EC (NSGA-II)
Atomix	0.64	0.66	0.83	0.72	0.61
Tablesaw	0.82	0.62	0.75	0.69	0.53
Vavr	0.93	0.71	0.83	0.77	0.64
Takes	0.72	0.63	0.91	0.82	0.68
Dkpro-core	0.89	0.74	0.84	0.91	0.59
Pac4j	0.74	0.61	0.88	0.73	0.71
Android	0.91	0.77	0.94	0.68	0.63
OpenStack	0.83	0.82	0.83	0.73	0.69
Qt	0.72	0.64	0.86	0.77	0.57

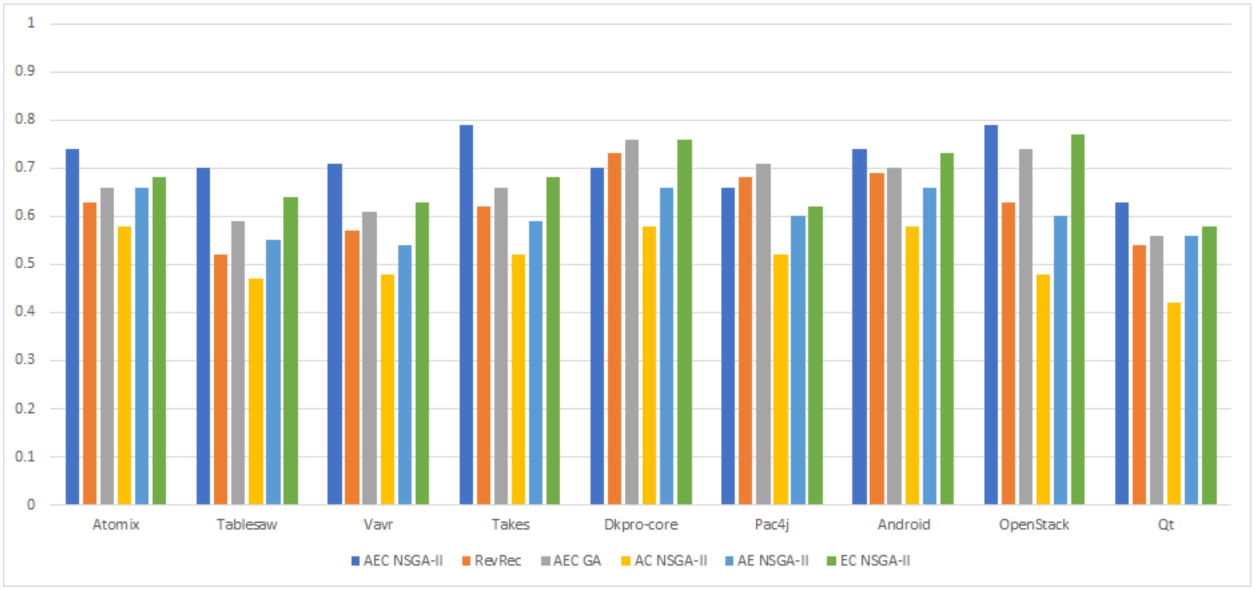


Figure 9.4: Median MMR results for the different search algorithms on all systems based on 30 runs. All the results are statistically significant using the Friedman test with a 95% confidence level ($\alpha = 5\%$)

based algorithms, we compared the performance of NSGA-II to three different peer reviewer recommendation techniques which are not based on heuristic search, as described in Tables 9.6 and 9.7, and Figure 9.5.

Similar to the comparison with RevRec, we used the precision@k, recall@k and MMR measures with k ranging from 1 to 10. NSGA-II achieves better results, on average than the other three methods on all the three projects. For example, our approach achieved a Precision@k median of 63%, 59%, 48% and 43% are achieved for k= 1, 3, 5 and 10 respectively as described in Table 9.6. In comparison, CHrev achieved a median Precision@k of 58%, 47%, 39%, and 34% are obtained for k= 1, 3, 5 and 10. CHRev has the highest precision among all the remaining tools of REVFINDER and ReviewBot. Similar observations are also valid for the recall@k and MMR.

Table 9.6: Median Precision@k results for all the approaches on three systems based on 30 runs. All the results are statistically significant using the Friedman test and Wilcoxon test with a 95% confidence level (α = 5%)

Project	K	**Precision@k**								
		ACE (NSGA-II)	AEC (GA)	AC (NSGA-II)	AE (NSGA-II)	EC (NSGA-II)	RevRec (GA)	cHRev	REVFINDER	ReviewBot
Android	1	0.68	0.62	0.51	0.60	0.64	0.58	0.50	0.34	0.21
	3	0.62	0.53	0.44	0.51	0.56	0.47	0.35	0.25	0.17
	5	0.53	0.43	0.37	0.41	0.45	0.39	0.30	0.22	0.12
	10	0.47	0.39	0.31	0.36	0.41	0.34	0.26	0.18	0.09
OpenStack	1	0.72	0.64	0.52	0.61	0.64	0.59	0.48	0.32	0.24
	3	0.61	0.54	0.46	0.52	0.56	0.51	0.42	0.27	0.20
	5	0.64	0.50	0.39	0.48	0.52	0.43	0.38	0.25	0.16
	10	0.54	0.39	0.33	0.36	0.43	0.36	0.31	0.21	0.11
Qt	1	0.58	0.51	0.46	0.47	0.53	0.49	0.45	0.30	0.22
	3	0.61	0.50	0.43	0.43	0.55	0.45	0.40	0.27	0.19
	5	0.54	0.45	0.39	0.38	0.48	0.41	0.37	0.21	0.13
	10	0.46	0.39	0.31	0.32	0.39	0.34	0.31	0.16	0.09

Table 9.7: Median Recall@k results for all the approaches on three systems based on 30 runs. All the results are statistically significant using the Friedman test and Wilcoxon test with a 95% confidence level (α = 5%)

Project	K	**Recall@k**								
		ACE (NSGA-II)	AEC (GA)	AC (NSGA-II)	AE (NSGA-II)	EC (NSGA-II)	RevRec (GA)	cHRev	REVFINDER	ReviewBot
Android	1	0.57	0.51	0.36	0.48	0.54	0.38	0.27	0.18	0.11
	3	0.72	0.63	0.48	0.60	0.67	0.51	0.50	0.39	0.19
	5	0.76	0.66	0.53	0.63	0.71	0.61	0.61	0.48	0.29
	10	0.79	0.77	0.66	0.71	0.77	0.71	0.65	0.54	0.38
OpenStack	1	0.59	0.49	0.38	0.45	0.56	0.41	0.31	0.15	0.12
	3	0.68	0.62	0.51	0.60	0.65	0.54	0.39	0.29	0.20
	5	0.76	0.68	0.53	0.64	0.72	0.61	0.52	0.37	0.32
	10	0.81	0.77	0.68	0.69	0.77	0.74	0.66	0.46	0.39
Qt	1	0.56	0.48	0.38	0.43	0.50	0.41	0.33	0.14	0.90
	3	0.66	0.58	0.47	0.50	0.61	0.50	0.47	0.27	0.16
	5	0.68	0.63	0.53	0.61	0.63	0.59	0.52	0.35	0.24
	10	0.76	0.68	0.57	0.65	0.71	0.65	0.60	0.43	0.30

9.5 Threats to Validity

Conclusion validity is concerned with the statistical relationship between the treatment and the outcome. We addressed conclusion threats to validity by performing 30 independent simulation runs for each problem instance and statistically analyzing the obtained results using the Friedman test with a 95% confidence level (α = 5%). However, the parameter tuning of the different optimization algorithms used in our experiments creates another

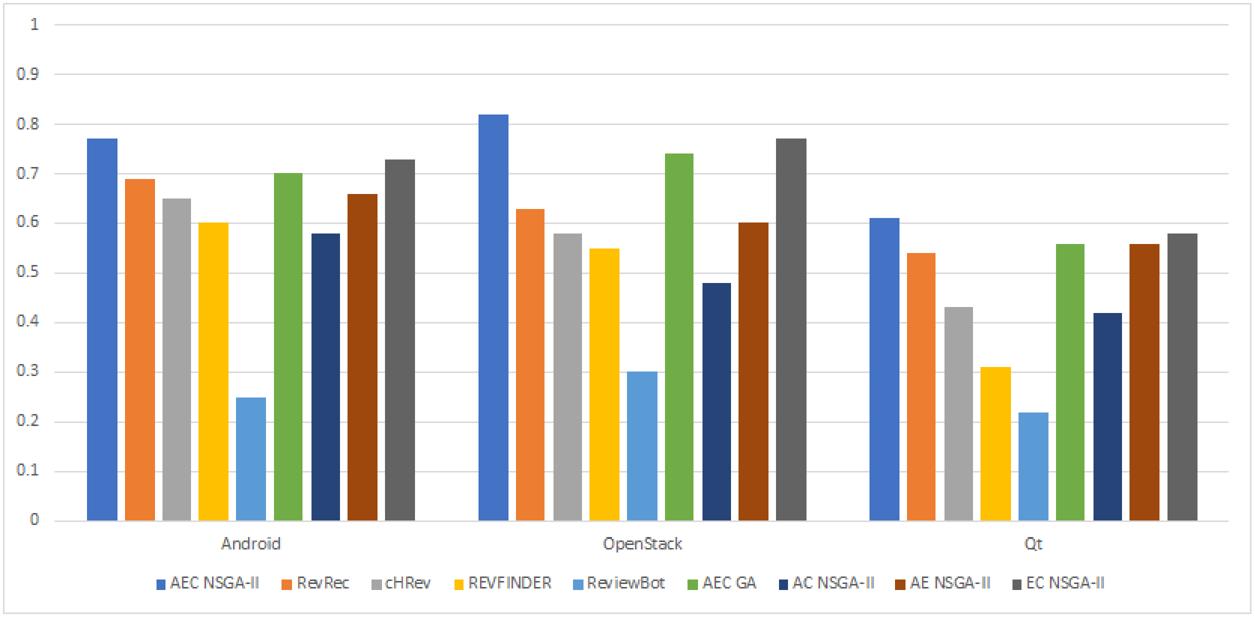


Figure 9.5: Median MMR results for all the approaches on three systems based on 30 runs. All the results are statistically significant using the Friedman test and Wilcoxon test with a 95% confidence level ($\alpha = 5\%$)

internal threat that we need to evaluate in our future work. The parameter values used in our experiments were determined by trial-and-error [301]. In addition, the estimation of the availability of reviewers on open source systems may not be very accurate.

Construct validity is concerned with the relationship between theory and what is observed. The definition of expertise and collaborations can be subjective and hard to formalize thus further empirical studies are required to validate the different metrics used in our work. We are planning to consider other possible formations as part of our future work and compare between them. Additionally, our current definition of the availability needs further improvement. In fact, reviewers can be assigned other types of development activities than coding (e.g., testing, design/architecture, requirements analysis, etc.). The data about these activities are not always available. However, the formulation of our fitness function is easy to modify in a way that enables managers to enter the number of tasks per reviewer, especially the ones that they are beyond code reviews.

External validity refers to the generalizability of our findings. In this study, we performed our experiments on different widely used open-source systems belonging to different domains and having different sizes. However, we cannot assert that our results can be generalized to other systems. Future replications of this study are necessary to confirm our results with a larger set of pull requests and reviewers.

Another threat to our approach could be the effort required by the manager to select the preferred solution. In general, the preferences are defined based on the current context such as: the urgency to release code quickly, available resources, speedy growth phase of the project, etc. These different contexts are not changing daily and they are not related to only one or few pull-requests but they are more related to the situation of the whole project. To mitigate this threat, we provide the distribution of the solutions of the Pareto front which can be ranked based on the preferred fitness functions or based on the current context. Thus, the preferred solution can be selected in an easier and faster way.

9.6 Conclusion

We formulated the recommendation of peer code reviewers as a multi-objective problem to find a trade-off between the competing objectives of expertise, availability and history of collaborations. Unlike existing approaches, our approach can sacrifice expertise to avoid a delay caused by limited resources (e.g. low peer reviewer availability). Our evaluation results confirm the efficiency of our multi-objective approach on 9 open source projects in finding better reviewer recommendations, as compared to the state of the art [79]. Furthermore, our survey with practitioners highlighted the importance of managing code reviews to reduce delays while ensuring high expertise as much as possible.

CHAPTER X

Conclusion

The features and improvements that were delivered in this dissertation and the results that were achieved are summarized in this chapter. In addition, the suggested possible improvements to the proposed contributions are discussed.

10.1 Summary

In **Chapter I** and **Chapter II**, we defined the research context and the challenges, the contributions of this thesis, required background, and state-of-the-art and related works to our approaches.

In **Chapter III**, we proposed a bi-level multi-objective approach for the web service antipatterns detection problem. In our approach adaptation, the upper level generates a set of detection rules which are a combination of QoS, Interface, and code level metrics, using two conflicting fitness functions. The first objective is to maximize the coverage of both the base of defect examples and artificial defects generated by the lower level and to minimize the coverage of well-designed web service examples. The second objective is to minimize the size of a detection rule. The lower level generates artificial defects that cannot be generated by the upper-level detection rules which will help to generate fitter rules.

We implemented our proposed approach and evaluated it on a benchmark of 662 web services and several common web service antipattern types. The empirical study shows that proposed bi-level multi-objective optimization approach outperforms our previous multi-objective approach, bi-level approach and other state-of-the-art approaches.

In **Chapter IV**, we presented a novel way to enable interactive refactoring by combining the exploration of quality improvements (objective space) and refactoring locations (decision space). Our approach helped developers to quickly explore the Pareto front of refactoring solutions that can be generated using multi-objective search. The clustering of the decision space helped the developers identify the most diverse refactoring solutions among ones located within the same cluster in the objective space, improving some desired quality attributes. To evaluate the effectiveness of our tool, we conducted an evaluation with human subjects who evaluated the tool and compared it with the state-of-the-art refactoring techniques. Our evaluation results provide evidence that the insights from both the decision and objective spaces helped developers to quickly express their preferences and converge towards relevant refactorings that met the developers' expectations.

Therefore, in **Chapter V**, we presented a first attempt to recommend refactorings by analyzing commit messages. The salient feature of the proposed **RefCom** approach is its ability to capture developers need, from their commit messages, and propose to them refactorings to enhance their changes to better address quality issues. To evaluate the effectiveness of our technique, we applied it to six open-source projects and compared it with state-of-the-art approaches that rely on static and dynamic analysis. Our results show promising evidence on the usefulness of the proposed commit-based refactoring approach.

To assist developers in documenting changes in terms quality attributes improvement and refactoring while submitting their code changes on GitHub, we presented in **Chapter VI**, an interactive documentation bot to document the developers changes. The bot enables 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.

Chapter VII is dedicated to our empirical study about refactoring documentation. In fact, we used a combination of interviews and a survey to understand refactoring documentation from practitioners' perspective. We started first with a set of interviews with practitioners to define a refactoring documentation model. Then, we performed a large online survey to gather the experiences of practitioners with the importance, frequency, and difficulty of refactoring documentation for the different components of our model. We found 5 main important refactoring documentation components for practitioners.

The outcomes of this empirical study can be used to improve the quality of refactoring documentation. Furthermore, researchers and tool builders can use the discovered components and the experiences of the developers to build refactoring documentation generation tools.

Furthermore, we addressed the problem of peer code reviewers recommendation in chapter IX. We proposed a multi-objective approach to find a trade-off between the competing objectives of expertise, availability and history of collaborations. Unlike existing approaches, our approach can sacrifice expertise to avoid a delay caused by limited resources (e.g. low peer reviewer availability).

Finally, in **Chapter VIII**, we have shown how NMT techniques can be enhanced to address the problem of commit messages generation for composite code changes. Our NMT-BERT model outperforms the state-of-the art techniques, shedding light on the importance of working with a pretrained model as the source for embeddings. Diff files can be very lengthy, which remains a problem that we are planning to address in the future. Lastly, we consider this presented work as a step forward, suggesting solutions and insights on how to advance the state-of-the-art techniques for commit messages generation.

10.2 Future Work

Some future works direction can be summarized as follows:

1. Nowadays, developers are continuously applying refactoring via commits/pull-requests

in CI environment. Therefore, there is a need to establish an empirical study (e.g. a large-scale survey) to understand the impact of continuous integration environment on refactoring. Multiple factors can impact refactoring, especially in CI environment. A new research study can focus on presenting continuous refactoring as multi-faced complex problem from different dimensions such as technical, social and business perspectives.

2. Regarding composite changes documentation, we plan to extend our refactoring documentation bot to reduce the developer's interaction effort and to improve the documentation of complex changes via linking the quality changes with quality issues, also called antipatterns. Once the complex changes are linked to the quality improvement and quality issues, the bot can document the changes that affect a specific architectural group. Additionally, the work can be extended by incorporating natural languages techniques to understand the dependencies between complex changes and feed the new knowledge into a better neural machine translation mode to generate more relevant commit messages for the developer.

3. With the popular use of version control and issue tracking systems, developers suffer from an increase workload to review code, deal with open issues, answer comments and merge pull requests. Thus, many open issues are subject to remain unresolved. From our research and previous analysis of code sources on GitHub (contributions V, VI, VII, and VIII), we found significant number of open issues are related to quality issues and refactoring and they remain open for long time as developers are busy with other issues related to bugs and functional requirements. Therefore, it could be interesting as future work to leverage the functionalities of our current documentation bot in addition to other functionalities (e.g, refactoring documentation bot, antipatterns detection bot, *etc.*) to build a management bot that can continuously check a software repository and try to support the developers and assist them in resolving the open issues related

to quality and refactoring.

10.3 Publications List

- Soumaya Rebai, Marouane Kessentini, Hanzhang Wang, Bruce R. Maxim, "Web service design defects detection: A bi-level multi-objective approach." Information and Software Technology Journal, Vol 121: 106255, 21 pages, Impact factor: 2.92 (2020) https://doi.org/10.1016/j.infsof.2019.106255

- S. Rebai, O. Ben Sghaier, V. Alizadeh, M. Kessentini and M. Chater, "Interactive Refactoring Documentation Bot," 2019 19th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM), Cleveland, OH, USA, 2019, pp. 152-162, Acceptance rate 24%
https://doi.org/10.1109/SCAM.2019.00026.

- Soumaya Rebai, Marouane Kessentini, Vahid Alizadeh, Oussama Ben Sghaier, and Rick Kazman, "Recommending Refactorings via Commit Message Analysis." Information and Software Technology (2020): Volume 126, 2020,106332, ISSN 0950-5849. Impact factor: 2.92 (2020) https://doi.org/10.1016/j.infsof.2020.106332

- Soumaya Rebai, Vahid Alizadeh, Marouane Kessentini, Houcem Fehri, and Rick Kazman, "*Enabling Decision and Objective Space: Exploration for Interactive Multi-Objective Refactoring.*" IEEE Transactions on Software Engineering (2020), DOI: 10.1109/TSE.2020.3024814, Impact Factor 6.11.

- Rebai, Soumaya, Abderrahmen Amich, Somayeh Molaei, Marouane Kessentini, and Rick Kazman. "Multi-objective code reviewer recommendations: balancing expertise, availability and collaborations." Automated Software Engineering (2020): 1-28. https://doi.org/10.1007/s10515-020-00275-6

- Soumaya Rebai, Siyuan Jiang, Marouane Kessentini, Weijing Huang and Fabio

Palomba , "*Commit Message Generation of Composite Changes.*" **under review** at the IEEE Transactions in Software Engineering journal.

- Soumaya Rebai, Marouane Kessentini, Tushar Sharma and Thiago Ferreira , "*4W+H Model for Refactoring Documentation: A Practitioners' Perspective.*" **under review** at the IEEE Transactions in Software Engineering journal.

BIBLIOGRAPHY

[1] V. Alizadeh, M. Kessentini, W. Mkaouer, M. Ocinneide, A. Ouni, and Y. Cai, "An interactive and dynamic search-based approach to software refactoring recommendations," *IEEE Transactions on Software Engineering*, 2018.

[2] 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*, pp. 464–474, ACM, 2018.

[3] M. W. Mkaouer, M. Kessentini, S. Bechikh, K. Deb, and M. Ó Cinnéide, "Recommendation system for software refactoring using innovization and interactive dynamic optimization," in *Proceedings of the 29th ACM/IEEE International Conference on Automated software Engineering*, pp. 331–336, ACM, 2014.

[4] A. Ouni, M. Kessentini, H. Sahraoui, K. Inoue, and K. Deb, "Multi-criteria code refactoring using search-based software engineering: an industrial case study," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 25, no. 3, p. 23, 2016.

[5] M. P. Papazoglou, P. Traverso, S. Dustdar, and F. Leymann, "Service-oriented computing: a research roadmap," *International Journal of Cooperative Information Systems*, vol. 17, no. 02, pp. 223–255, 2008.

[6] E. Newcomer and G. Lomow, *Understanding SOA with Web Services.* Addison-Wesley, 2005.

[7] M. P. Papazoglou, "Service-oriented computing: Concepts, characteristics and directions," in *Web Information Systems Engineering, 2003. WISE 2003. Proceedings of the Fourth International Conference on*, pp. 3–12, IEEE, 2003.

[8] J. Král and M. Zemlicka, "Popular SOA Antipatterns," in *Computation World: Future Computing, Service Computation, Cognitive, Adaptive, Content, Patterns*, pp. 271–276, 2009.

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

[10] R. Almhana, W. Mkaouer, M. Kessentini, and A. Ouni, "Recommending relevant classes for bug reports using multi-objective search," in *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 286–295, IEEE, 2016.

[11] M. Kessentini and A. Ouni, "Detecting android smells using multi-objective genetic programming," in *Proceedings of the 4th International Conference on Mobile Software Engineering and Systems*, pp. 122–132, IEEE Press, 2017.

[12] E. A. AlOmar, M. W. Mkaouer, A. Ouni, and M. Kessentini, "On the impact of refactoring on the relationship between quality attributes and design metrics," in *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pp. 1–11, IEEE, 2019.

[13] J. Kerievsky, *Refactoring to patterns.* Pearson Deutschland GmbH, 2005.

[14] R. Kazman, Y. Cai, R. Mo, Q. Feng, L. Xiao, S. Haziyev, V. Fedak, and A. Shapochka, "A case study in locating the architectural roots of technical debt," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 2, pp. 179–188, IEEE, 2015.

[15] J. Carriere, R. Kazman, and I. Ozkaya, "A cost-benefit framework for making architectural decisions in a business context," in *2010 ACM/IEEE 32nd International Conference on Software Engineering*, vol. 2, pp. 149–157, IEEE, 2010.

[16] M. Kessentini, H. Sahraoui, and M. Boukadoum, "Example-based model-transformation testing," *Automated Software Engineering*, vol. 18, no. 2, pp. 199–224, 2011.

[17] 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*, p. 2, ACM, 2010.

[18] A. Ghannem, M. Kessentini, and G. El Boussaidi, "Detecting model refactoring opportunities using heuristic search," in *Proceedings of the 2011 Conference of the Center for Advanced Studies on Collaborative Research*, pp. 175–187, 2011.

[19] S. Kalboussi, S. Bechikh, M. Kessentini, and L. B. Said, "Preference-based many-objective evolutionary testing generates harder test cases for autonomous agents," in *International Symposium on Search Based Software Engineering*, pp. 245–250, Springer, 2013.

[20] M. Kessentini, R. Mahaouachi, and K. Ghedira, "What you like in design use to correct bad-smells," *Software Quality Journal*, vol. 21, no. 4, pp. 551–571, 2013.

[21] M. Kessentini, A. Ouni, P. Langer, M. Wimmer, and S. Bechikh, "Search-based metamodel matching with structural and syntactic measures," *Journal of Systems and Software*, vol. 97, pp. 1–14, 2014.

[22] U. Mansoor, M. Kessentini, P. Langer, M. Wimmer, S. Bechikh, and K. Deb, "MOMM: Multi-objective model merging," *Journal of Systems and Software*, vol. 103, pp. 423–439, 2015.

[23] "The developer coefficient." URL: https://stripe.com/reports/developer-coefficient-2018.

[24] B. Du Bois, S. Demeyer, and J. Verelst, "Refactoring-improving coupling and cohesion of existing code," in *11th working conference on reverse engineering*, pp. 144–151, IEEE, 2004.

[25] S. R. Foster, W. G. Griswold, and S. Lerner, "Witchdoctor: Ide support for real-time auto-completion of refactorings," in *2012 34th International Conference on Software Engineering (ICSE)*, pp. 222–232, IEEE, 2012.

[26] X. Ge and E. Murphy-Hill, "Benefactor: a flexible refactoring tool for eclipse," in *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, pp. 19–20, 2011.

[27] M. Kim, M. Gee, A. Loh, and N. Rachatasumrit, "Ref-finder: a refactoring reconstruction tool based on logic query templates," in *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pp. 371–372, ACM, 2010.

[28] R. Marinescu, "Detection strategies: Metrics-based rules for detecting design flaws," in *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.*, pp. 350–359, IEEE, 2004.

[29] E. Murphy-Hill, C. Parnin, and A. P. Black, "How we refactor, and how we know it," *IEEE Transactions on Software Engineering*, vol. 38, no. 1, pp. 5–18, 2011.

[30] D. Dig, C. Comertoglu, D. Marinov, and R. Johnson, "Automated detection of refactorings in evolving components," in *European Conference on Object-Oriented Programming*, pp. 404–428, Springer, 2006.

[31] D. Batory, J. N. Sarvela, and A. Rauschmayer, "Scaling step-wise refinement," *IEEE Transactions on Software Engineering*, vol. 30, no. 6, pp. 355–371, 2004.

[32] J. Kim, D. Batory, D. Dig, and M. Azanza, "Improving refactoring speed by 10x," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pp. 1145–1156, IEEE, 2016.

[33] W. Kessentini, M. Kessentini, H. Sahraoui, S. Bechikh, and A. Ouni, "A cooperative parallel search-based software engineering approach for code-smells detection," *IEEE Transactions on Software Engineering*, vol. 40, no. 9, pp. 841–861, 2014.

[34] A. Ouni, M. Kessentini, H. Sahraoui, and M. Boukadoum, "Maintainability defects detection and correction: a multi-objective approach," *Automated Software Engineering*, vol. 20, no. 1, pp. 47–79, 2013.

[35] M. W. Mkaouer, M. Kessentini, S. Bechikh, K. Deb, and M. Ó Cinnéide, "Recommendation system for software refactoring using innovization and interactive dynamic optimization," in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, pp. 331–336, ACM, 2014.

[36] A. Ouni, M. Kessentini, H. Sahraoui, K. Inoue, and K. Deb, "Multi-criteria code refactoring using search-based software engineering: An industrial case study," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 25, no. 3, p. 23, 2016.

[37] D. Dig, C. Comertoglu, D. Marinov, and R. Johnson, "Automated detection of refactorings in evolving components," in *European Conference on Object-Oriented Programming*, pp. 404–428, Springer, 2006.

[38] M. Kessentini, T. J. Dea, and A. Ouni, "A context-based refactoring recommendation approach using simulated annealing: two industrial case studies," in *Proceedings of the Genetic and Evolutionary Computation Conference*, pp. 1303–1310, ACM, 2017.

[39] Y. Cai and K. Sullivan, "A formal model for automated software modularity and evolvability analysis," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 21, no. 4, p. 21, 2012.

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

[41] W. Brown, R. Malveau, S. McCormick, and T. Mowbray, *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis.* Wiley, 1998.

[42] E. Murphy-Hill, C. Parnin, and A. P. Black, "How we refactor, and how we know it," *IEEE Transactions on Software Engineering*, vol. 38, no. 1, pp. 5–18, 2011.

[43] 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.

[44] 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*, pp. 68–75, ACM, 2005.

[45] 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*, pp. 255–265, IEEE Press, 2012.

[46] 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)*, pp. 70–79, IEEE, 2007.

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

[48] M. Fowler, *Refactoring: improving the design of existing code.* Addison-Wesley Longman Publishing Co., Inc., 1999.

[49] M. Alshayeb, "Empirical investigation of refactoring effect on software quality," *Information and software technology*, vol. 51, no. 9, pp. 1319–1326, 2009.

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

[51] J. Mylopoulos, L. Chung, and B. Nixon, "Representing and using nonfunctional requirements: A process-oriented approach," *IEEE Transactions on Software Engineering*, no. 6, pp. 483–497, 1992.

[52] G. Szőke, C. Nagy, R. Ferenc, and T. Gyimóthy, "A case study of refactoring large-scale industrial systems to efficiently improve source code quality," in *International Conference on Computational Science and Its Applications*, pp. 524–540, Springer, 2014.

[53] Y. Lin, X. Peng, Y. Cai, D. Dig, D. Zheng, and W. Zhao, "Interactive and guided architectural refactoring with search-based recommendation," in *ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 535–546, ACM, 2016.

[54] E. Mealy, D. Carrington, P. Strooper, and P. Wyeth, "Improving usability of software refactoring tools," in *2007 Australian Software Engineering Conference (ASWEC'07)*, pp. 307–318, IEEE, 2007.

[55] S. Negara, N. Chen, M. Vakilian, R. E. Johnson, and D. Dig, "A comparative study of manual and automated refactorings," in *European Conference on Object-Oriented Programming*, pp. 552–576, Springer, 2013.

[56] X. Ge, Q. L. DuBose, and E. Murphy-Hill, "Reconciling manual and automatic refactoring," in *2012 34th International Conference on Software Engineering (ICSE)*, pp. 211–221, IEEE, 2012.

[57] E. Lim, N. Taksande, and C. Seaman, "A balancing act: What software practitioners have to say about technical debt," *IEEE software*, vol. 29, no. 6, pp. 22–27, 2012.

[58] P. Kruchten, R. L. Nord, I. Ozkaya, and D. Falessi, "Technical debt: towards a crisper definition report on the 4th international workshop on managing technical debt," *ACM SIGSOFT Software Engineering Notes*, vol. 38, no. 5, pp. 51–54, 2013.

[59] C. Seaman and Y. Guo, "Measuring and monitoring technical debt," in *Advances in Computers*, vol. 82, pp. 25–46, Elsevier, 2011.

[60] W. Cunningham, "The wycash portfolio management system," *ACM SIGPLAN OOPS Messenger*, vol. 4, no. 2, pp. 29–30, 1992.

[61] M. M. Lehman and J. F. Ramil, "Software evolution—background, theory, practice," *Information Processing Letters*, vol. 88, no. 1-2, pp. 33–44, 2003.

[62] W. J. Brown, R. C. Malveau, and W. Brown, "Hwm iii, and tj mowbray," *Anti Patterns: Refactoring Software, Architectures, and Projects in Crisis*, 1998.

[63] "git." URL: https://git-scm.com/.

[64] 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*, pp. 414–423, IEEE Press, 2017.

[65] 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*, pp. 373–384, ACM, 2018.

[66] Q. Liu, Z. Liu, H. Zhu, H. Fan, B. Du, and Y. Qian, "Generating commit messages from diffs using pointer-generator network," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pp. 299–309, IEEE, 2019.

[67] S. Jiang, A. Armaly, and C. McMillan, "Automatically generating commit messages from diffs using neural machine translation," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, pp. 135–146, IEEE Press, 2017.

[68] M. S. Ahmed and A. Tabassum, "Automatic contextual commit message generation: A two-phase conversion approach," 2018.

[69] S. Xu, Y. Yao, F. Xu, T. Gu, H. Tong, and J. Lu, "Commit message generation for source code changes," in *IJCAI*, 2019.

[70] S. E. S. Committee *et al.*, "IEEE standard for software reviews," *IEEE Std*, pp. 1028–1997, 1997.

[71] V. Balachandran, "Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation," in *Proceedings of the 2013 International Conference on Software Engineering*, pp. 931–940, IEEE Press, 2013.

[72] P. C. Rigby and C. Bird, "Convergent contemporary software peer review practices," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pp. 202–212, ACM, 2013.

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

[74] Z. Liu, X. Xia, C. Treude, D. Lo, and S. Li, "Automatic generation of pull request descriptions," in *Proceedings of the 34nd IEEE/ACM International Conference on Automated Software Engineering*, IEEE Press, 2019.

[75] S. Rebai, O. B. Sghaier, V. Alizadeh, M. Kessentini, and M. Chater, "Interactive refactoring documentation bot," in *2019 19th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pp. 152–162, IEEE.

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

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

[78] R. P. Buse and W. R. Weimer, "Automatically documenting program changes," in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, pp. 33–42, ACM, 2010.

[79] A. Ouni, R. G. Kula, and K. Inoue, "Search-based peer reviewers recommendation in modern code review," in *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 367–377, IEEE, 2016.

[80] M. B. Zanjani, H. Kagdi, and C. Bird, "Automatically recommending peer reviewers in modern code review," *IEEE Transactions on Software Engineering*, vol. 42, no. 6, pp. 530–543, 2016.

[81] 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 *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pp. 141–150, IEEE, 2015.

[82] 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.

[83] M. O'Keeffe and M. Ó. Cinnéide, "Search-based refactoring: an empirical study," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 20, no. 5, pp. 345–364, 2008.

[84] M. Ó Cinnéide, L. Tratt, M. Harman, S. Counsell, and I. Hemati Moghadam, "Experimental assessment of software metrics using automated refactoring," in *Proceedings of the ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, pp. 49–58, ACM, 2012.

[85] 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*, pp. 1341–1348, ACM, 2010.

[86] 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.

[87] R. Khatchadourian and H. Masuhara, "Automated refactoring of legacy java software to default methods," in *Proceedings of the 39th International Conference on Software Engineering*, pp. 82–93, IEEE Press, 2017.

[88] 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*, pp. 26–33, ACM, 2002.

[89] J.-C. Chen and S.-J. Huang, "An empirical analysis of the impact of software development problem factors on software maintainability," *Journal of Systems and Software*, vol. 82, no. 6, pp. 981–992, 2009.

[90] M. P. Robillard, "What makes apis hard to learn? answers from developers," *IEEE software*, vol. 26, no. 6, pp. 27–34, 2009.

[91] B. Dagenais and M. P. Robillard, "Creating and evolving developer documentation: understanding the decisions of open source contributors," in *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pp. 127–136, 2010.

[92] M. P. Robillard and R. Deline, "A field study of api learning obstacles," *Empirical Software Engineering*, vol. 16, no. 6, pp. 703–732, 2011.

[93] G. Garousi, V. Garousi, M. Moussavi, G. Ruhe, and B. Smith, "Evaluating usage and quality of technical software documentation: an empirical study," in *Proceedings of the 17th International Conference on Evaluation and Assessment in Software Engineering*, pp. 24–35, 2013.

[94] G. Garousi, V. Garousi-Yusifoğlu, G. Ruhe, J. Zhi, M. Moussavi, and B. Smith, "Usage and usefulness of technical software documentation: An industrial case study," *Information and Software Technology*, vol. 57, pp. 664–682, 2015.

[95] R. Plösch, A. Dautovic, and M. Saft, "The value of software documentation quality," in *2014 14th International Conference on Quality Software*, pp. 333–342, IEEE, 2014.

[96] G. Uddin and M. P. Robillard, "How api documentation fails," *IEEE Software*, vol. 32, no. 4, pp. 68–75, 2015.

[97] S. Sohan, F. Maurer, C. Anslow, and M. P. Robillard, "A study of the effectiveness of usage examples in rest api documentation," in *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pp. 53–61, IEEE, 2017.

[98] K. A. Safwan and F. Servant, "Decomposing the rationale of code commits: the software developer's perspective," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 397–408, 2019.

[99] E. Aghajani, C. Nagy, O. L. Vega-Márquez, M. Linares-Vásquez, L. Moreno, G. Bavota, M. Lanza, and D. C. Shepherd, "Software documentation: The practitioners' perspective," in *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, IEEE, 2020.

[100] E. AlOmar, M. W. Mkaouer, and A. Ouni, "Can refactoring be self-affirmed? an exploratory study on how developers document their refactoring activities in commit messages," in *2019 IEEE/ACM 3rd International Workshop on Refactoring (IWoR)*, pp. 51–58, IEEE, 2019.

[101] J. Zhi, V. Garousi-Yusifoğlu, B. Sun, G. Garousi, S. Shahnewaz, and G. Ruhe, "Cost, benefits and quality of software development documentation: A systematic mapping," *Journal of Systems and Software*, vol. 99, pp. 175–198, 2015.

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

[103] 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 (ICSME)*, pp. 451–455, IEEE, 2014.

[104] 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.

[105] 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*, pp. 279–290, ACM, 2014.

[106] 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)*, pp. 23–32, IEEE, 2013.

[107] 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)*, pp. 230–232, IEEE, 2013.

[108] 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*, pp. 43–52, ACM, 2010.

[109] 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.

[110] 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.

[111] 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*, pp. 101–110, ACM, 2011.

[112] 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*, pp. 35–44, IEEE, 2010.

[113] 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*, pp. 223–226, ACM, 2010.

[114] 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.

[115] 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)*, pp. 2073–2083, 2016.

[116] E. Murphy-Hill and A. P. Black, "Refactoring tools: Fitness for purpose," *IEEE Softw.*, vol. 25, no. 5, pp. 38–44, 2008.

[117] E. Murphy-Hill and A. P. Black, "Programmer-friendly refactoring errors," *IEEE Transactions on Software Engineering*, vol. 38, no. 6, pp. 1417–1431, 2012.

[118] E. Murphy-Hill and A. P. Black, "Breaking the barriers to successful refactoring: observations and tools for extract method," in *Proceedings of the 30th International Conference on Software Engineering*, pp. 421–430, ACM, 2008.

[119] X. Ge and E. Murphy-Hill, "Manual refactoring changes with automated refactoring validation," *36th International Conference on Software Engineering (ICSE)*, vol. 36, pp. 1095–1105, 2014.

[120] S. R. Foster, W. G. Griswold, and S. Lerner, "Witchdoctor: IDE support for real-time auto-completion of refactorings," in *Proceedings of the International Conference on Software Engineering*, pp. 222–232, 2012.

[121] L. Tahvildari and K. Kontogiannis, "A metric-based approach to enhance design quality through meta-pattern transformations," in *7th European Conference on Software Maintenance and Reengineering (CSMR)*, pp. 183–192, 2003.

[122] D. Dig, "A refactoring approach to parallelism," *IEEE software*, vol. 28, no. 1, pp. 17–22, 2011.

[123] Y. Kataoka, D. Notkin, M. D. Ernst, and W. G. Griswold, "Automated support for program refactoring using invariants," in *International Conference on Software Maintenance (ICSM)*, p. 736, IEEE Computer Society, 2001.

[124] W. F. Opdyke, "Refactoring object-oriented frameworks," 1992.

[125] M. Fokaefs, N. Tsantalis, E. Stroulia, and A. Chatzigeorgiou, "Jdeodorant: identification and application of extract class refactorings," in *2011 33rd International Conference on Software Engineering (ICSE)*, pp. 1037–1039, IEEE, 2011.

[126] M. Harman, S. A. Mansouri, and Y. Zhang, "Search-based software engineering: Trends, techniques and applications," *ACM Computing Surveys (CSUR)*, vol. 45, no. 1, p. 11, 2012.

[127] O. Seng, J. Stammel, and D. Burkhart, "Search-based determination of refactorings for improving the class structure of object-oriented systems," in *Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pp. 1909–1916, ACM, 2006.

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

[129] 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.

[130] M. Kessentini, W. Kessentini, H. Sahraoui, M. Boukadoum, and A. Ouni, "Design defects detection and correction by example," in *IEEE International Conference on Program Comprehension*, (DIRO, Université de Montréal, Canada), pp. 81–90, 2011.

[131] H. Kilic, E. Koc, and I. Cereci, "Search-based parallel refactoring using population-based direct approaches," in *International Symposium on Search Based Software Engineering*, pp. 271–272, Springer, 2011.

[132] M. Harman and L. Tratt, "Pareto Optimal Search Based Refactoring at the Design Level," in *Proceddings GECCO 2007*, (Department of Computer Science, King's College London, Strand, London, WC2R 2LS), pp. 1106–1113, 2007.

[133] A. Ouni, M. Kessentini, H. Sahraoui, and M. S. Hamdi, "Search-based refactoring: Towards semantics preservation," in *IEEE International Conference on Software Maintenance, ICSM*, (DIRO, Université de Montréal, Canada), pp. 347–356, 2012.

[134] M. Ó Cinnéide, L. Tratt, M. Harman, S. Counsell, and I. Hemati Moghadam, "Experimental assessment of software metrics using automated refactoring," in *Proceedings of the ACM-IEEE International Symposium on Empirical Software Engineering and Measurement - ESEM '12*, (School of Computer Science and Informatics, University College Dublin, Ireland), p. 49, 2012.

[135] E. Murphy-Hill, C. Parnin, and A. P. Black, "How we refactor, and how we know it," *IEEE Transactions on Software Engineering*, vol. 38, no. 1, pp. 5–18, 2012.

[136] M. Hall, N. Walkinshaw, and P. McMinn, "Supervised software modularisation," in *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, pp. 472–481, IEEE, 2012.

[137] G. Bavota, F. Carnevale, A. De Lucia, M. Di Penta, and R. Oliveto, "Putting the developer in-the-loop: an interactive GA for software re-modularization," in *International Symposium on Search Based Software Engineering*, pp. 75–89, Springer, 2012.

[138] Y. Lin, X. Peng, Y. Cai, D. Dig, D. Zheng, and W. Zhao, "Interactive and guided architectural refactoring with search-based recommendation," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 535–546, ACM, 2016.

[139] M. W. Mkaouer, M. Kessentini, S. Bechikh, K. Deb, and M. Ó Cinnéide, "Recommendation system for software refactoring using innovization and interactive dynamic optimization," *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering - ASE '14*, pp. 331–336, 2014.

[140] A. Ouni, M. Kessentini, H. Sahraoui, and M. S. Hamdi, "The use of development history in software refactoring using a multi-objective evolutionary algorithm," in *Proceedings of the 15th annual conference on Genetic and evolutionary computation*, pp. 1461–1468, 2013.

[141] O. Seng, J. Stammel, and D. Burkhart, "Search-based determination of refactorings for improving the class structure of object-oriented systems," in *Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pp. 1909–1916, ACM, 2006.

[142] P. Thongtanunam, R. G. Kula, A. E. C. Cruz, N. Yoshida, and H. Iida, "Improving code review effectiveness through reviewer recommendations," in *Proceedings of the 7th International Workshop on Cooperative and Human Aspects of Software Engineering*, pp. 119–122, ACM, 2014.

[143] X. Xia, D. Lo, X. Wang, and B. Zhou, "Accurate developer recommendation for bug resolution," in *2013 20th Working Conference on Reverse Engineering (WCRE)*, pp. 72–81, IEEE, 2013.

[144] J. Cohen, E. Brown, B. DuRette, and S. Teleki, *Best kept secrets of peer code review.* Smart Bear Somerville, 2006.

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

[146] P. C. Rigby, D. M. German, and M.-A. Storey, "Open source software peer review practices: a case study of the apache server," in *Proceedings of the 30th international conference on Software engineering*, pp. 541–550, ACM, 2008.

[147] C. Bird, D. Pattison, R. D'Souza, V. Filkov, and P. Devanbu, "Latent social structure in open source projects," in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pp. 24–35, ACM, 2008.

[148] P. C. Rigby and M.-A. Storey, "Understanding broadcast based peer review on open source software projects," in *2011 33rd International Conference on Software Engineering (ICSE)*, pp. 541–550, IEEE, 2011.

[149] A. Bosu and J. C. Carver, "Impact of peer code review on peer impression formation: A survey," in *2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, pp. 133–142, IEEE, 2013.

[150] A. Bosu and J. C. Carver, "How do social interaction networks influence peer impressions formation? a case study," in *IFIP International Conference on Open Source Systems*, pp. 31–40, Springer, 2014.

[151] O. Kononenko, O. Baysal, L. Guerrouj, Y. Cao, and M. W. Godfrey, "Investigating code review quality: Do people and participation matter?," in *2015 IEEE international conference on software maintenance and evolution (ICSME)*, pp. 111–120, IEEE, 2015.

[152] X. Yang, N. Yoshida, R. G. Kula, and H. Iida, "Peer review social network (person) in open source projects," *IEICE Transactions on Information and Systems*, vol. 99, no. 3, pp. 661–670, 2016.

[153] O. Baysal, O. Kononenko, R. Holmes, and M. W. Godfrey, "The influence of non-technical factors on code review," in *2013 20th Working Conference on Reverse Engineering (WCRE)*, pp. 122–131, IEEE, 2013.

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

[155] A. Bosu and J. C. Carver, "Impact of developer reputation on code review outcomes in oss projects: An empirical investigation," in *Proceedings of the 8th ACM/IEEE international symposium on empirical software engineering and measurement*, p. 33, ACM, 2014.

[156] A. Ouni, M. Kessentini, M. Ó Cinnéide, H. Sahraoui, K. Deb, and K. Inoue, "More: A multi-objective refactoring recommendation approach to introducing design patterns and fixing code smells," *Journal of Software: Evolution and Process*, vol. 29, no. 5, p. e1843, 2017.

[157] H. Wang, M. Kessentini, and A. Ouni, "Bi-level identification of web service defects," in *International Conference on Service-Oriented Computing*, pp. 352–368, Springer, 2016.

[158] A. Ghannem, G. El Boussaidi, and M. Kessentini, "On the use of design defect examples to detect model refactoring opportunities," *Software Quality Journal*, vol. 24, no. 4, pp. 947–965, 2016.

[159] B. Amal, M. Kessentini, S. Bechikh, J. Dea, and L. B. Said, "On the use of machine learning and search-based software engineering for ill-defined fitness function: a case study on software refactoring," in *International Symposium on Search Based Software Engineering*, pp. 31–45, Springer, Cham, 2014.

[160] D. Athanasopoulos, A. V. Zarras, G. Miskos, and V. Issarny, "Cohesion-Driven Decomposition of Service Interfaces Without Access to Source Code," *IEEE Transactions on Services Computing*, vol. 8, no. JUNE, pp. 1–18, 2015.

[161] D. Romano and M. Pinzger, "Analyzing the evolution of web services using fine-grained changes," in *IEEE International Conference on Web Services (ICWS)*, pp. 392–399, June 2012.

[162] A. Ouni, R. Gaikovina Kula, M. Kessentini, and K. Inoue, "Web service antipatterns detection using genetic programming," in *Proceedings of the 2015 on Genetic and Evolutionary Computation Conference*, GECCO'15, pp. 1351–1358, ACM, 2015.

[163] A. Ouni, M. Kessentini, K. Inoue, and M. O Cinneide, "Search-based web service antipatterns detection," *IEEE Transactions on Services Computing*, vol. PP, no. 99, 2015.

[164] H. Wang, M. Kessentini, T. Hassouna, and A. Ouni, "On the value of quality of service attributes for detecting bad design practices," in *Web Services (ICWS), 2017 IEEE International Conference on*, pp. 341–348, IEEE, 2017.

[165] J. F. Bard, *Practical bilevel optimization: algorithms and applications*, vol. 30. Springer Science & Business Media, 2013.

[166] F. Palma, N. Moha, G. Tremblay, and Y.-G. Guéhéneuc, "Specification and detection of soa antipatterns in web services," in *European Conference on Software Architecture*, pp. 58–73, Springer, 2014.

[167] J. L. O. Coscia, C. Mateos, M. Crasso, and A. Zunino, "Refactoring code-first web services for early avoiding wsdl anti-patterns: Approach and comprehensive assessment," *Science of Computer Programming*, vol. 89, pp. 374–407, 2014.

[168] M. P. Singh and M. N. Huhns, *Service-oriented computing - semantics, processes, agents.* Wiley, 2005.

[169] A. Rotem-Gal-Oz, *SOA Patterns.* Manning Publications, 2012.

[170] J. Král and M. Žemlička, “Crucial service-oriented antipatterns,” *International Journal On Advances in Software*, vol. 2, no. 1, pp. 160–171, 2009.

[171] N. Moha, F. Palma, M. Nayrolles, B. J. Conseil, Y.-G. Guéhéneuc, B. Baudry, and J.-M. Jézéquel, “Specification and detection of soa antipatterns,” in *Service-Oriented Computing*, pp. 1–16, Springer, 2012.

[172] S. R. Chidamber and C. F. Kemerer, “A metrics suite for object oriented design,” *IEEE Transactions on software engineering*, vol. 20, no. 6, pp. 476–493, 1994.

[173] E. Al-Masri and Q. H. Mahmoud, “The QWS dataset.” URL: https://qwsdata.github.io.

[174] D. Athanasopoulos and A. Zarras, “Fine-grained metrics of cohesion lack for service interfaces,” in *IEEE International Conference on Web Services (ICWS)*, pp. 588–595, July 2011.

[175] D. Sahin, M. Kessentini, S. Bechikh, and K. Deb, “Code-smell detection as a bilevel problem,” *ACM Trans. Softw. Eng. Methodol.*, vol. 24, no. 1, pp. 6:1–6:44, 2014.

[176] W. Candler and R. Townsley, “Linear two-level programming problem.,” *COMP. & OPER. RES.*, vol. 9, no. 1, pp. 59–76, 1982.

[177] E. Aiyoshi and K. Shimizu, “Hierarchical decentralized systems and its new solution by a barrier method.,” *IEEE Transactions on Systems, Man and Cybernetics*, no. 6, pp. 444–449, 1981.

[178] B. Colson, P. Marcotte, and G. Savard, “An overview of bilevel optimization,” *Annals of operations research*, vol. 153, no. 1, pp. 235–256, 2007.

[179] K. Deb and A. Sinha, “An efficient and accurate solution methodology for bilevel multi-objective programming problems using a hybrid evolutionary-local-search algorithm,” *Evolutionary computation*, vol. 18, no. 3, pp. 403–449, 2010.

[180] F. Legillon, A. Liefooghe, and E.-G. Talbi, “Cobra: A cooperative coevolutionary algorithm for bi-level optimization,” in *Evolutionary Computation (CEC), 2012 IEEE Congress on*, pp. 1–8, IEEE, 2012.

[181] A. Koh, “A metaheuristic framework for bi-level programming problems with multi-disciplinary applications,” in *Metaheuristics for Bi-level Optimization*, pp. 153–187, Springer, 2013.

[182] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, “A fast and elitist multiobjective genetic algorithm: Nsga-ii,” *IEEE transactions on evolutionary computation*, vol. 6, no. 2, pp. 182–197, 2002.

[183] M. Harman, S. A. Mansouri, and Y. Zhang, “Search-based software engineering: Trends, techniques and applications,” *ACM Computing Surveys (CSUR)*, vol. 45, no. 1, p. 11, 2012.

[184] F. Wilcoxon, S. Katti, and R. A. Wilcox, "Critical values and probability levels for the wilcoxon rank sum test and the wilcoxon signed rank test," *Selected tables in mathematical statistics*, vol. 1, pp. 171–259, 1970.

[185] F. Palma, N. Moha, G. Tremblay, and Y.-G. Guéhéneuc, "Specification and detection of soa antipatterns in web services," in *Software Architecture*, pp. 58–73, Springer, 2014.

[186] "Gan: A beginner's guide to generative adversarial networks." URL: https://skymind.ai/wiki/generative-adversarial-network-gan.

[187] Y. Lin, X. Peng, Y. Cai, D. Dig, D. Zheng, and W. Zhao, "Interactive and guided architectural refactoring with search-based recommendation," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 535–546, ACM, 2016.

[188] W. Mkaouer, M. Kessentini, A. Shaout, P. Koligheu, S. Bechikh, K. Deb, and A. Ouni, "Many-objective software remodularization using nsga-iii," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 24, no. 3, p. 17, 2015.

[189] M. W. Mkaouer, M. Kessentini, S. Bechikh, M. Ó. Cinnéide, and K. Deb, "On the use of many quality attributes for software refactoring: a many-objective search-based software engineering approach," *Empirical Software Engineering*, vol. 21, no. 6, pp. 2503–2545, 2016.

[190] I. H. Moghadam and M. O. Cinneide, "Automated refactoring using design differencing," in *2012 16th European Conference on Software Maintenance and Reengineering*, pp. 43–52, IEEE, 2012.

[191] G. Bavota, A. De Lucia, A. Marcus, R. Oliveto, and F. Palomba, "Supporting extract class refactoring in eclipse: The aries project," in *Proceedings of the 34th International Conference on Software Engineering*, pp. 1419–1422, IEEE Press, 2012.

[192] 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.

[193] M. W. Mkaouer, M. Kessentini, M. Ó. Cinnéide, S. Hayashi, and K. Deb, "A robust multi-objective approach to balance severity and importance of refactoring opportunities," *Empirical Software Engineering*, vol. 22, no. 2, pp. 894–927, 2017.

[194] D. Dig, J. Marrero, and M. D. Ernst, "Refactoring sequential java code for concurrency via concurrent libraries," in *Proceedings of the 31st International Conference on Software Engineering*, pp. 397–407, IEEE Computer Society, 2009.

[195] "The proposed refactoring tool." URL: https://sites.google.com/view/tse2020decision.

[196] M. Feathers, *Working Effectively with Legacy Code: WORK EFFECT LEG CODE _p1*. Prentice Hall Professional, 2004.

[197] L. Tokuda and D. Batory, “Evolving object-oriented designs with refactorings,” *Automated Software Engineering*, vol. 8, no. 1, pp. 89–120, 2001.

[198] E. R. Murphy-Hill and A. P. Black, “Why don’t people use refactoring tools?,” in *Proceedings of the 1st Workshop on Refactoring Tools (WRT ’07)*, pp. 60–61, 2007.

[199] L. Xiao, Y. Cai, R. Kazman, R. Mo, and Q. Feng, “Identifying and quantifying architectural debt,” in *Proceedings of the 38th International Conference on Software Engineering*, pp. 488–498, ACM, 2016.

[200] R. Moser, W. Pedrycz, and G. Succi, “A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction,” in *Proceedings of the 30th International Conference on Software Engineering*, pp. 181–190, ACM, 2008.

[201] L. C. Briand, J. Wust, S. V. Ikonomovski, and H. Lounis, “Investigating quality factors in object-oriented designs: an industrial case study,” in *Proceedings of the 1999 International Conference on Software Engineering (IEEE Cat. No. 99CB37002)*, pp. 345–354, IEEE, 1999.

[202] R. Shatnawi and W. Li, “An empirical assessment of refactoring impact on software quality using a hierarchical quality model,” *International Journal of Software Engineering and Its Applications*, vol. 5, no. 4, pp. 127–149, 2011.

[203] T. Caliński and J. Harabasz, “A dendrite method for cluster analysis,” *Communications in Statistics-theory and Methods*, vol. 3, no. 1, pp. 1–27, 1974.

[204] G. Xuan, W. Zhang, and P. Chai, “Em algorithms of gaussian mixture model and hidden markov model,” in *Proceedings 2001 International Conference on Image Processing (Cat. No. 01CH37205)*, vol. 1, pp. 145–148, IEEE, 2001.

[205] R. A. Redner and H. F. Walker, “Mixture densities, maximum likelihood and the em algorithm,” *SIAM review*, vol. 26, no. 2, pp. 195–239, 1984.

[206] A. Vargha and H. D. Delaney, “A critique and improvement of the CL common language effect size statistics of McGraw and Wong,” *Journal of Educational and Behavioral Statistics*, vol. 25, no. 2, pp. 101–132, 2000.

[207] M. Feathers, *Working Effectively with Legacy Code: WORK EFFECT LEG CODE _p1*. Prentice Hall Professional, 2004.

[208] J. Kerievsky, *Refactoring to Patterns*. Pearson Higher Education, 2004.

[209] R. Kazman, Y. Cai, R. Mo, Q. Feng, L. Xiao, S. Haziyev, V. Fedak, and A. Shapochka, “A case study in locating the architectural roots of technical debt,” in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 2, pp. 179–188, IEEE, 2015.

[210] J. Carriere, R. Kazman, and I. Ozkaya, “A cost-benefit framework for making architectural decisions in a business context,” in *2010 ACM/IEEE 32nd International Conference on Software Engineering*, vol. 2, pp. 149–157, IEEE, 2010.

[211] M. Kim, M. Gee, A. Loh, and N. Rachatasumrit, "Ref-finder: a refactoring reconstruction tool based on logic query templates," in *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pp. 371–372, ACM, 2010.

[212] D. Batory, J. N. Sarvela, and A. Rauschmayer, "Scaling step-wise refinement," *IEEE Transactions on Software Engineering*, vol. 30, no. 6, pp. 355–371, 2004.

[213] A. Ouni, M. Kessentini, H. Sahraoui, and M. Boukadoum, "Maintainability defects detection and correction: a multi-objective approach," *Automated Software Engineering*, vol. 20, no. 1, pp. 47–79, 2013.

[214] B. Du Bois, S. Demeyer, and J. Verelst, "Refactoring-improving coupling and cohesion of existing code," in *11th working conference on reverse engineering*, pp. 144–151, IEEE, 2004.

[215] I. H. Moghadam and M. Ó Cinnéide, "Code-imp: a tool for automated search-based refactoring," in *Proceedings of the 4th Workshop on Refactoring Tools*, pp. 41–44, ACM, 2011.

[216] Y. Zhang, G. Huang, X. Liu, W. Zhang, H. Mei, and S. Yang, "Refactoring android java code for on-demand computation offloading," in *ACM Sigplan Notices*, vol. 47, pp. 233–248, ACM, 2012.

[217] 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*, pp. 464–474, ACM, 2018.

[218] J. Yackley, M. Kessentini, G. Bavota, V. Alizadeh, and B. Maxim, "Simultaneous refactoring and regression testing: A multi-tasking approach," in *Proceedings of the 19th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM '19)*, pp. 216–227, 2019.

[219] 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*, pp. 80–91, 2018.

[220] E. Murphy-Hill and A. P. Black, "Refactoring tools: Fitness for purpose," *IEEE Software*, vol. 25, no. 5, pp. 38–44, 2008.

[221] G. Bavota, B. D. Carluccio, A. D. Lucia, M. D. Penta, R. Oliveto, and O. Strollo, "When does a refactoring induce bugs? an empirical study," in *12th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM*, pp. 104–113, 2012.

[222] M. Kim, T. Zimmermann, and N. Nagappan, "An empirical study of refactoringchallenges and benefits at microsoft," *Software Engineering, IEEE Transactions on*, vol. 40, pp. 633–649, July 2014.

[223] E. Murphy-Hill, C. Parnin, and A. P. Black, “How we refactor, and how we know it,” *IEEE Transactions on Software Engineering (TSE)*, vol. 38, no. 1, pp. 5–18, 2011.

[224] W. Mkaouer, M. Kessentini, A. Shaout, P. Koligheu, S. Bechikh, K. Deb, and A. Ouni, “Many-objective software remodularization using nsga-iii,” *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 24, no. 3, pp. 17:1–17:45, 2015.

[225] A. Alali, H. Kagdi, and J. I. Maletic, “What’s a typical commit? a characterization of open source software repositories,” in *Proc. 16th*, pp. 182–191, June 2008.

[226] V. Alizadeh, H. Fehri, and M. Kessentini, “Less is more: From multi-objective to mono-objective refactoring via developers knowledge extraction,” in *Proceedings of the 19th IEEE International Working Conference on Source Code Analysis and Manipulation SCAM ’19)*, pp. 181–192, 2019.

[227] A. Bachmann, C. Bird, F. Rahman, P. Devanbu, and A. Bernstein, “The missing links: Bugs and bug-fix commits,” in *Proc. 16th*, Nov. 2010.

[228] A. Bosu, J. C. Carver, C. Bird, J. Orbeck, and C. Chockley, “Process aspects and social dynamics of contemporary code review: insights from open source development and industrial practice at microsoft,” *IEEE Transactions on Software Engineering*, vol. 43, no. 1, pp. 56–75, 2017.

[229] M. Beller, A. Bacchelli, A. Zaidman, and E. Juergens, “Modern code reviews in open-source projects: Which problems do they fix?,” in *Proceedings of the 11th working conference on mining software repositories*, pp. 202–211, ACM, 2014.

[230] 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, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pp. 483–494, 2018.

[231] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, “A fast and elitist multiobjective genetic algorithm: Nsga-ii,” *IEEE Transactions on Evolutionary Computation*, vol. 6, no. 2, pp. 182–197, 2002.

[232] “Tool demo: Recommending refactorings via commit message analyis,” 2020. URL: https://sites.google.com/view/istrefcom.

[233] A. J. Ko, T. D. Latoza, and M. M. Burnett, “A practical guide to controlled experiments of software engineering tools with human participants,” *Empirical Software Engineering*, vol. 20, no. 1, pp. 110–141, 2015.

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

[235] 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*, pp. 483–494, ACM, 2018.

[236] 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*, pp. 337–342, ACM, 2014.

[237] 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.

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

[239] 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.

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

[241] "Interactive refactoring documentation bot demo," 2019. URL: https://sites.google.com/view/scam-2019.

[242] 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*, pp. 345–355, ACM, 2014.

[243] 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.

[244] G. Bavota and B. Russo, "A large-scale empirical study on self-admitted technical debt," in *Proceedings of the 13th International Conference on Mining Software Repositories*, pp. 315–326, 2016.

[245] 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*, 2017.

[246] E. Aghajani, C. Nagy, O. L. Vega-Márquez, M. Linares-Vásquez, L. Moreno, G. Bavota, and M. Lanza, "Software documentation issues unveiled," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pp. 1199–1210, IEEE, 2019.

[247] "Replication package," 2020. URL: https://sites.google.com/view/tse2020refactoringmodel/home.

[248] M. Codoban, S. S. Ragavan, D. Dig, and B. Bailey, "Software history under the lens: A study on why and how developers examine it," in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 1–10, IEEE, 2015.

[249] M. Hilton, N. Nelson, T. Tunnell, D. Marinov, and D. Dig, "Trade-offs in continuous integration: assurance, security, and flexibility," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pp. 197–207, 2017.

[250] S. Easterbrook, J. Singer, M.-A. Storey, and D. Damian, "Selecting empirical methods for software engineering research," in *Guide to advanced empirical software engineering*, pp. 285–311, Springer, 2008.

[251] Y. Tao, Y. Dang, T. Xie, D. Zhang, and S. Kim, "How do software engineers understand code changes? an exploratory study in industry," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, pp. 1–11, 2012.

[252] N. Tsantalis, A. Ketkar, and D. Dig, "Refactoringminer 2.0," *IEEE Transactions on Software Engineering*, 2020.

[253] P. K. Goyal and G. Joshi, "Qmood metric sets to assess quality of java program," in *2014 International Conference on Issues and Challenges in Intelligent Computing Techniques (ICICT)*, pp. 520–533, IEEE, 2014.

[254] T. Sharma and D. Spinellis, "A survey on software smells," *Journal of Systems and Software*, vol. 138, pp. 158–173, 2018.

[255] E. Oliveira, E. Fernandes, I. Steinmacher, M. Cristo, T. Conte, and A. Garcia, "Code and commit metrics of developer productivity: a study on team leaders perceptions," *Empirical Software Engineering*, vol. 25, no. 4, pp. 2519–2549, 2020.

[256] W. Maalej and H.-J. Happel, "Can development work describe itself?," in *2010 7th IEEE working conference on mining software repositories (MSR 2010)*, pp. 191–200, IEEE, 2010.

[257] S. Liu, C. Gao, S. Chen, N. L. Yiu, and Y. Liu, "Atom: Commit message generation based on abstract syntax tree and hybrid ranking," *IEEE Transactions on Software Engineering*, 2020.

[258] L. Y. Nie, C. Gao, Z. Zhong, W. Lam, Y. Liu, and Z. Xu, "Contextualized code representation learning for commit message generation," *arXiv preprint arXiv:2007.06934*, 2020.

[259] "Replication package for commit messages generation for composite changes," 2021. URL: https://sites.google.com/view/refacomgeneration/home.

[260] "Github api." URL:https://developer.github.com/v3/.

[261] B. Ghotra, S. McIntosh, and A. E. Hassan, "Revisiting the impact of classification techniques on the performance of defect prediction models," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1, pp. 789–800, IEEE, 2015.

[262] "Stanfordnlp." url:=https://stanfordnlp.github.io/stanza/tokenize.html#use-spacy-for-fast-tokenization-and-sentence-segmentation.

[263] M.-T. Luong and C. D. Manning, "Achieving open vocabulary neural machine translation with hybrid word-character models," *arXiv preprint arXiv:1604.00788*, 2016.

[264] J. Crego, J. Kim, G. Klein, A. Rebollo, K. Yang, J. Senellart, E. Akhanov, P. Brunelle, A. Coquard, Y. Deng, *et al.*, "Systran's pure neural machine translation systems," *arXiv preprint arXiv:1610.05540*, 2016.

[265] W. Chen, E. Matusov, S. Khadivi, and J.-T. Peter, "Guided alignment training for topic-aware neural machine translation," *arXiv preprint arXiv:1607.01628*, 2016.

[266] S. Murakami, M. Morishita, T. Hirao, and M. Nagata, "Ntt's machine translation systems for wmt19 robustness task," *arXiv preprint arXiv:1907.03927*, 2019.

[267] M.-T. Luong, H. Pham, and C. D. Manning, "Effective approaches to attention-based neural machine translation," *arXiv preprint arXiv:1508.04025*, 2015.

[268] A. Ouni, M. Kessentini, and H. Sahraoui, "Search-based refactoring using recorded code changes," in *2013 17th European Conference on Software Maintenance and Reengineering*, pp. 221–230, IEEE, 2013.

[269] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," in *Advances in neural information processing systems*, pp. 3104–3112, 2014.

[270] J. Gu, Z. Lu, H. Li, and V. O. Li, "Incorporating copying mechanism in sequence-to-sequence learning," *arXiv preprint arXiv:1603.06393*, 2016.

[271] D. Britz, A. Goldie, M.-T. Luong, and Q. Le, "Massive exploration of neural machine translation architectures," *arXiv preprint arXiv:1703.03906*, 2017.

[272] K. Chen, J. Wang, L.-C. Chen, H. Gao, W. Xu, and R. Nevatia, "Abc-cnn: An attention based convolutional neural network for visual question answering," *arXiv preprint arXiv:1511.05960*, 2015.

[273] J. Mao, W. Xu, Y. Yang, J. Wang, Z. Huang, and A. Yuille, "Deep captioning with multimodal recurrent neural networks (m-rnn)," *arXiv preprint arXiv:1412.6632*, 2014.

[274] A. LeClair, S. Jiang, and C. McMillan, "A neural model for generating natural language summaries of program subroutines," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pp. 795–806, IEEE, 2019.

[275] M. E. Peters, M. Neumann, M. Iyyer, M. Gardner, C. Clark, K. Lee, and L. Zettlemoyer, "Deep contextualized word representations," *arXiv preprint arXiv:1802.05365*, 2018.

[276] A. Radford, K. Narasimhan, T. Salimans, and I. Sutskever, "Improving language understanding by generative pre-training (2018)," *URL https://s3-us-west-2. amazonaws. com/openai-assets/research-covers/language-unsupervised/language_ understanding_paper. pdf*, 2018.

[277] J. Phang, T. Févry, and S. R. Bowman, "Sentence encoders on stilts: Supplementary training on intermediate labeled-data tasks," *arXiv preprint arXiv:1811.01088*, 2018.

[278] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[279] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in neural information processing systems*, pp. 5998–6008, 2017.

[280] Y. Zhu, R. Kiros, R. Zemel, R. Salakhutdinov, R. Urtasun, A. Torralba, and S. Fidler, "Aligning books and movies: Towards story-like visual explanations by watching movies and reading books," in *Proceedings of the IEEE international conference on computer vision*, pp. 19–27, 2015.

[281] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "Roberta: A robustly optimized bert pretraining approach," *arXiv preprint arXiv:1907.11692*, 2019.

[282] L. v. d. Maaten and G. Hinton, "Visualizing data using t-SNE," *Journal of machine learning research*, vol. 9, no. 11, pp. 2579–2605, 2008.

[283] J. M. Morse, *Mixed method design: Principles and procedures*, vol. 4. Routledge, 2016.

[284] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "Bleu: a method for automatic evaluation of machine translation," in *Proceedings of the 40th annual meeting on association for computational linguistics*, pp. 311–318, Association for Computational Linguistics, 2002.

[285] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," *arXiv preprint arXiv:1409.0473*, 2014.

[286] C. D. Manning, P. Raghavan, and H. Schütze, *Introduction to information retrieval.* Cambridge university press, 2008.

[287] J. Cohen, "Weighted kappa: nominal scale agreement provision for scaled disagreement or partial credit.," *Psychological bulletin*, vol. 70, no. 4, p. 213, 1968.

[288] J. Vig, "A multiscale visualization of attention in the transformer model," *arXiv preprint arXiv:1906.05714*, 2019.

[289] H. Yanagimto, K. Hashimoto, and M. Okada, "Attention visualization of gated convolutional neural networks with self attention in sentiment analysis," in *2018 International Conference on Machine Learning and Data Engineering (iCMLDE)*, pp. 77–82, IEEE, 2018.

[290] X. Xia, D. Lo, X. Wang, and X. Yang, "Who should review this change?: Putting text and file location analyses together for more accurate recommendations," in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 261–270, IEEE, 2015.

[291] Y. Yu, H. Wang, G. Yin, and T. Wang, "Reviewer recommendation for pull-requests in github: What can we learn from code review and bug assignment?," *Information and Software Technology*, vol. 74, pp. 204–218, 2016.

[292] R. Almhana, W. Mkaouer, M. Kessentini, and A. Ouni, "Recommending relevant classes for bug reports using multi-objective search," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, (New York, NY, USA), pp. 286–295, ACM, 2016.

[293] A. Ouni, M. Kessentini, K. Inoue, and M. O. Cinnéide, "Search-based web service antipatterns detection," *IEEE Transactions on Services Computing*, vol. 10, no. 4, pp. 603–617, 2017.

[294] A. A. Keller, *Multi-Objective Optimization in Theory and Practice II: Metaheuristic Algorithms.* Bentham Science Publishers, 2019.

[295] M. T. Emmerich and A. H. Deutz, "A tutorial on multiobjective optimization: fundamentals and evolutionary methods," *Natural computing*, vol. 17, no. 3, pp. 585–609, 2018.

[296] K. Deb and S. Gupta, "Understanding knee points in bicriteria problems and their implications as preferred solution principles," *Engineering optimization*, vol. 43, no. 11, pp. 1175–1204, 2011.

[297] L. Rachmawati and D. Srinivasan, "Multiobjective evolutionary algorithm with controllable focus on the knees of the pareto front," *IEEE Transactions on Evolutionary Computation*, vol. 13, no. 4, pp. 810–824, 2009.

[298] A. Bosu, J. C. Carver, C. Bird, J. Orbeck, and C. Chockley, "Process aspects and social dynamics of contemporary code review: Insights from open source development and industrial practice at microsoft," *IEEE Transactions on Software Engineering*, vol. 43, no. 1, pp. 56–75, 2016.

[299] M. B. Zanjani, H. Kagdi, and C. Bird, "Automatically recommending peer reviewers in modern code review," *IEEE Transactions on Software Engineering*, vol. 42, no. 6, pp. 530–543, 2015.

[300] X. Yang, R. G. Kula, N. Yoshida, and H. Iida, “Mining the modern code review repositories: A dataset of people, process and product,” in *Proceedings of the 13th International Conference on Mining Software Repositories*, pp. 460–463, ACM, 2016.

[301] R. R. Jackson, C. M. Carter, and M. S. Tarsitano, “Trial-and-error solving of a confinement problem by a jumping spider, portia fimbriata,” *Behaviour*, vol. 138, no. 10, pp. 1215–1234, 2001.