

Universidade Federal de Campina Grande
Centro de Engenharia Elétrica e Informática
Coordenação de Pós-Graduação em Ciência da Computação

Characterizing Refactoring-Inducing Pull Requests

Flávia Coelho

Proposta de Tese submetida à Coordenação do Curso de Pós-Graduação em Ciência da Computação da Universidade Federal de Campina Grande – Campus I, como parte dos requisitos necessários para obtenção do grau de Doutora em Ciência da Computação.

Área de Concentração: Ciência da Computação
Linha de Pesquisa: Metodologia e Técnicas da Computação

Dr. Tiago Lima Massoni
Dr. Everton Leandro Galdino Alves
(Orientadores)

Campina Grande, Paraíba, Brasil
©Flávia Coelho, 18/02/2021

Resumo

O desenvolvimento baseado em *pull* constitui uma estratégia prática para Revisão de Código Moderna (RCM), uma vez que os desenvolvedores podem contribuir para melhorias no código tais como edições de refatoramento. A caracterização de *pull requests* que induzem a refatoramento fornece uma nova visão para tratar *pull requests* em estudos e práticas de RCM. Nós definimos um *pull request* indutor de refatoramento como aquele em que refatoramentos são realizados como um resultado da revisão de código ou por ações espontâneas do desenvolvedor do *pull request*. Para preencher essa lacuna, nós mineramos dados de revisão de código a partir do GitHub, examinamos características das edições de refatoramento e investigamos diferenças/similiaridades entre *pull requests* indutores e não-indutores de refatoramento, explorando aspectos técnicos de revisão de código. Especificamente, nós propomos dois estudos sobre *merged pull requests*: (i) caracterizar refatorações em *pull requests* indutores de refatoramento e (ii) comparar *pull requests* indutores e não-indutores de refatoramento. O primeiro estudo investiga quantitativamente as edições de refatoramento efetuadas nos commits de *pull requests* indutores de refatoramento. O segundo estudo envolve triangulação de métodos para análise: (1) uma comparação entre *pull requests* indutores e não-indutores de refatoramento, examinando dados quantitativos relativos a alterações, revisão de código e refatoramento, apoiada por agrupamento (*clustering*) e aprendizagem de regras de associação e (2) uma inspeção manual dos comentários de revisão, aplicando análise de conteúdo. Em uma amostra de 4.525 *pull requests*, identificamos 46,4% de *pull requests* indutores de refatoramento, compreendendo edições de refatoramento de 39 tipos que diminuem gradualmente ao longo dos commits e operam em níveis alto, médio e baixo. Descobrimos que os *pull requests* indutores de refatoramento diferem significativamente dos não-indutores de refatoramento em termos do número de linhas alteradas no código, número de arquivos alterados, número de comentários de revisão, quantidade de discussão e tempo para *merge*. No entanto, não encontramos evidências estatísticas de que o número de revisores está relacionado ao incentivo à refatoramento. A próxima etapa é explorar comentários de revisão. O principal impacto desta pesquisa

constitui uma expansão do conhecimento sobre revisão do código contemporânea em *pull requests*, enquanto fornece implicações acionáveis para pesquisadores, profissionais e desenvolvedores de ferramentas.

Abstract

Pull-based development constitutes a practical strategy for Modern Code Review (MCR) since developers can contribute to code improvements such as refactoring edits. Characterizing refactoring-inducing pull requests provides a novel view to treat pull requests in MCR studies and practice. We define a refactoring-inducing pull request as one in which refactorings are performed either as a result of the code reviewing or by spontaneous actions carried out by the pull request developer. To fill this gap, we mine code review data from GitHub, examine characteristics of refactoring edits, and investigate differences/similarities between refactoring-inducing and non-refactoring-inducing pull requests by exploring technical aspects of code review. Specifically, we propose two studies on merged pull requests: (i) characterizing refactorings in refactoring-inducing pull requests and (ii) comparing refactoring-inducing and non-refactoring-inducing pull requests. The first study quantitatively investigates the refactoring edits over commits in refactoring-inducing pull requests. The second study involves methods triangulation for analysis: (1) a comparison between refactoring-inducing and non-refactoring-inducing pull requests by examining quantitative data concerning changes, code review, and refactorings, supported by clustering and Association Rule Learning (ARL) and (2) a manual inspection of review comments, by applying content analysis. In a sample of 4,525 pull requests, we identified 46.4% of refactoring-inducing pull requests comprising refactoring edits from 39 types that gradually decrease over the commits, and operate in high, medium, and low levels. We found that refactoring-inducing pull requests significantly differ from non-refactoring-inducing ones in terms of code churn, number of changed files, number of review comments, length of discussion, and time to merge. However, we found no statistical evidence that the number of reviewers is related to refactoring-inducement. The next step is to explore the review comments. The main impact of this research constitutes an expansion of knowledge on the contemporary pull request code review while providing actionable implications to researchers, practitioners, and tool builders.

Contents

1	Introduction	1
1.1	Motivation	1
1.2	Problem Statement	5
1.3	Objectives	6
1.4	Research Questions	6
1.5	Implications of Research	7
1.6	Document Structure	8
2	Background	9
2.1	Refactoring	9
2.1.1	Identification of Candidates for Refactoring	10
2.1.2	Application of Refactoring	12
2.1.3	Refactoring Detection	13
2.2	Modern Code Review	15
2.3	Git-based Development and Pull Requests	17
2.4	Unsupervised Learning	22
2.4.1	Clustering	22
2.4.2	Association Rule Learning	25
2.5	Concluding Remarks	33
3	Related Work	34
3.1	Mining of Code Repositories	34
3.2	Characterization of Refactorings throughout Code Evolution	36
3.3	Characterization of Code Review	37

3.4	Concluding Remarks	38
4	Characterizing Refactoring Edits in Refactoring-Inducing Pull Requests	39
4.1	Research Design	39
4.1.1	Mining of Merged Pull Requests	40
4.1.2	Refactoring Detection	44
4.1.3	Mining of Code Review Data	46
4.1.4	Data Analysis	50
4.1.5	Validity and Reliability	53
4.2	Results and Discussion	53
4.2.1	How Common are Refactoring-Inducing Pull Requests?	53
4.2.2	What Refactoring Types often Take Place in Pull Requests?	54
4.2.3	How are the Refactoring Edits Characterized?	60
4.3	Limitations	62
4.4	Concluding Remarks	63
5	Comparing Refactoring-Inducing and non-Refactoring-Inducing Pull Requests	64
5.1	Research Design	64
5.1.1	Clustering	66
5.1.2	Association Rule Learning	70
5.1.3	Data Analysis	72
5.1.4	Validity and Reliability	74
5.2	Results and Discussion	75
5.2.1	Clustering	75
5.2.2	Association Rule Learning	79
5.2.3	Statistical Testing of Hypotheses	83
5.3	Limitations	91
5.4	Concluding Remarks	92

6	Next Steps	93
6.1	Preliminary Conclusions	93
6.2	Planning of the Manual Inspection of Review Comments	94
6.3	Towards “Strict” Refactoring-Inducing Pull Requests	96
6.4	Concluding Remarks	99
7	Work Plan	100
7.1	Research Schedule	100
7.2	Risks	102
7.3	Concluding Remarks	102
8	Concluding Remarks	103
A	Initial Investigations on Refactoring and Modern Code Review	121
B	Strategy for Mining Squashed and Merged Pull Requests from GitHub	124

List of Symbols

API – *Application Programming Interface*

ASF – *Apache Software Foundation*

AST – *Abstract Syntax Tree*

ARL – *Association Rule Learning*

CLES – *Common-Language Effect Size*

CROP – *Code Review Open Platform*

DBCV – *Density-Based Clustering Validation index*

DVCS – *Distributed Version Control System*

FP – *Frequent Pattern*

HSD – *Honestly Significant Difference*

IDEs – *Integrated Development Environments*

IQR – *InterQuartile Range*

MCR – *Modern Code Review*

OPTICS – *Ordering Points To Identify the Clustering Structure*

OSS – *Open-Source Software*

p-value – *Probability value*

REST – *REpresentational State Transfer*

SD – *Standard Deviation*

SHA-1 – *Secure Hash Algorithm 1*

SPLab/UFCG – *Software Practices Laboratory*

UI – *User Interface*

URL – *Uniform Resource Locator*

List of Figures

1.1	Neil Brown’s tweet on February 13, 2018	2
2.1	The commit message #141112 of the Eclipse’s repository egit on Gerrit (a) and the respective RefactoringMiner’s output (b)	14
2.2	A commit from pull request #1033 of the Apache’s repository hadoop- ozone on GitHub	17
2.3	Examples of Git branches	19
2.4	An overview of the Git pull request in GitHub	19
2.5	An overview of Git pull request review in GitHub	20
2.6	A partial diff output (reviewing feature) of committed changes into the pull request #1888 from Apache’s repository drill in GitHub	20
2.7	A partial diff output (differentiating feature) of committed changes into the pull request #1888 from Apache’s repository drill in GitHub	21
2.8	Example of core-point, core-distance, and reachability-distance in the scenario of the OPTICS clustering	25
2.9	Example of a OPTICS reachability plot	26
2.10	Example of one-hot encoding	28
2.11	Example of transactions	28
2.12	Example of the FP-tree formation	30
2.13	Example of the bottom-up approach for generation of frequent itemsets	31
4.1	Design of the characterization study of refactoring edits	40
4.2	Creation and merge dates of the final sample’s pull requests	50
4.3	Refactoring edits in the refactoring-inducing pull requests	54
4.4	Number of detected refactoring types by refactoring-inducing pull request	56

4.5	Distribution of refactoring types in the refactoring-inducing pull requests from the second until the sixth commit	58
4.6	Commits in refactoring-inducing pull requests	60
4.7	Scatterplot of number of refactorings in relation to number of commits	61
5.1	Design of the comparative study	65
5.2	OPTICS's reachability plot	76
5.3	Distribution of selected features by cluster	77
6.1	A refactoring-inducing pull request (PR #2010 from Apache's book- keeper repository), illustrating initial commits ($c_1 - c_3$) and subsequent commits ($c_4 - c_9$).	97
6.2	Empirical research design	98
B.1	An overview of the pull request #1807 events in GitHub	124

List of Tables

2.1	Refactoring types supported by RefactoringMiner 2.0 in September 2019	16
2.2	A summary of a few code review aspects raised from empirical studies .	18
2.3	Relationship between antecedent and consequent according to the lift value	27
2.4	Example of conditional pattern base, conditional FP-tree, and the respective generated frequent pattern rules	32
4.1	Search results for merged pull requests from Apache’s non-archived Java repositories in GitHub	41
4.2	Top 5 repositories containing non-squashed and merged pull requests .	42
4.3	Top 5 repositories containing squashed and merged pull requests	43
4.4	A RefactoringMiner output example	44
4.5	Output of RefactoringMiner execution in Apache’s repositories	45
4.6	RefactoringMiner output, at commit and pull request levels, for Apache’s repositories	46
4.7	Top 5 repositories containing squashed and merged pull requests in the sample	46
4.8	Top 5 repositories containing non-squashed and merged pull requests in the sample	47
4.9	Pull requests’ attributes selected for mining	48
4.10	A summary of the final sample from mining of Apache’s merged pull requests in GitHub	49
4.11	Top 5 Apache’s repositories in the final sample	49

4.12	Proposed mapping, adapted from [2][1][89], between levels of operation and refactoring types	51
4.13	Validity and reliability countermeasures for the characterization study of refactorings	53
4.14	Refactoring types detected in the pull requests	55
4.15	Number of detected refactoring kinds in the refactoring-inducing pull requests	59
4.16	Number of detected refactorings from the second until the sixth commit	61
4.17	Levels of detected refactorings in the refactoring-inducing pull requests	62
5.1	Output from the selection strategy of OPTICS's input parameters . . .	70
5.2	Proposed one-hot encoding for binning of features	71
5.3	Validity and reliability procedures	74
5.4	Clustering output	76
5.5	High-level characterization of the clusters	78
5.6	Number of associations obtained by cluster	79
5.7	Association rules selected by manual inspection of ARL output (A1–A15 from refactoring-inducing pull requests and A16–128 from non-refactoring-inducing pull requests)	79
7.1	Research schedule (in quarters)	100
A.1	A summary of the preliminary investigation results	123
B.1	GraphQL API's HeadRefForcePushedEvent fields for pull request #1807	125
B.2	The recovered history of the pull request #1807's commits	125

List of Source Codes

2.1	A simple Java source code	9
2.2	A simple refactored Java source code	10
2.3	A simple example of a Java source code with feature envy	11
2.4	A simple example of a Java source code with no feature envy after a move method refactoring	11
4.1	A simple Java method	57
4.2	A simple Java refactored method	57
5.1	Fragment of the Python script for clustering, highlighting the procedure to find the best input parameters to OPTICS execution	69

Chapter 1

Introduction

1.1 Motivation

Modern Code Review (MCR) is driven by reviewing source code changes in a lightweight, tool-assisted, and asynchronous manner [26]. Through the years, we perceive the transition from formal software inspection toward MCR, in both open source and industrial software development. In this context, finding defects, the main objective of software inspection [50], became a limited operation of inspection [68], whereas regular change-based reviewing, in which code improvements are also valuable, became an essential practice in the MCR scenario [26; 107].

Code changes may comprise new features, bug fixes, and general maintenance constituting potential opportunities for refactoring edits [96] that, in turn, form a significant part of the changes [27; 123]. Empirical evidence reveals differences between refactoring-dominant changes and other ones. For instance, reviewing bug fixes is more time-consuming than reviewing refactorings, since they preserve code behavior [110]. Given that the nature of the change significantly affects the code review effectiveness [103], because it directly influences how reviewers perceive the changes, the provision of suitable resources for assisting the reviewing process becomes useful. In this perspective, claims from developers may provide insights for improvements in MCR, such as Neil Brown’s tweet¹ on February 13, 2018 (Figure 1.1).

¹That tweet (available in <https://t.ly/PHk1>), shared by professor Tiago Massoni in our first technical meeting, inspired our preliminary investigations regarding refactorings and MCR.

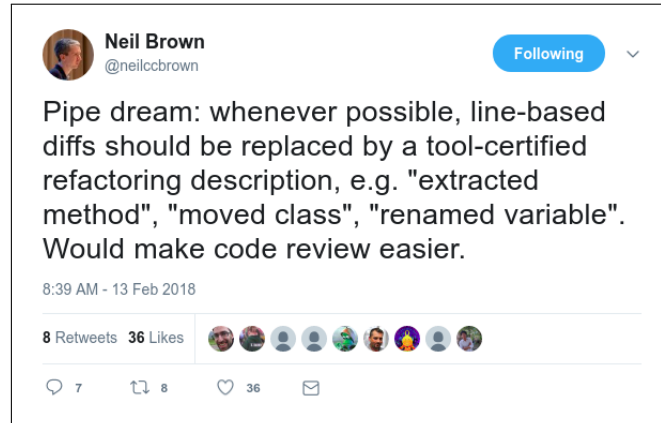


Figure 1.1: Neil Brown’s tweet on February 13, 2018

In this sense, empirical studies have focused on the need for understanding composite changes [27; 47; 61; 85; 124], the demand for refactoring-aware code review tools [23; 28; 41; 56; 128], and the identification of refactoring-related patterns from the change history [22; 37; 83; 90; 95; 84; 96; 98; 130], concomitantly to research on code reviewing characterization [29; 34; 35; 65; 72; 80; 106; 107; 108; 109; 114].

Composite changes (also referred to as tangled changes [85]) encompass code modifications that address diverse development issues in a single commit, for instance, in the presence of floss refactoring, in turn, characterized by implementing refactoring edits while developing other programming activities, such as a new feature [89]. In order to decompose complex changes for MCR, works have proposed techniques for summarising regions of changes [27], semantically decomposing cohesive change-slices [124], reorganizing code changes according to the automated refactoring-related commit policy [85], and expanding the decomposition for interactive use [61]. In brief, those studies aim the support for code reviewing in presence of refactoring edits as elements of composite changes. Moreover, there is empirical evidence on the positive effects of change decomposition for code reviewing [47].

A *refactoring-aware review* happens when a reviewer is informed about the refactorings applied to code, for instance, by getting details from the output of a refactoring detection tool or from a commit message. It can help a reviewer improve his/her understanding, reduce reviewing time, and increase its accuracy [28; 128]. In this respect, there are approaches that, at code reviewing, highlight types of refactoring [56;

23], and inspect clone refactorings [41].

Concerning the characterization of refactorings performed throughout code evolution, research has addressed the peculiarities of the manual and automated refactoring edits over time [90], the relationship between different categories of code changes and specific refactoring types [96], and the commit-level assessment of contemporary refactoring practices [130]. Studies have investigated how developers document refactorings during the code evolution [22], and the impacts of refactorings on merge conflicts [83]. Recently, studies have explored the reason for systems' architecture degradation after refactoring edits [95], the commit-level description of refactorings over time [37], the motivations behind refactorings at the pull request level [98], and the characterization of the intents and evolution of refactorings during code reviewing [84].

Empirical studies to characterize the MCR process, in both open-source and industrial settings, have been developed with the purpose of investigating technical aspects of reviewing [109; 107; 108; 29; 65; 34; 114], proposing recommendations from OSS code review experience for industry [106], investigating factors leading to useful code review [35], exploring circumstances that contribute to code review quality [72], and identifying general code review patterns in the pull-based development [80]. These characterization studies are relevant because MCR is critical in repository-based software development, such as in *agile* and *continuous delivery* approaches. Agile software development comprises practices guided by principles established in the sense of software development driven by change and based on collaboration [12]; whereas continuous delivery is a chain of processes (e.g., build, review, test, packaging) that produce deployable software releases from code changes, in a fast, automated, and replicable manner [64].

In that scenario, Git-based software repositories have increasingly been adopted [11]. Particularly, Git *pull requests* play an important role in MCR because they provide a well-defined and collaborative code reviewing. Through pull requests, the code follows a review process in which developers (called *reviewers*) may suggest improvements before merging the code to the main branch. *Branching* and *merging* are features of Git-based development. Branching allows the creation of development lines without interfering with the main development line, and merging provides the procedure to inte-

grate the code developed in a branch into the main development line [40]. Pull requests may be refactoring-inducing, reflecting that design and quality concerns are addressed over the commits, before the merge. From the context, we realized a gap concerning the characterization of refactoring edits and technical aspects of code reviewing in the pull-based development in line with our definition of refactoring-inducing pull request (Section 1.2).

It is worth clarifying that we propose an investigation at the pull request level because we understand a pull request as a complete scenario for exploring code reviewing practices in a well-defined scope of development, which allows us to go beyond an investigation at the commit level. For instance, at the pull request level, we can obtain a global comprehension of contributions to the original code, in terms of both commits and reviewing-related aspects, such as reviewers' comments. This conception is mainly inspired by empirical evidence that the pull-based development model is associated with larger numbers of contributions, at least, in relation to patch-based code contribution tools, such as mailing lists [136], and by findings from Pantiuchina et al. that indicate discussions at the pull request level as one of the motivating factors to refactorings [98].

By better understanding how refactoring-inducing pull requests work, we might assist researchers, practitioners, and tool builders in face of challenges of the MCR in pull-based development, such as responsiveness (i.e., timely feedback)[60] and code change size [82]. Accordingly, this thesis proposal aims to provide a systematical and practical understanding of the refactoring-inducing code review process in the pull-based development.

In this direction, we address the gap through mining Git-based software repositories for investigating how common are refactoring-inducing pull requests, what are the typical refactoring types and their respective properties (in a characterization study of refactoring edits in refactoring-inducing pull requests, Chapter 4), and what are the main differences between refactoring-inducing pull requests and non-refactoring-inducing ones, considering metrics related to changes (e.g., number of changed files), particular characteristics of reviewing (e.g., review comments), and refactoring edits (in a comparative study between refactoring-inducing and non-refactoring-inducing pull

requests, Chapter 5). Specifically, we propose a comparative study between refactoring-inducing and non-refactoring-inducing pull requests, driven by method triangulation because it may help to comprehensively understand different aspects of a phenomenon [45]. As a result, we might systematically understand refactoring-inducing pull requests, so producing a more accurate characterization. Note that those studies are complementary, intending to characterize both refactorings and code reviewing-related aspects in the refactoring-inducement context (Definition 1).

It is worth mentioning that this thesis proposal has evolved from initial investigations regarding refactorings and MCR [15] (summarized in Appendix A), developed in order to get a better understanding of the topic and plan the research design. We selected *case study* as our main empirical strategy due to the possibility of systematically investigating data collected from real software repositories intending to identify aspects influencing a process and relationships between variables [112].

1.2 Problem Statement

Pull-based development, as implemented in GitHub, constitutes an useful strategy for MCR practice. Through pull requests, developers can contribute to source code improvements, such as triggering refactoring edits [98]. This scenario leads to a question: what characterizes pull requests that induce refactorings?

Filling this knowledge gap states an extension in the understanding of MCR in the pull-based development model and the provision of support for practical recommendations and further research. In this sense, this thesis proposal investigates the technical aspects of refactorings and code review in GitHub pull requests, in light of the following definition of a refactoring-inducing pull request².

Definition 1. *Refactoring-inducing means the occurrence of refactoring edits performed as part of the changes in commits within a GitHub pull request. Accordingly, suppose an initial commit with 0 or more refactorings in a pull request. Then, developers add new subsequent commits with refactorings whether or not as a result of the*

²This designation was proposed by professor Nikolaos Tsantalis (Concordia University, Canada) in one of our technical meetings, so initially inspiring the current thesis proposal.

reviewing process. Let $U = \{u_1, u_2, \dots, u_w\}$, a set of repositories in GitHub. Each repository u_q , $1 \leq q \leq w$, has a set of pull requests $P(u_q) = \{p_1, p_2, \dots, p_m\}$ over time. Each pull request p_j , $1 \leq j \leq m$, has a set of commits $C(p_j) = \{c_1, c_2, \dots, c_n\}$. A refactoring-inducing pull request is that in which $\exists c_k \mid R(c_k) \neq \emptyset$, where $R(c_k)$ denotes the set of refactoring edits performed in commit c_k , where $1 < k \leq n$.

1.3 Objectives

The main goal of this thesis proposal is to establish an in-depth understanding of refactoring-inducing pull requests. Given that, the following objectives are raised:

- Composing datasets of code review data and detected refactorings in merged pull requests;
- Identifying the differences between refactoring-inducing and non-refactoring-inducing pull requests;
- Identifying aspects of code review that influence refactoring-inducement in pull requests; and
- Characterizing refactoring-inducing pull requests.

1.4 Research Questions

This thesis proposal addresses the following research questions:

- RQ₁ How common are refactoring-inducing pull requests?
- RQ₂ What refactoring types often take place in pull requests?
- RQ₃ What characterize these refactoring edits?
- RQ₄ How do refactoring-inducing pull requests compare to non-refactoring-inducing ones?
- RQ₅ How do review comments influence refactoring-inducing pull requests?

Specifically, we investigate RQ₁, RQ₂, and RQ₃ in the characterization study of refactorings in refactoring-inducing pull requests (Chapter 4), whereas RQ₄ and RQ₅ in the comparative study between refactoring-inducing and non-refactoring-inducing pull requests (Chapter 5).

1.5 Implications of Research

By characterizing refactoring-inducing pull requests, we can potentially advance the understanding of code review process at the pull request level and expand the research regarding the contemporary code review through recommendations for practitioners, suggestions for the development of efficient code review tools, and guidelines for future research.

In this context, our findings provide two actionable implications for tool builders and practitioners, when proposing refactoring-awareness and code quality-awareness at reviewing time in the GitHub pull-based development model, as argued as follows. Overall, those features might improve the practice of code contribution at the pull request level.

- Refactoring-aware pull requests. By adding refactoring-awareness to the Git-based review board, we suggest make traceable the refactorings performed over the commits in a pull request. Hence, reviewers could provide feedback on refactorings promptly, and concentrate efforts on other aspects of the changes, for instance, searching for collateral effects of the refactorings and proposing specific tests on those edits; and
- Code quality-aware pull requests. Particularly, our research reinforces the need for impact analysis on pull requests, as previously argued by Gousios et al. [60]. Accordingly, we propose adding code quality-awareness to the Git-based review board, as a feature to assist practitioners in assessing the code quality of refactored code at reviewing time. As a result, they could analyze the impact of changes, so preventing issues in the future.

For future research, we propose investigations on code quality and fault-proneness

in refactoring-inducing pull requests, beyond an exploration regarding the impacts of refactoring-inducing pull requests related to merging (since this thesis proposal focuses on merged pull requests). Replication studies on different projects that follow the pull-based development model are welcome in sense of developing a theory on refactoring-inducing pull requests. We also register that our mined data and findings could assist researchers who investigate the practical impacts of code reviewing at the pull-based development since they can reuse our research methods and datasets, publicly available [18].

1.6 Document Structure

The next chapters are organized as follows. In Chapter 2, we provide a conceptual background regarding refactorings, MCR, Git-based development and pull requests, and unsupervised learning. In Chapter 3, we discuss the characterization studies related to this thesis proposal. Then, we present the research design, results and respective discussion, and limitations of the studies proposed for characterizing refactoring-inducing pull requests: a characterization study of refactorings in refactoring-inducing pull requests (Chapter 4), and a comparative study between refactoring-inducing and non-refactoring-inducing pull requests (Chapter 5). We propose the next steps in Chapter 6 and suggest a work plan to elucidate the research steps in chronological order in Chapter 7, followed by concluding remarks in Chapter 8, and appendices.

Chapter 2

Background

This chapter addresses the main concepts that contextualize this thesis proposal: the fundamentals of refactoring (Section 2.1), modern code review (Section 2.2), Git-based development and pull requests (Section 2.3), and unsupervised learning (Section 2.4).

2.1 Refactoring

As software evolves to meet new requirements, its source code becomes more complex. Throughout this process, design and quality deserve attention [70]. Design comprises the internal software structure, while quality addresses the functional requirements and the structural aspects such as maintainability, extensibility, and reusability [119].

For that, restructuring operations, originally named refactorings by Opdyke and Johnson [93], are performed to directly favor design and the quality of object-oriented software, whereas preserving its external behavior. To exemplify, suppose a developer, inspired by Fowler [52], implemented the Java code 2.1.

```
1 // A simple Hello World in Java!
2 public class SimpleHelloWorld {
3     public static void main(String ... args) {
4         String name = "developer.", hw = "Hello World!";
5         System.out.println("Hi, I'm " + name);
6         System.out.println("This is my " + hw);
7     }
8 }
```

Source Code 2.1: A simple Java source code

After compiling and running the code, the developer realizes that it is possible to create a method that displays the printing details in standard output, encompassing the last two lines of the code. He/she does a transformation, so that the external behavior (the program output) remains unchanged, as in listing 2.2.

```
1 // A simple Hello World in Java!
2 public class SimpleHelloWorld {
3     public static void main(String ... args) {
4         String name = "developer.", hw = "Hello World!";
5         printDetails(name, hw);
6     }
7     //this is a simple example of extract method
8     static void printDetails(String name, String hw) {
9         System.out.println("Hi, I'm " + name);
10        System.out.println("This is my " + hw);
11    }
12 }
```

Source Code 2.2: A simple refactored Java source code

After compiling and running the restructured code, the developer notes that the program's output remains unchanged. Thus, he/she applied a refactoring called *Extract Method*, which objective is to extract a method from a code snippet, so that the name of the method indicates its purpose. With this refactoring, developers often aim at improve code readability, achieved from an internal restructuring of the code, thereby supporting its maintainability.

A general recommendation is that refactorings should be performed in a systematic and structured manner based on specific steps to (1) identify candidates for refactoring, and (2) perform the refactorings [52; 92], as described in Subsections 2.1.1 and 2.1.2. Complementarily, developers may consult the applied refactorings into code, supported by refactoring detection tools, when performing a few development tasks, as explained in Subsection 2.1.3.

2.1.1 Identification of Candidates for Refactoring

The decision on whether performing a refactoring is often based on the presence of *bad smells*. Bad smells are signs of deeper problematic situations [30], so potential

candidates for refactoring in a code. For instance, consider one of the main problems for object-oriented design: *Feature Envy*, exemplified in listing 2.3. This bad smell arises when a method (`printEarthGreeting()`) is a member of a class (`HelloUniverse`), however, its responsibilities are more similar to the responsibilities of another class (`HelloEarth`).

```
1 // A simple Hello Universe in Java!
2 class HelloUniverse{
3     private String greeting = "Hello Universe!";
4     public void printGreetings() {
5         System.out.println(greeting);
6         HelloEarth earth = new HelloEarth();
7         printEarthGreeting(earth);
8     }
9     public void printEarthGreeting(HelloEarth earth) {
10        System.out.println(earth.getGreeting());
11    }
12 }
13 // A simple Hello Earth in Java!
14 class HelloEarth{
15     private String greeting = "Hello Earth!";
16     public String getGreeting() {
17         return greeting;
18     }
19 }
```

Source Code 2.3: A simple example of a Java source code with feature envy

This situation does not aid software maintainability, because it reduces cohesion while increasing coupling between the classes. A possible solution to address this issue is to move `printEarthGreeting()` from class `HelloUniverse` to class `HelloEarth`, applying the refactoring *Move Method* [52], as in listing 2.4.

```
1 // A simple Hello Universe in Java!
2 class HelloUniverse{
3     private String greeting = "Hello Universe!";
4     public void printGreetings() {
5         System.out.println(greeting);
6         HelloEarth earth = new HelloEarth();
7         earth.printEarthGreeting();
8     }
9 }
```



```
10 // A simple Hello Earth in Java!
11 class HelloEarth{
12     private String greeting = "Hello Earth!";
13     public String getGreeting() {
14         return greeting;
15     }
16     public void printEarthGreeting() {
17         System.out.println(getGreeting());
18     }
19 }
```

Source Code 2.4: A simple example of a Java source code with no feature envy after a move method refactoring

The identification of candidates for refactoring is a time-consuming activity, even when supported by tools [81]. For this thesis proposal, it is only relevant to be aware that developers utilize their own techniques or support tools during this task.

2.1.2 Application of Refactoring

Experts and researchers recommend refactoring in small steps, even when in the case of floss refactoring (including the restructuring and other programming activities, e.g., a new feature) [89]. These edits can be performed manually or supported by tools.

When manual, the transformations are time-consuming and susceptible to errors [23; 46]. For instance, when manually renaming a variable, a programmer needs to review all references to that variable, apply the refactoring, and then run a test to validate the behavior of the post-refactoring code [118].

Modern *Integrated Development Environments* (IDEs), such as *Eclipse*¹, *IntelliJ IDEA*², and *Netbeans*³ include automatic refactoring engines, which execution comprises the following phases:

1. the developer chooses the type of refactoring to apply;

¹<https://eclipse.org>

²<https://jetbrains.com/idea/>

³<https://netbeans.org>

2. the engine checks whether the restructuring meets the preconditions for the type of refactoring (e.g., no conflicts between identifiers is a precondition to rename a variable), and the external behavior of the software remains unchanged after refactoring; and
3. if so, the engine restructures the source code.

For this thesis proposal, it is only pertinent to be conscious that developers refactor code manually, supported by IDEs, or a mix of both.

2.1.3 Refactoring Detection

Refactoring detection consists of the automatic identification of refactoring types applied to the source code, for assisting development tasks such as studies on code evolution [96], understanding of changes [101], and code review [23; 56]. In practice, the refactoring detection is supported by tools based on different techniques, such as predicate logic in RefFinder [101], heuristics based on static analysis and code similarity in RefDiff [117], and *Abstract Syntax Tree* (AST)-based associations in RefactoringMiner⁴ [128; 127].

To illustrate the tool-supported refactoring detection, Figure 2.1 (a) shows a commit message⁵ informing on rename refactorings, and Figure 2.1 (b) displays the respective list of refactorings identified by RefactoringMiner, indicating the types of refactorings applied to the commit. This suggests, for example, that a reviewer could run the refactoring detection tool for obtaining complementary information to the ones indicated in the commit message Figure 2.1 (a), so getting details regarding the rename refactorings applied Figure 2.1 (b).

The effectiveness measure of a detection technique is *accuracy*, calculated from the *precision* and *recall*, according to Equations (2.1) and (2.2).

$$precision = \frac{|TP|}{|TP| + |FP|} \quad (2.1)$$

⁴<https://github.com/tsantalis/RefactoringMiner>

⁵<https://git.eclipse.org/r/#/c/141112/>

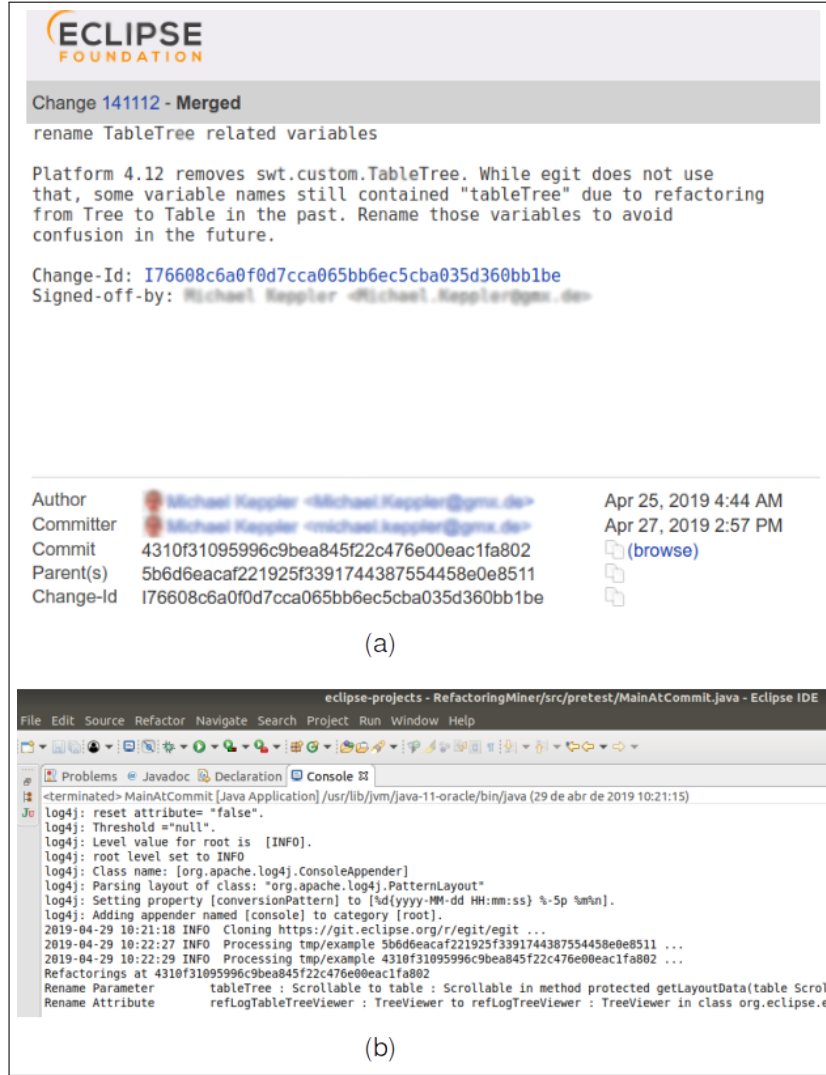


Figure 2.1: The commit message #141112 of the Eclipse’s repository egit on Gerrit (a) and the respective RefactoringMiner’s output (b)

$$recall = \frac{|TP|}{|TP| + |FN|} \quad (2.2)$$

where:

$|TP|$ = (true positive) number of detected refactorings,

$|FP|$ = (false positive) number of instances without refactoring, detected as refactoring; and

$|FN|$ = (false negative) number of undetected (or missing) refactorings.

False positives and false negatives impact tasks that depend on refactoring de-

tection, since they may respectively lead to wrong operations (e.g., when a reviewer requires a code change based on a detected but non-existent refactoring) and incomplete operations (e.g., when a reviewer does not require an additional code test due to an undetected refactoring).

From this perspective, RefactoringMiner is currently considered a state-of-the-art refactoring detection tool (precision of 99.6% and recall of 94%) [127]. It is implemented as a Java *Application Programming Interface* (API) to detect refactorings applied to in the history of a Java project [9], which running may be simplified in Equation (2.3).

$$f(a, b) = \text{list of applied refactorings} \quad (2.3)$$

where:

- a and b are revisions of a project developed in Java (i.e., a commit and its parent in the history of commits in a Git-based repository [3]);
- *list of applied refactorings* reports the types of refactoring edits carried out between revisions a and b , according to Table 2.1.

In this thesis proposal, we selected RefactorinMiner 2.0, a version available in September 2019, for refactoring detection purposes.

2.2 Modern Code Review

MCR, as defined by Bacchelli and Bird [26], consists of a lightweight code review (in opposition to the formal code inspection specified by Fagan [50]), tool-assisted, asynchronous (i.e., with no needs of face-to-face meetings), and driven by reviewing code changes. In practice, code review yields a positive impact on software quality in both open-source and industrial development scenarios [87]. This resource is supported by platforms such as GitHub [6], Gerrit [2], Phabricator [8], and Review Board [10].

Essentially, the code review process is a manual examination of the code changes (*patch*), submitted by a developer (author), by another developer (reviewer) - or reviewers. The process usually proceeds the following workflow, as illustrated in Figure

Table 2.1: Refactoring types supported by RefactoringMiner 2.0 in September 2019

<i>Refactoring types</i>
Extract Attribute, Class, Interface, Method, Subclass, Superclass, Variable
Inline Method, Parameter, Variable
Rename Attribute, Class, Method, Parameter, Variable
Move Attribute, Class, Method, Source Folder
Pull Up Attribute, Method
Push Down Attribute, Method
Extract and Move Method
Change Package (Move, Rename, Split, Merge)
Parameterize Variable
Move and Rename Attribute, Class, Method
Replace Variable with Attribute
Replace Attribute (with Attribute)
Merge Attribute, Parameter, Variable
Split Attribute, Parameter, Variable
Change Attribute Type, Parameter Type, Return Type, Variable Type
Move and Inline Method

2.2⁶, until the changes are accepted or discarded:

1. an author submits the code changes – the added lines 117–120 in the file `UnRegisterEndpointTask.java`, in turn, component of a PULL REQUEST’s commit;
2. the reviewer(s) – REVIEWER – inspects the code changes highlighted by code differentiating tools (*diff tool*) – DIFF OUTPUT;
3. the reviewer(s) can raise issues regarding specific lines, through threaded comments – REVIEW COMMENT; and
4. the author decides how to deal with the reviewers’ comments [114] – e.g., the comment ”done” indicates that the author agrees with the reviewer’s suggestion.

⁶Example of a diff output and comments during the code review process of a GitHub’s pull request, available in <https://github.com/apache/hadoop-ozone/pull/1033>

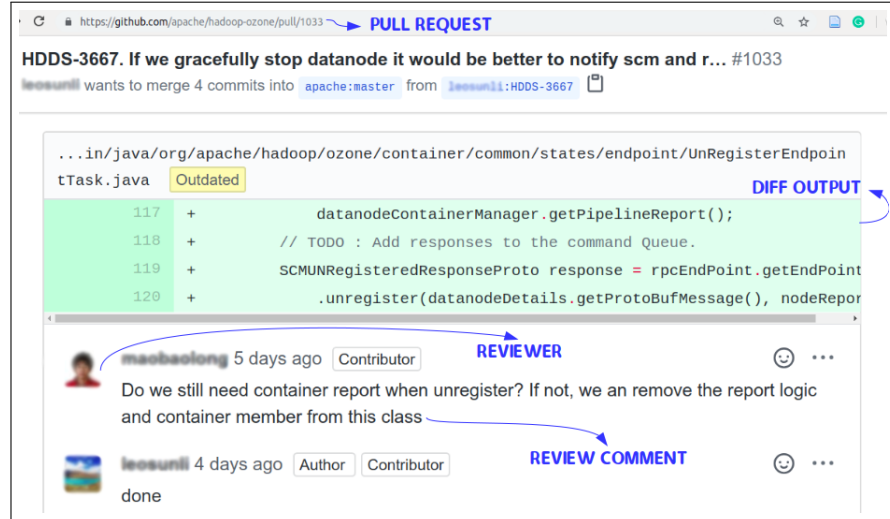


Figure 2.2: A commit from pull request #1033 of the Apache’s repository hadoop-ozone on GitHub

Empirical studies have investigated code review efficiency and effectiveness in order to understand the practice, elaborate recommendations, and develop improvements. Together, these works share a set of useful code review aspects, a few summarized in Table 2.2, for further investigation.

For this thesis proposal, it is essential to know the fundamentals of code reviewing in the context of pull requests on GitHub, in turn, described as follows.

2.3 Git-based Development and Pull Requests

Git, a *Distributed Version Control System* (DVCS), has been embraced by a large community of developers and organizations, in both open-source and industrial software development scenarios, as results from the 2020 Stack Overflow development survey [19], which indicates that more than 82% of developers use GitHub – a collaborative development platform built upon Git. GitHub has presented a frequent growth in terms of the number of developers (more than 40 million) that contribute to more than 2.9 million organizations around the world, in accordance with the last GitHub survey report [16]. Given that, we consider the Git-based development as implemented in GitHub.

Table 2.2: A summary of a few code review aspects raised from empirical studies

<i>Code review aspect</i>	<i>Empirical study(ies)</i>
change description	[35; 123]
code churn (number of changed lines)	[29; 72; 107; 126]
length of discussion	[72; 87; 107; 126]
number of changed files	[35; 72]
number of commits	[87; 108]
number of people in the discussion	[72]
number of resubmissions	[72; 107]
number of review comments	[31; 87; 107]
number of reviewers	[107; 115]
time to merge	[59; 65]

As a DVCS, Git supplies collaborative development through mirroring of repositories to the developers' computers. Each repository is composed of the files and folders of a project, including a full history of all changes applied to the files [40]. This change history is structured like a linked-list of snapshots in time, denominated commits that, in turn, are uniquely identified by 40 hexadecimal characters (e.g., 0a66da598a986971a77900442e20025ebfc56e9d⁷) calculated by the SHA-1 hash algorithm [91].

The commits are organized into multiple lines of development, named branches. Each repository has a main (or master) branch, so each additional line of development is constituted by a copy of the repository, which can be modified without altering the main branch. Figure 2.3⁸ illustrates the idea behind the Git branches, as implemented in the GitHub, displaying two lines of development created after the forking (copying) of the master-branch. As result, a developer can alter the repository's content to submit a new feature, via a new-feature-branch, and to fix a bug, via a bug-fix-branch, without altering the repository's content in the master-branch.

⁷A commit of an Apache repository, available in <https://git.io/JUBG5>

⁸A figure inspired in <https://guides.github.com/activities/hello-world/>

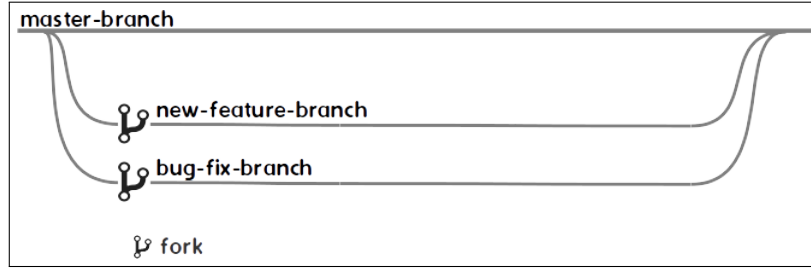


Figure 2.3: Examples of Git branches

After forking a Git branch, a developer can implement contributions (i.e., changes) through commits, and then, open a pull request to submit them for discussion. At that moment, it occurs the code review in line with the MCR process - the focus of investigations of this thesis proposal. In this setting, as shown in Figures 2.4 and 2.5⁹, a developer (author) forks the main repository branch (master-branch), creating a new branch (pr-branch), to make changes (a set of n commits, i.e., c_1, c_2, \dots, c_n) and submit them through a pull request (status *open pull request*). Next, the submitted commits are reviewed (CODE REVIEW) by one or more reviewers, resulting in either adding new commits to the pull request (CODE UPDATE) or closing the pull request (status *closed pull request*) by the author.

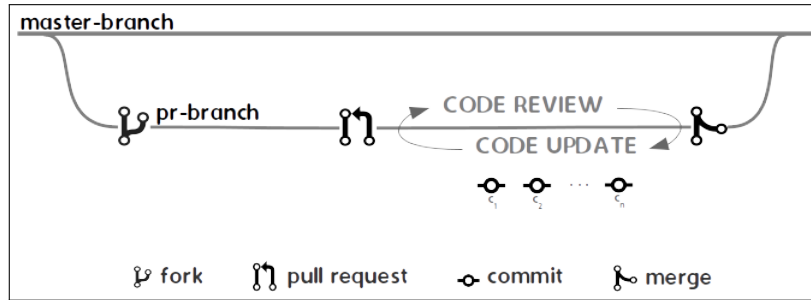


Figure 2.4: An overview of the Git pull request in GitHub

Figure 2.5 emphasizes the MCR process inside an open pull request. At that point, any developer with reading access to the repository can review the pull request. In this scenario, as exemplified in Figure 2.6 and 2.7, reviewers can submit comments (REVIEW COMMENT) while reviewing (REVIEWING) the proposed changes, based

⁹Both figures are inspired in <https://t.ly/EXH9>

on a diff output (detailed in Figure 2.7) that highlights the CHURN (number of added lines + number of deleted lines), and the changed lines (DELETED LINE (-) in red and ADDED LINE (+) in green) in each CHANGED FILE of a commit.

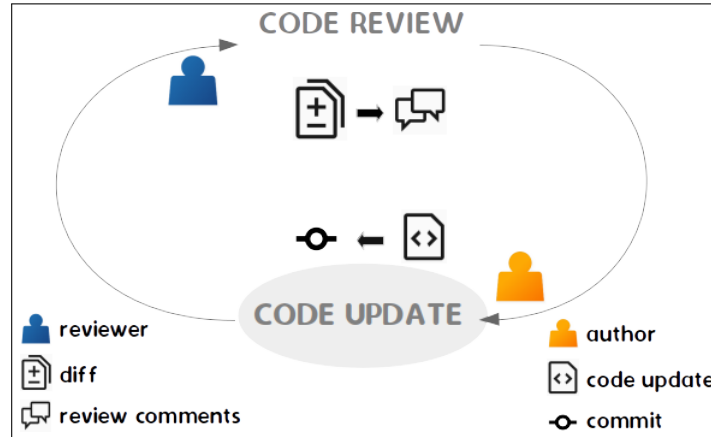


Figure 2.5: An overview of Git pull request review in GitHub

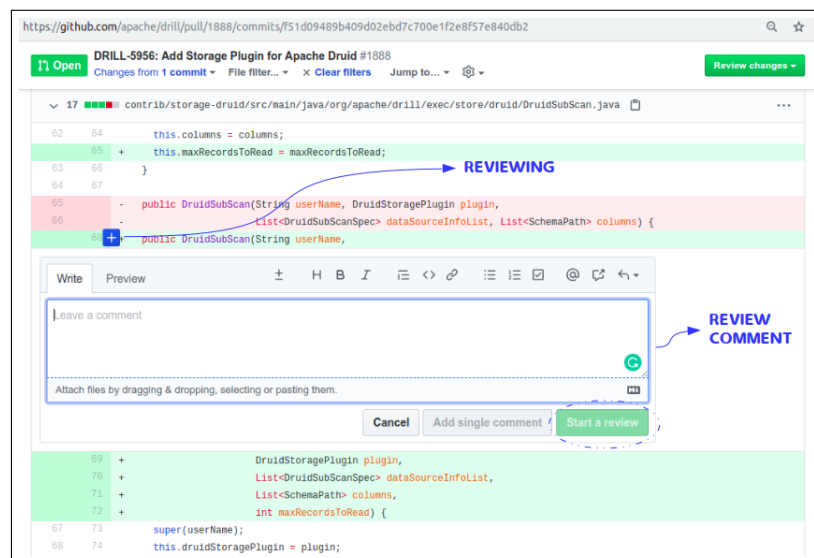


Figure 2.6: A partial diff output (reviewing feature) of committed changes into the pull request #1888 from Apache’s repository drill in GitHub

A reviewer can leave one of three types of feedback: *approve* agreeing with the merge of the proposed changes, *request changes* demanding for new changes before the merge, or *comment* submitting comments. As a result, the author and other contributors can

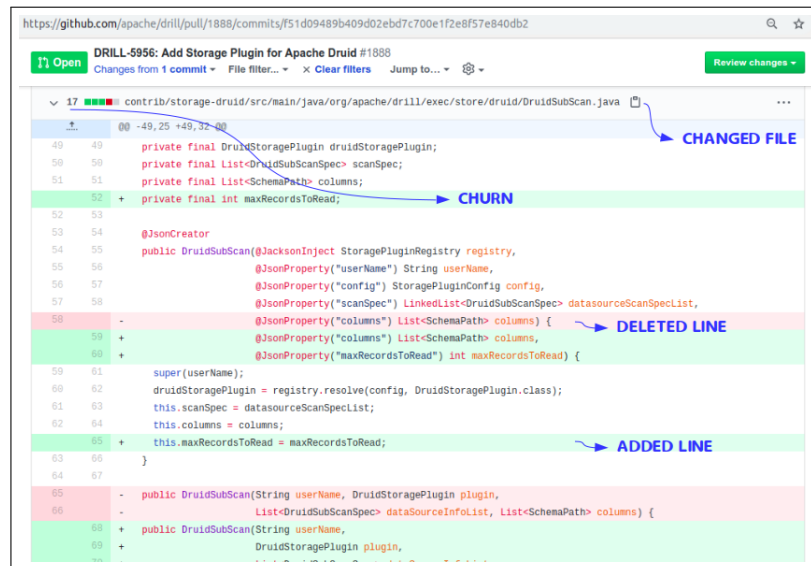


Figure 2.7: A partial diff output (differentiating feature) of committed changes into the pull request #1888 from Apache’s repository drill in GitHub

answer the review through general comments (also called pull request comments). In this way, pull requests provide a valuable environment for discussion.

After a review, the author can implement the changes that deal with the reviewer’s comments – a process that lasts until there are no new review submissions, and the mergeability requirements are met. These requirements consist of a customized set of checks, performed before merging a pull request, such as demanding a specified number of reviews before the merge. In terms of the Git merge in GitHub, there are three options:

- *merge pull request* that merges all commits into a *merge commit* and adds it into the main branch;
- *squash and merge* that squashes the commits into a single commit and merges it into the main branch; or
- *rebase and merge* that adds all commits onto the main branch without a merge commit.

The first option is useful to deal with high commits volume in branches, the second one is worthwhile to handle a lot of small commits in the branches, typically useful

in open-source projects, and the third one is useful in case of need to retain the full details of all commits after merge [55]. Once merged (status *merged pull request*), the pull requests are searchable, that is, supply the search for code changes history, except in case of the "squash and merge" option.

Concretely, pull requests are a method for providing fast turnaround and decreased time to integrate contributions into software development [59]. For this thesis proposal, GitHub's pull requests denote a contemporary infrastructure from where it is feasible getting real-life review data.

2.4 Unsupervised Learning

Unsupervised learning is an approach for modeling the structure of a dataset by looking for non-obvious patterns without external guidance [33]. Usually, the process follows this workflow [131]:

- selecting features for modeling in line with the research context;
- feature engineering by transforming the selected features into proper formats for modeling;
- choosing and running an appropriate algorithm based on the context and constraints of research; and
- practical interpreting results, by analyzing the meaningfulness of the algorithm's output aligned with the research context.

Specifically, to the context of this thesis proposal, unsupervised learning through clustering and Association Rule Learning (ARL) plays a proper role in the comparison between refactoring-inducing and non-refactoring-inducing pull requests detailed in Chapter 5.

2.4.1 Clustering

Clustering is a process for discovering natural groupings in datasets [20], for instance, grouping pull requests by performed refactoring type. For composing groups (*clusters*),

clustering algorithms apply a similarity measure to data points. A data point means a set of measurements on a single observation in a dataset, such as the values of the repository's name, pull request number, commit SHA, refactoring type, and details of one refactoring detected in a pull request. A similarity measure is critical to constitute meaningful clusters, that is, which denote high similarity among members within a given cluster and low similarity among clusters [69]. As an example, distance metrics such as *Euclidean distance* may be used to calculate the shortest distance between any two data points, assisting a clustering algorithm in the grouping the closer data points in clusters [66]. Euclidean distance measures the length of a segment connecting two points in an n-dimensional space, as Equation 2.4 [25].

$$d(x, y) = \left(\sum_{i=1}^n |x_i - y_i|^2 \right)^{\frac{1}{2}} \quad (2.4)$$

The **selection of features** is driven by the context of modeling; for instance, selecting of the features number of reviewers, number of review comments, and length of discussion when clustering pull requests by considering reviewing-related aspects. **Feature engineering** concerns the transformation of features into appropriate formats for modeling. For instance, raw counts of the number of review comments might span several orders of magnitude and present outliers, influencing the modeling. Thus, it is possible to apply quantile binning to equally portion the data, e.g., using quartiles (four quarters). That transformation maps a sequence of continuous numbers to discrete ones, furthering the identification of more patterns by learning algorithms [134].

There are several categories of **clustering algorithms**, based on different strategies to form clusters. Nevertheless, we concentrate effort in detailing a density-based clustering algorithm, denominated *Ordering Points To Identify the Clustering Structure* (OPTICS) [24], since it has presented a middle time complexity $O(n \log n)$, adequacy to arbitrary shapes of data, low sensibility to the sequence of input data, noise (data points not belonging to any cluster, which aid to restrict data overfitting [105]) and outliers (data points that deviate from other ones), and no requirement to preset the number of cluster.

OPTICS produces a linear ordering of the density-based clustering structure of the

data. Thus, a cluster is composed of a region of higher density data points than ones outside of the cluster, and noise data points have a lower density than the density of data points in any clusters. For each data point of a density-based cluster, the neighborhood of distance ε must comprise at least a minimum number of data points. Empirical evidence has determined satisfying results when regarding values between 10 and 20 for ε [24]. OPTICS operates by computing an infinite number of distance ε_i such that $0 \leq \varepsilon_i \leq \varepsilon$, and storing the *order* of processing of the data points, *core-distance*, and *reachability-distance*. Properly, clusters are constituted of closest points, that is, those with low reachability-distance to their nearest neighbors. In this context, OPTICS requires the presetting of two parameters: ε specifying the maximum distance to consider, and *min_points* indicating the minimum number of points needed to comprise a cluster.

In order to clarify, firstly, it is worth knowing the *core-point* definition. A point p is a core-point if at least *min_points* are located within its ε -neighborhood. A core-distance is the smallest distance required to recognize a point as a core-point. The reachability-distance between a point p and q is the maximum value between the core-distance of p and the distance between p and q [24]. Figure 2.8¹⁰ shows an example in which p is a core-point because 10 points are found in its ε -neighborhood ($\varepsilon = 5$), in a core-distance 2, whereas the reachability-distance between points p and q is 4.

An output OPTICS consists of a linear ordering of points arranged visually through a reachability plot that, in turn, may be explored to identify the clusters. In a reachability plot, as illustrated in Figure 2.9, the reachability-distances (y-axis) are plotted for each data point (x-axis). The reachability-distance of a point comprises the distance of this point to the set of its predecessors [24]. From this, points belonging to a cluster have a low reachability-distance to their nearest neighbors, whereas clusters can be visually identified as valleys (e.g., green points) as from the plot. Colors work as cluster labels, so colored points belong to clusters while black ones denote noise. Hence, four clusters are visually noticed from the plot.

The quality of density-based and arbitrarily shaped clusters can be assessed based on indexes, such as *Density-Based Clustering Validation* (DBCV) [88]. DBCV com-

¹⁰A figure inspired in [24].

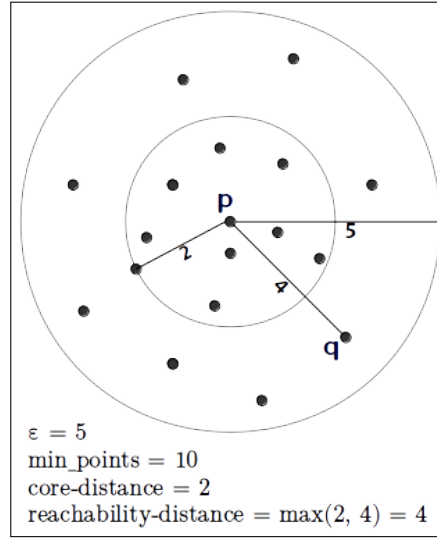


Figure 2.8: Example of core-point, core-distance, and reachability-distance in the scenario of the OPTICS clustering

putes the density of data points and evaluates the within- and between-cluster connectedness of clustering output, generating values in the range $[-1, 1]$ so that greater values mean better clustering. Lastly, the **interpretation of results** comprises a practical analysis of the meaningfulness of the obtained clusters in line with the research context.

2.4.2 Association Rule Learning

Association Rule Learning (ARL) figures out rules that denote non-obvious relationships between variables in large datasets, for instance, that changes comprising a high number of deleted lines and added lines tend to involve a high number of changed files. Let $I = \{i_1, i_2, \dots, i_n\}$, a set of n binary attributes (*items*) and $D = \{t_1, t_2, \dots, t_m\}$, a set of m transactions (*dataset*), in which each transaction in D consists of items in I . Thus, an association rule $\{X\} \rightarrow \{Y\}$ indicates the co-occurrence of the tuples $\{X\}$ (*antecedent*) and $\{Y\}$ (*consequent*), where $\{X\}, \{Y\} \subseteq I$, $\{X\} \cap \{Y\} = \emptyset$ [21]. In this context, *support* (Equation 2.5) expresses the number of transactions in D that supports an association rule, so expressing its statistical significance.

$$\text{supp}(\{X\} \rightarrow \{Y\}) = \frac{\text{frequency}(\{X\}, \{Y\})}{n} \quad (2.5)$$

in the interval $[0, \infty]$. In practical terms, conviction 1 denotes that antecedent and consequent are completely unrelated, while conviction ∞ determines logical implications, in which the confidence is 1 [36].

$$conv(\{X\} \rightarrow \{Y\}) = \frac{1 - supp(\{Y\})}{1 - conf(\{X\} \rightarrow \{Y\})} \quad (2.8)$$

Table 2.3: Relationship between antecedent and consequent according to the lift value

<i>Lift</i>	<i>Meaning</i>
0	No association
< 1	The occurrence of the antecedent has negative effect on the occurrence of the consequent and vice versa
> 1	The two occurrences are dependent on one another, and the association rules are useful

As for the steps of the ARL workflow, **feature selection** also depends on the problem context, but in terms of **feature engineering**, it is required to apply any encoding technique, such as *one-hot encoding*, for uniquely identifying the selected features. The one-hot encoding uses a group of bits to represent mutually exclusive categories, so each bit on represents a category [134]. For instance, consider a simple dataset of pull requests composed of the following features: each pull request is distinguished by a number (1 – 5), repository’s name to which it belongs (example-one or example-two), and associated labels (improvement, bug-fix, or new-feature). As shown in Figure 2.10, the dataset comprises eight pull request numbers, two distinct repository names, and three labels. Especially, by using hot-encoding, the repository’s names are identified by 0 1 (example-one) and 1 0 (example-two), while the labels are recognized by 0 0 1 (new-feature), 0 1 0 (bug-fix), and 1 0 0 (improvement). Thus, the pull request number #5, labeled as ‘improvement’, belongs to repository example-one.

From the previous example, it is feasible to identify five items, $I = \{example-one, example-two, improvement, bug-fix, new-feature\}$, and four distinct transactions as exhibited in Figure 2.11, $D = \{\{example-two, improvement\}, \{example-one, bug-$

Number	Repository	Label
1	0 1	1 0 0
2	1 0	0 1 0
3	1 0	0 0 1
4	0 1	0 1 0
5	0 1	1 0 0
6	1 0	0 1 0
7	1 0	0 0 1
8	0 1	1 0 0

Figure 2.10: Example of one-hot encoding

fix}, {*example-one*, *new-feature*}, {*example-two*, *bug-fix*}}. It follows that after feature engineering, the input data are prepared to the next step of the workflow: selecting and running an ARL algorithm.

transaction	example-one	example-two	improvement	bug-fix	new-feature
1	0	1	1	0	0
2	1	0	0	1	0
3	1	0	0	0	1
4	0	1	0	1	0

Figure 2.11: Example of transactions

The ARL process comprises the successive phases, namely: discovering of all sets of frequent itemsets that occur frequently (*frequent itemsets*) in the dataset, based on the minimum support level (Phase 1), and creating strong association rules from the most frequent itemsets, based on the minimum confidence level (Phase 2) [78]. In Phase 1, **ARL algorithms** require as input the minimum support level for an itemset be considered frequent. Phase 2 output is constituted by a set of association rules that meet both minimum support and confidence, and that can be searchable for the highest values of lift and conviction in order to make decisions. For instance, a minimum

support of 80 for the association rule $\{\text{low number of review comments} \rightarrow \text{low number of reviewers}\}$ denotes that both itemsets $\{\text{number of review comments}\}$ and $\{\text{number of reviewers}\}$ occur at least 80 times in a dataset of reviewing-related attributes in GitHub pull requests, whereas 80% of the pull requests that have a low number of review comments also have a low number of reviewers, for a minimum confidence level of 80. From that, a researcher might exclusively select association rules with lift > 1 and conviction > 1 for further investigation.

It is worth clarifying that a lower support level influences the ARL algorithms performance since a larger number of itemsets will be considered, while higher values result in fewer frequent itemsets. Concerning the confidence level, lower values cause association rules that are not very accurate, while higher ones will generate a fewer number of association rules [78].

Performance is the main issue surrounding ARL algorithms since Phase 1 requires searching $2^{|I|}$ sets [58]. Traditional algorithms, such as *Apriori* [120], perform multiple scans over the full dataset in order to generate *candidate itemsets* as from the dataset's tuples. At the end of a scan, it is checked if the support for a candidate itemset reaches the minimum support (defined by the user). Alternatively, *Frequent Pattern (FP)-growth* [62] stands out due to the use of an internal structure, named *FP-tree*, that gives the algorithm a high performance. Next, the FP-growth algorithm is detailed, supported by an illustrative example¹¹, based on the data from Figures 2.10 and 2.11, that is, $I = \{\text{example-one}, \text{example-two}, \text{improvement}, \text{bug-fix}, \text{new-feature}\}$, $D = \{\{\text{example-two}, \text{improvement}\}, \{\text{example-one}, \text{bug-fix}\}, \{\text{example-one}, \text{new-feature}\}, \{\text{example-two}, \text{bug-fix}\}\}$, and minimum support of 25%.

In the beginning, FP-growth scans the input data to compute the support count of each item of the itemset, thus generating a descending order of frequent itemsets, so that non-frequent items are dropped. In the example, the frequent itemsets and respective support count are $\{\{\text{example-two}\}, 4\}$, $\{\{\text{example-one}\}, 4\}$, $\{\{\text{improvement}\}, 3\}$, $\{\{\text{bug-fix}\}, 3\}$, $\{\{\text{new-feature}\}, 2\}$, and none disposal, since a minimum support of 25% determines two occurrences to recognize an item as frequent.

Then, FP-growth scans again the input data to compresses it into a FP-tree (which

¹¹Inspired in [122] and <https://t.ly/AbFO>

goal is to prevent repeated data scans), and mines the frequent itemsets directly from the FP-tree [122]. At that point, the FP-tree is formed by scanning one transaction at a time and mapping it onto a branch in the FP-tree. The compression happens when it occurs overlapping of branches, indicating that distinct transactions present items in common. Figure 2.12 illustrates the step-by-step of the FP-tree formation.

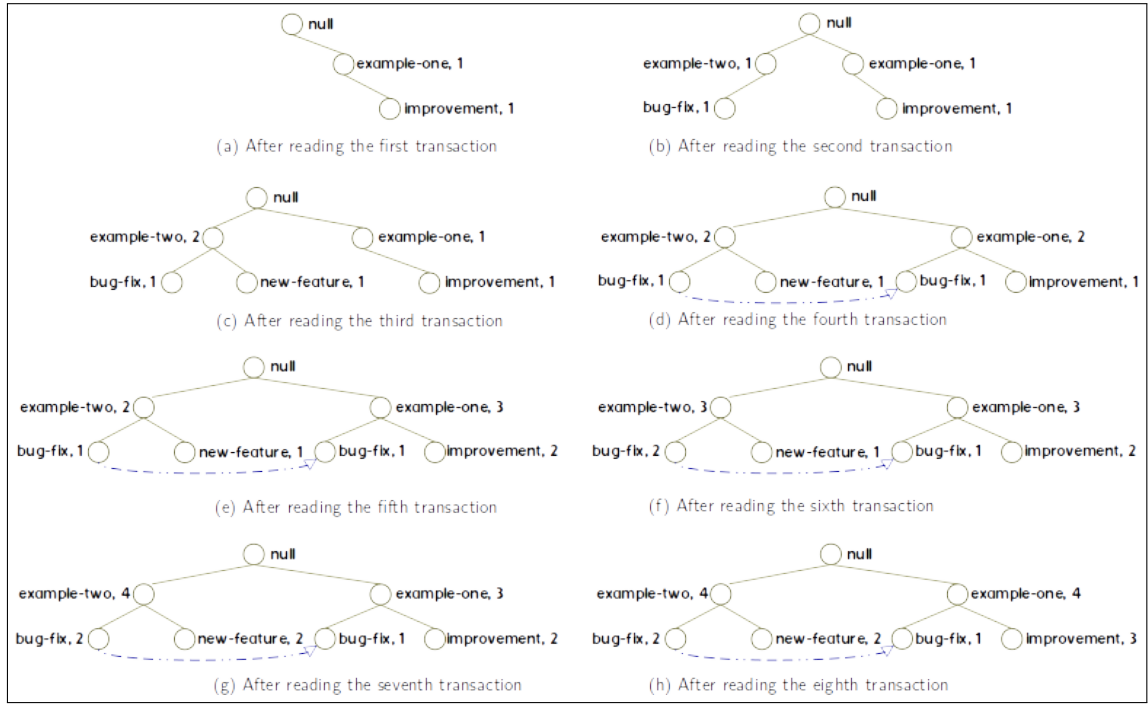


Figure 2.12: Example of the FP-tree formation

Figure 2.12 (a) displays the representation of the first transaction, in which each node has a frequency count of 1. A similar representation of the second transaction is shown in Figure 2.12 (b). The third transaction shares a common item, $\{example-one\}$, with the second one, so the branch for the third transaction overlaps with the branch for the second one, and the count is updated, as presented in Figure 2.12 (c). A similar situation takes place with the fourth transaction that has a common item, $\{example-one\}$, with the first transaction, thus, it occurs overlapping of branches and updating of the count, as registered in Figure 2.12 (d). FP-tree also maintains a list of pointers connecting nodes consisting of the same items, $\{bug-fix\}$, depicted as a blue arrow in Figure 2.12 (d). The process continues until all transactions are scanned, as demonstrated in Figure 2.12 (e) – (h).

Next, the generation of frequent itemsets is performed by examining the FP-tree in a bottom-up manner. The key idea behind this strategy is to derive frequent itemsets ending with a specific item, by exploring only the branches carrying that item, as illustrated in Figure 2.13. The nodes are visited in descending order. Accordingly, after finding the frequent itemsets ending in $\{new-feature\}$ (Figure 2.13 (a)), the algorithm sequentially search for frequent itemsets ending in $\{bug-fix\}$ (Figure 2.13 (b)), $\{improvement\}$ (Figure 2.13 (c)), $\{example-one\}$ (Figure 2.13 (d)), and $\{example-two\}$ (Figure 2.13 (e)).

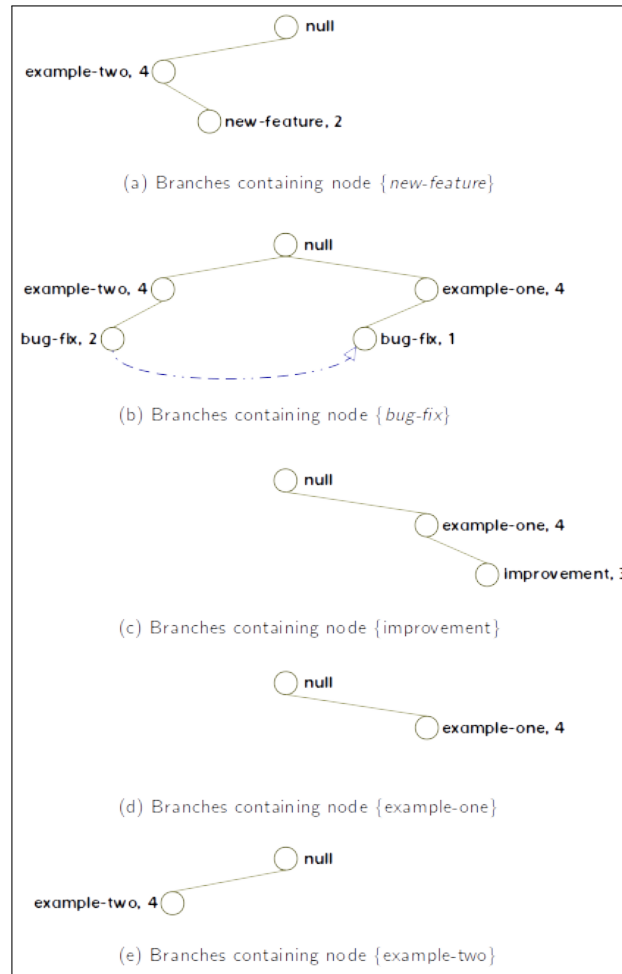


Figure 2.13: Example of the bottom-up approach for generation of frequent itemsets

It follows that a *conditional pattern base* is computed for each item, as registered in Table 2.4 (second column). A conditional pattern base is a set of frequent items co-occurring with a given item [62]. For instance, when analyzing the item $\{new-$

feature} (Figure 2.13 (a)), it is possible noting two co-occurrences with $\{example-two\}$, so $\{\{example-one\}, 2\}$ expresses the corresponding conditional pattern base for $\{new-feature\}$. The same reasoning applies to other items, except for $\{example-one\}$ and $\{example-two\}$ that do not have any common co-occurrence with other items (Figure 2.13 (d), (e)).

After that, a *conditional FP-tree* is built for each item by (i) considering the set of common items in all branches in the conditional pattern base of that item, and (ii) adding the frequency counts of all items of the branches in the conditional pattern base to compute the support count, as outlined in Table 2.4 (third column). Thus, the conditional FP-tree for $\{new-feature\}$ consists of $\{\{example-two\}, 2\}$, since there is only one conditional pattern base. The same occurs for $\{improvement\}$, which conditional FP-tree is composed by $\{\{example-one\}, 3\}$. In case of $\{bug-fix\}$, $\{\{example-two\}, 1\}$ is discarded because its support is lower than the support minimum, so $\{\{example-one\}, 2\}$ is the single node of the FP-tree.

Table 2.4: Example of conditional pattern base, conditional FP-tree, and the respective generated frequent pattern rules

<i>Item</i>	<i>Conditional pattern base</i>	<i>Conditional FP-tree</i>	<i>Frequent pattern rules</i>
$\{new-feature\}$	$\{example-two\}, 2$	$\{example-two\}, 2$	$\{example-two, new-feature\}, 2$
$\{bug-fix\}$	$\{example-two\}, 2$ $\{example-two\}, 1$	$\{example-two\}, 2$	$\{example-two, bug-fix\}, 2$
$\{improvement\}$	$\{example-one\}, 3$	$\{example-one\}, 3$	$\{example-one, improvement\}, 3$
$\{example-one\}$	null	null	null
$\{example-two\}$	null	null	null

Finally, the *frequent pattern rules* are generated by matching the items of the conditional FP-tree to the respective item, as registered in Table 2.4 (fourth column). As a result, three frequent pattern rules meet the support threshold minimum. Note that the association rules can be inferred from frequent pattern rules according to either $\{X\} \rightarrow \{Y\}$ or $\{Y\} \rightarrow \{X\}$. For that, the confidence, lift, and conviction can be regarded to decide on the validity of the generated rules, assisting the **interpreting**

of the results.

In order to clarify, when analysing the frequent pattern rule $\{\{example-one, improvement\}, 3\}$, the confidence of both rules, $\{example-one\} \rightarrow \{improvement\}$ and $\{improvement\} \rightarrow \{example-one\}$, can be computed and checked against a confidence threshold minimum. In particular, $conf(\{example-one\} \rightarrow \{improvement\}) = 0.75$ and $conf(\{improvement\} \rightarrow \{example-one\}) = 1$. Then, by considering the last one association rule, $lift(\{improvement\} \rightarrow \{example-one\}) = 1.25$ and $conv(\{improvement\} \rightarrow \{example-one\}) = \infty$. It can be concluded that this association rule is useful, since the computed lift is greater than 1, and a logical implication because conviction is ∞ .

2.5 Concluding Remarks

Refactoring, MCR, Git pull requests, clustering, and ARL constitute the underlying concepts needed to understand the contextualization of this thesis proposal. In the next chapter, we address related work that make apparent a knowledge gap on an in-depth characterization of refactoring-inducing pull requests.

Chapter 3

Related Work

Since this thesis proposal aims to provide a characterization of refactoring-inducing pull requests, the related research encloses contributions from three fields: mining of code repositories (Section 3.1), characterization of refactorings throughout code evolution (Section 3.2), and characterization of code review (Section 3.3).

3.1 Mining of Code Repositories

Empirical characterization studies of refactorings have explored source code change histories considering different strategies. By mining version histories, Murphy-Hill et al. analyzed commits from Eclipse and JUnit projects in order to characterize refactoring practices [89], and Vassallo et al. studied practices of refactoring, considering 579,671 commits, from 200 projects of Android, Apache, and Eclipse ecosystems [130]. Palomba et al. explored the relationship between three specific types of code changes and 28 refactoring types based on 63 releases of Apache Ant, ArgoUML, and Apache Xerces-J [96]. In this thesis proposal, we intend to characterize 90,583 refactoring edits identified from the history of commits within 2,100 Apache’s pull requests, considering up to 40 distinct types of refactoring detectable by a state-of-the-art tool.

As for mining code reviewing-related data, several studies have used resources from code review tools, such as Gerrit [2]. Rigby and Bird examined data gathered from distinct code reviewing tools when exploring the convergent peer code review practices in proprietary and OSS projects [107]. Considering code review data mined from Gerrit,

Beller et al. characterized the practical benefits of MCR on reviewed code, examining the issues fixed through code reviewing in ConQAT and GROMACS projects [31], Pangsakulyanont et al. assessed the usefulness of MCR discussion in the QT project [97], Pascarella et al. inspected 900 review comments from OpenStack, Android, and QT projects to identify information needs in code reviewing [99], whilst Paixão et al. investigated the impacts of code reviewing on architectural changes from seven OSS projects [95] and examined refactoring practices performed during code reviewing from six systems of Eclipse and Couchbase projects [84].

As the pull-based development model has advanced, several studies have employed the resources offered by code host systems, such as GitHub [6], to mine data concerning technical aspects of development. In this context, Silva et al. extracted version histories from 748 projects from GitHub in order to identify motivations behind the refactored code [116], Brito et al. characterized refactorings over time considering gathered data from ten OSS projects [37], Gousios et al. mined version histories of 292 projects proposing to understand the functioning of the pull-based development of code [59], Li et al. collected 27,339 pull requests and 147,367 review comments from Rails, Elasticsearch, and Angular.js projects in GitHub intending to get a better understanding of code reviews in the pull-based development model [80], Kononenko et al. mined 1,657 pull requests from the Active Merchant project in GitHub to examine factors contributing to the pull request merge time and outcome [73], and Pantiuchina et al. mined change history from 150 repositories in GitHub to investigate the motivations driving developers to refactor code [98].

Given that scenario on code review tools and GitHub pull requests, to fill up the knowledge gap on refactoring-inducing pull requests, this thesis proposal mined code reviewing-related aspects, such as 48,735 review comments, from 4,525 Apache's merged pull requests in GitHub.

3.2 Characterization of Refactorings throughout Code Evolution

Firstly, studies on the refactoring practices revealed evidence on floss refactoring and the shortage of information about the performed refactorings in commit messages at control version repositories, as found by Murphy-Hill et al. [89]. Later, Negara et al. extended the knowledge about manual and tool-supported refactoring, by discovering refactoring patterns across code evolution [90]. Palomba et al. found evidence of refactoring edits in the presence of different types of changes, that is, bug fixes, new features, and general maintenance; the difference is on the number and complexity of refactorings [96]. These works support this thesis proposal because they reinforce the relevance of the analysis of changes applied to the code as a means to get details on the composition and practices of refactoring.

Moreover, motivations for refactoring have been empirically explored. In this perspective, Kim et al. identified readability [71], Silva et al. discovered changes in the requirements (e.g., bug fixes) [116], whereas Vassallo et al. uncovered understandability [130] as main reasons to refactor code. Pantiuchina et al. observed that code readability, change- and fault-proneness, and experience of developers are factors influencing refactorings, when exploring refactoring operations in the pull request level [98]. It is worth clarifying that Pantiuchina et al. analyzed discussion and commits of merged pull requests, containing at least one refactoring in any one of their commits, without pre-processing squashed and merged pull requests. They found that most of the refactorings are triggered from either the original intents of pull requests or discussion with other developers. This thesis proposal differs from those previous works because we specifically provide a characterization of refactoring edits in refactoring-inducing pull requests (Chapter 4). Particularly, we propose an exploration of refactorings performed as part of changes after the first commit on merged pull requests, as a complementary study to the investigation regarding code reviewing-related aspects and refactoring-inducement (Chapter 5). In this context, the findings from Pantiuchina et al. are motivating, since they indicate the influence that code reviewing has on refactoring edits at the pull request level. Nevertheless, we provide findings independent of the

type of merge applied to pull requests because we recover the commits from squashed and merged pull requests before analysis.

Lately, works have extended the exploration on refactoring practices. In this sense, Brito et al. mined refactorings, at the GitHub commit level, and figured that most of the edits are of different types and performed in up to three commits [37]; whereas Paixão et al. investigated intentions behind refactorings performed during code reviewing, by analyzing Gerrit reviews, and found that motivations for refactorings may emerge from code reviews and, in turn, influence the composition of edits and number of reviews [84]. Those findings inspire this thesis proposal in the direction to suggest aspects of refactoring and code review to be explored at the GitHub pull request level. In specif, Paixão et al. answered their research questions based on the evolution of refactorings at code reviewing time in light of distinct types of developers' intents. This thesis proposal, aiming to characterize refactoring-inducing pull requests, provides a study of refactorings-related aspects according to the refactoring-inducement context, independently of developers' intents.

3.3 Characterization of Code Review

The MCR practice has been the focus of characterization studies that explore both OSS and industrial scenarios. As a result, it is perceptible the enhancement of techniques, tools, and recommendations for supporting the MCR over time. Initially, the characterization studies of code reviewing aimed to understand the differences between software inspection as performed in industry and peer code review in OSS development. Rigby et al., in this purpose, provide theory and study methodology in light of metrics similar to those considered in software inspection [109]. They suggest recommendations suitable to the industry as from the OSS code review practices [106], and found empirical evidence on the differences between OSS code review and traditional software inspection [108]. Those works inspired this thesis proposal to expand the knowledge concerning code review practices in pull-based development.

Later, by exploring the motivations and challenges of MCR, Bacchelli and Bird identified code improvements as one of the objectives of code reviewing [26]. A finding

also confirmed by subsequent research on convergent practices of code reviewing by Rigby and Bird [107], Beller et al. [31], and MacLeod et al. [82]. Those findings reinforce the pertinence of investigating refactoring as a relevant contribution from code reviewing.

The analysis of technical aspects of code reviewing has been the focus of several empirical studies. In this perspective, the following measures have been considered: number of reviewers by Jiang et al. [67], review comments by Rigby and Bird [107] and by Beller et al. [31], time to merge by Izquierdo-Cortazar [65], and patch size by Baysal et al. [29]. Those analyses provided the first insights on code reviewing aspects to be investigated in this thesis proposal.

Also, studies explored the factors influencing code review quality. Bosu et al. discovered that changes' properties affect the review comments usefulness [35]. Kononenko et al. carried out analysis concerning how developers perceive code review quality [72], and figured out that the thoroughness of feedback is the main influencing factor to code review quality, besides the size of the change, number of changed files, number of developers in the discussion, number of resubmits, review response time, and length of discussion. Those results corroborate with the findings on the technical aspects empirically studied in [67; 107; 31; 29], thus constituting an enriched set of technical aspects from which we selected the ones under investigation in this thesis proposal.

Recently, code review in pull-based development was examined by Li et al. to comprehend the typical review comment patterns [80]. This thesis proposal goes further in the sense of encompassing code review properties while investigating the refactoring-inducing pull requests.

3.4 Concluding Remarks

Previous research findings on the characterization of refactoring activities and code review promote the establishment of a knowledge gap regarding the characterization of refactoring-inducing pull requests. Next, we characterize refactoring edits in refactoring-inducing pull requests, based on data mined from GitHub repositories.

Chapter 4

Characterizing Refactoring Edits in Refactoring-Inducing Pull Requests

In this chapter, we describe the research design (Section 4.1); then we explain the procedures for data collection (Subsections 4.1.1 and 4.1.3), refactoring detection (Subsection 4.1.2) and data analysis (Subsection 4.1.4), as well as a few countermeasures to attend to validity and reliability issues (Subsection 4.1.5); next, we present the results and discuss them (Section 4.2). Finally, we argue the limitations of the study (Section 4.3).

4.1 Research Design

The goal of this first study is to investigate the refactoring edits in order to characterize the refactoring-inducing pull requests, from the developer’s perspective. In this sense, we formulated the following **research questions**:

- RQ₁ How common are refactoring-inducing pull requests?
- RQ₂ What refactoring types often take place in pull requests?
- RQ₃ How are the refactoring edits characterized?

In particular, these questions aim a better understanding of the refactorings along commits in the context of refactoring-inducing pull requests. Answering those questions drives the proposed research design, in turn, elaborated in accordance with the

guidelines specified by Runeson et al. [112; 113]. Thereby, the **research design** of this study consists of four steps: mining of raw pull requests data, detection of refactoring edits, mining of raw code review data, and data analysis, as illustrated in Figure 4.1 and detailed in the next Subsections (4.1.1 – 4.1.4). We discuss the issues related to the validity and reliability of the study in Subsection 4.1.5.

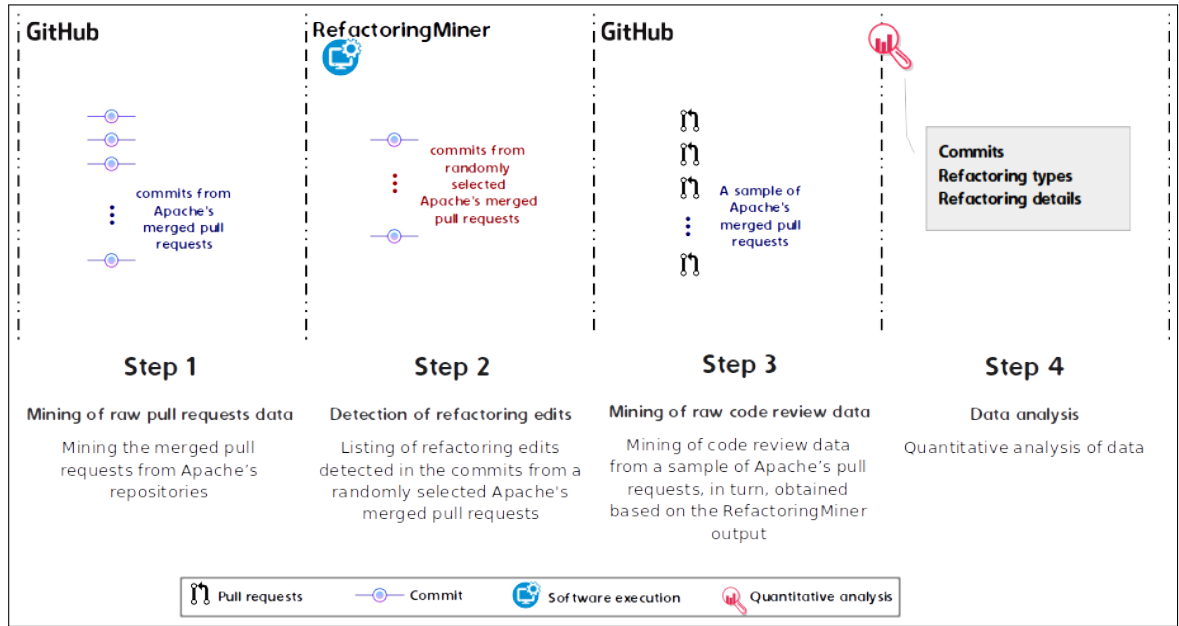


Figure 4.1: Design of the characterization study of refactoring edits

4.1.1 Mining of Merged Pull Requests

The **mining of raw pull requests data** (Step 1) was carried out by mining merged pull requests from Apache's repositories at GitHub. We investigated only merged pull requests because they reveal actions that were in fact finalized, therefore, we can get a more in-depth understanding of refactoring-inducement. GitHub was the chosen platform for gathering pull requests due to its popularity [16] and to the mining resources available through extensive APIs – GitHub REST API v3 [5] and GitHub GraphQL API v4 [4].

The *Apache Software Foundation* (ASF) manages more than 350 open-source projects, completely migrated to GitHub since February 2019, and has more than 7,000 contributors from all over the world [13]. Thus, given the Apache's popularity

in the open-source software development context, we selected it for the mining of pull requests. In special, almost 43% of Apache’s source code is developed in Java language [17], so only Java repositories were taken into consideration in the data collection. It is worth emphasizing that although Apache follows a geographically distributed development as well as other OSS projects [135], the Apache’s development process obey to particular principles, named "the Apache way" [14]: collaborative software development; commercial-friendly standard license; consistently high-quality software; respectful, honest, technical-based interaction; faithful implementation of standards; and security as a mandatory feature.

Before mining the data, in August 26, 2019, we performed a search for Apache’s non-archived Java repositories in GitHub (to take into account only actively maintained repositories) by inputting¹ the search string *user:apache language:java archived:false*, which the results are indicated in Table 4.1. From this, 65,006 merged pull requests were detected in 467 from a total of 956 repositories (48.85%)².

Table 4.1: Search results for merged pull requests from Apache’s non-archived Java repositories in GitHub

<i>Number of repositories</i>	<i>Number of repositories with merged pull requests</i>	<i>Number of merged pull requests</i>
956	467	65,006

Then, we mined those merged pull requests from August 26, 2019 to September 7, 2019. For that, we implemented a Python script³, employing resources available in *PyGithub*⁴ and *Requests*⁵ libraries to use, respectively, GitHub REST API v3 and

¹<https://github.com/search>

²The list of the number of merged pull requests by Apache’s repository is available in <https://git.io/JUWM7>

³Available in <https://git.io/JUWMx>

⁴<https://pygithub.readthedocs.io>

⁵<https://requests.readthedocs.io>

GraphQL API v4. The script generated two datasets⁶, *pull requests dataset* and *commits dataset*. The pull requests dataset consists of 48,338 non-squashed and merged pull requests (74.4% of the total number of Apache’s merged pull requests) from 453 distinct repositories (Table 4.2 displays the top 5 ones). The commits dataset contains 53,915 recovered commits from 16,668 squashed and merged pull requests (25.6% of the total number of Apache’s merged pull requests) from 255 different repositories (Table 4.3 shows the top 5 ones).

Table 4.2: Top 5 repositories containing non-squashed and merged pull requests

<i>Repository</i>	<i>Number of non-squashed and merged pull requests</i>
incubator-druid	3,702
beam	2,838
pulsar	2,805
incubator-pinot	2,673
ambari	2,588

It is worth reminding that when the commits of a pull request are squashed into a single commit (via squash and merge option), in practice, the original commits become not directly accessible when searching the pull request. To deal with this issue, we recovered the commits’ history of each squashed and merged pull request, before any exploration of its original commits. For this reason, there is a specific dataset of recovered commits from squashed and merged pull requests (commits dataset).

The recovery of squashed commits from GitHub, detailed in Appendix B, was implemented and built in the mining script³ (lines 33-45, 126-130, 163-164). Aiming the strategy’s effectiveness, we followed a few open pull requests up to the merge, such as the apache/drill’s pull request 1807⁷, then we (1) ran the RefactoringMiner to detect refactorings in the commits from open pull requests; (2) executed the recovery strategy

⁶Available in <https://git.io/JUWDU>

⁷<https://github.com/apache/drill/pull/1807>

after the pull requests merge; (3) ran the RefactoringMiner in the recovered commits; and (4) compared the outputs from steps (1) and (3).

Table 4.3: Top 5 repositories containing squashed and merged pull requests

<i>Repository</i>	<i>Number of squashed and merged pull requests</i>
beam	2,452
incubator-pinot	1,184
incubator-druid	1,081
cloudstack	894
geode	778

The output datasets are structured in line with the input format required to run the RefactoringMiner, so containing records for the following variables:

- repository’s URL, pull request number, and commit SHA (commits dataset)⁸; and
- repository’s URL, and pull request number (pull requests dataset)⁹.

Therefore, Step 1 effected a pre-processing of Apache’s merged pull requests for detecting refactoring edits (Step 2), in turn, performed in two levels of running: at the *pull request level* for entries from the pull request dataset, and at the *commit level* for entries from the commits dataset. Specifically, the datasets’ entries individually meet the format of input required for the RefactoringMiner running in accordance with its API usage guidelines¹⁰ (by invoking the methods *detectAtCommit()* to commit level and *detectAtPullRequest()* to pull request level).

⁸For example, (<https://www.github.com/apache/dubbo-admin.git>, 481, 77579afb66e1a78eb491f0a783705d40484de5a7).

⁹For example, (<https://www.github.com/apache/dubbo-admin.git>, 479).

¹⁰<https://github.com/tsantalis/RefactoringMiner#api-usage-guidelines>

4.1.2 Refactoring Detection

RefactoringMiner [9] is a state-of-the-art refactoring detection tool (precision of 99.6% and recall of 94%), which has been shown superior to competitive tools [127], to detect refactoring edits applied to Java source codes that follow pull-based development [128]. For this reason, RefactoringMiner is the tool chosen for **refactoring detection** purposes (Step 2). In essence, it identifies the refactoring edits performed in a commit in relation to its parent commit, displaying a description of the applied refactorings, that is, type and associated targets, for instance, the methods and classes involved in an *Extract and Move Method* refactoring¹¹, as exposed in Table 4.4. To clarify, the public method `onAssignment()` is extracted from the protected method `onJoinComplete()` in the class `ConsumerCoordinator` and moved to another class (`PartitionAssignor`) in the same package.

Table 4.4: A RefactoringMiner output example

<i>Refactoring type</i>	<i>Refactoring details</i>
Extract and Move Method	public onAssignment(assignment Assignment, generation int):void extracted from protected onJoinComplete(generation int, memberId String, assignmentStrategy String, assignmentBuffer ByteBuffer):void in class org.apache.kafka.clients.consumer.internals.ConsumerCoordinator & moved to class org.apache.kafka.clients.consumer.internals.PartitionAssignor

In this step, only merged pull requests containing two or more commits were taken into account for refactoring detection, to conform with the definition of refactoring-inducing pull requests (Definition 1). Given the input datasets (pull requests dataset and commits dataset), the RefactoringMiner execution revealed to be a time-consuming task, demanding on average 2.3 minutes by pull request. For that reason, we consider

¹¹From the refactoring detection in Apache’s kafka pull request #6763; a commit available in <https://git.io/JU1Dq>.

a sample, summarized in Table 4.5, randomly obtained from the RefactoringMiner execution from September 18, 2019 to October 2, 2019. Our sample consists of 225,127 detected refactorings in 8,761 merged pull requests (13.5% of the total number of Apache’s merged pull requests) from 209 different repositories (44.7% of the total number of Apache’s repositories with merged pull requests). In particular, those 8,761 pull requests comprise a total of 68,209 commits.

Table 4.5: Output of RefactoringMiner execution in Apache’s repositories

<i>Number of repositories</i>	<i>Number of pull requests</i>	<i>Number of commits</i>	<i>Number of detected refactorings</i>
209	8,761	68,209	225,127

We used the RefactoringMiner, a version available in September 2019 that, in turn, has provided the detection of up to 40 different types of refactoring (Chapter 2, Subsection 2.1.3). We performed RefactoringMiner at the commit level for the commits dataset, and at the pull request level for the pull requests dataset, which results are summarized in Table 4.6. Accordingly, the sample is composed of 8,761 pull requests, from which 38.5% are squashed and merged pull requests containing 45.4% of the total number of commits, and 38.4% of the detected refactorings. In specific, 3,370 squashed and merged pull requests belong to 146 repositories (top 5 repositories are listed in Table 4.7), while 5,391 non-squashed and merged pull requests are from 116 repositories (top 5 repositories are listed in Table 4.8).

The output dataset¹² after running Step 2 comprises the following variables: repository’s name, pull request number, commit(s) SHA, refactoring type(s), refactoring detail(s), and level¹³. Level is a flag indicating the RefactoringMiner running level, either commit level for squashed and merged pull requests or pull request level for non-squashed and merged pull requests.

¹²Available in <https://git.io/JU1Dk>

¹³For example, (*apache/flink*, *9595*, *886419f12f60df803c9d757e381f201920a8061a*, *Rename Variable, table:Table to src:Table in method public testPartitionPrunning():void in class org.apache.flink.connectors.hive.HiveTableSourceTest*, *commit*).

Table 4.6: RefactoringMiner output, at commit and pull request levels, for Apache’s repositories

<i>Level</i>	<i>Number of repositories</i>	<i>Number of pull requests</i>	<i>Number of commits</i>	<i>Number of detected refactorings</i>
commit	146	3,370	30,955	86,414
pull request	116	5,391	37,254	138,713

Table 4.7: Top 5 repositories containing squashed and merged pull requests in the sample

<i>Repository</i>	<i>Number of squashed and merged pull requests</i>
flink	380
beam	263
cloudstack	242
geode	237
incubator-pinot	226

4.1.3 Mining of Code Review Data

The **mining of raw code review data** (Step 3) consists of collecting code reviewing-related attributes, listed in Table 4.9, to the pull requests for which the refactoring detection was executed (Step 2). Those attributes were selected from empirical studies of related work (Chapter 3), and mined by using the GitHub REST API v3 [5], concerning changes (number of commits [87; 108; 132], code churn (the number of changed lines, i.e., number of added lines plus number of deleted lines) [31; 108; 126], and number of changed files [31; 35; 72]), and code reviewing (number of reviewers [67; 106; 107; 108; 115], number of review comments [31; 59; 87; 95; 107], length of discussion [59; 72; 73; 87; 107; 126], review comments [80; 84; 98], and time to merge [59; 65; 73]). Moreover, the attributes number, title, labels, and repository’s name are use-

Table 4.8: Top 5 repositories containing non-squashed and merged pull requests in the sample

<i>Repository</i>	<i>Number of non-squashed and merged pull requests</i>
kafka	647
dubbo	611
beam	604
tinkerpops	304
cloudstack	280

ful to uniquely identify a pull request. It is worth clarifying that length of discussion and time to merge of a pull request are respectively derived as follows:

- length of discussion = review comments + non-review comments, and
- time to merge (in number of days) = merge date – creation date.

For mining the code review related-attributes from GitHub, we developed a Python script¹⁴, employing resources available in the *PyGithub*¹⁵ library to use the GitHub REST API v3. Specifically, a *precondition* was imposed on mining: only merged pull requests, comprising at least one review comment, should be mined in conforming with the justification for the investigation of review comments established in the comparative study design (Chapter 5, Section 5.1).

The mining occurred from February 20 to 26, 2020 and it generated two datasets¹⁶, *code review dataset* and *review comments dataset*. After that, we refined them according to following procedures:

- dropping of merged pull requests containing inconsistencies in terms of zero

¹⁴Available in <https://git.io/JU813>

¹⁵<https://pygithub.readthedocs.io>

¹⁶Available in <https://git.io/JU81G>

Table 4.9: Pull requests' attributes selected for mining

<i>Attribute</i>	<i>Type</i>	<i>Description</i>
number	continuous	Numerical identifier of a pull request
title	categorical	Title of a pull request
repository	categorical	Repository's name of a pull request
labels	categorical	Labels associated with a pull request
commits	continuous	Number of commits in a pull request
additions	continuous	Number of added lines in a pull request
deletions	continuous	Number of deleted lines in a pull request
changed files	continuous	Number of changed files in a pull request
creation date	continuous	Date and time of a pull request creation
merge date	continuous	Date and time of a pull request merge
review comments	continuous	Number of review comments in a pull request
non-review comments	continuous	Number of non-review comments in a pull request
review comments text	text	The review comments of a pull request

changed files¹⁷, and zero reviewers due to problems with reviewers' username¹⁸;

- checking for duplicates; and
- mining from non-mirrored repositories¹⁹.

As a result, 4,236 pull requests were discarded. Therefore, the final sample, summarized in Table 4.10, consists of code review data from 4,525 merged pull requests (6.9% of the total number of Apache's merged pull requests and 51.6% of the number of sample's pull requests obtained from Step 2), encompassing 31,509 commits, 90,583 detected refactorings, and 48,735 review comments, mined from 134 different Apache's

¹⁷In these repositories/pull requests: guacamole-client/#328, and maven-plugins/#18.

¹⁸In these repositories/pull requests: brooklyn-server/#570, cloudstack/#2346, dubbo/#1607/#2662/#3326, flink/#4551/#5561, geode/#1017/#1776, hadoop/#663, incubator-iceberg/#21, incubator-iotdb/#203, maven-jxr/#6, and netbeans/#594.

¹⁹It is an additional step of checking for replicas to drop duplicates. Anyway, no mirrored repository was found in the process.

repositories. Table 4.11 lists the top 5 repositories and, particularly, the first three repositories are expressive in number of commits concerning the other Apache’s repositories, because they together comprise 8,97% of the Apache’s commits, in 2019 [17]. The final sample contains 2,278 squashed and merged pull requests (50.3%), and 2,247 non-squashed and merged pull requests (49.7%).

Table 4.10: A summary of the final sample from mining of Apache’s merged pull requests in GitHub

<i>Number of repositories</i>	<i>Number of pull requests</i>	<i>Number of commits</i>	<i>Number of detected refactorings</i>	<i>Number of review comments</i>
134	4,525	31,509	90,583	48,735

Table 4.11: Top 5 Apache’s repositories in the final sample

<i>Repository</i>	<i>Number of merged pull requests</i>
kafka	517
beam	500
flink	360
cloudstack	274
servicecomb-java-chassis	233

The final sample includes pull requests created from March 18, 2014 to September 09, 2019, and merged from July 18, 2014 to September 05, 2019. As presented in Figure 4.2, most of these pull requests were created and merged in 2018 and 2019, what corroborates with the progressive migration of Apache’s projects to GitHub [13].

The code review dataset consists of the following variables by pull request: repository’s name, number, title, a list of labels, date and time of creation, date and time of merge, number of commits, number of changed files, number of added lines, number of deleted lines, number of reviewers, number of review comments, number of non-review

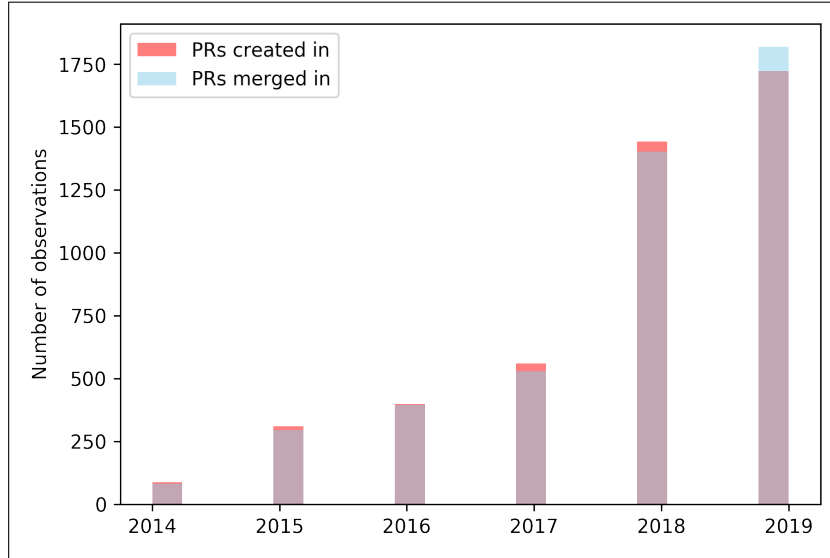


Figure 4.2: Creation and merge dates of the final sample’s pull requests

comments, length of discussion, a flag indicating the presence of refactoring edits, and number of detected refactorings.

The code review comments dataset encompasses records of reviewers’ comments related-attributes of all pull requests registered in the code review dataset: the name of the repository that comment comes from, the pull request number from which the review comment comes from, the original commit SHA from which the review comment comes from, the review comment identifier, the review comment identifier to which the review comment is a reply, the identifier of the hunk²⁰ from which the review comment comes from, the review comment body, the date and time of the review comment creation, and the date and time of the pull request creation.

4.1.4 Data Analysis

Based on the dataset obtained from Step 3, we run the **data analysis** (Step 4) which includes the exploration of detected refactorings (number of refactoring edits, refac-

²⁰A hunk contains the differing lines between two files, as displayed by the diff tool in GitHub. For example, @@ -27,7 +27,7 @@ class HelloWorld.java indicates that the hunk starts at line 27 and has a total of 7 lines in the previous HelloWorld.java file, while the hunk starts at line 27 and has a total of 7 lines in the new HelloWorld.java file.

toring types, refactorings along the commits, and levels of refactoring edits) by pull request in order to answer the research questions.

Specifically, we computed the number of refactorings by considering the edits detected after the initial commit in each pull request. Also, we computed a 95% confidence interval for the percentual of refactoring-inducing pull requests in Apache’s merged pull requests, by performing *bootstrap resampling* in order to complement the answer to the first research question. Bootstrap is a method of statistical inference to estimate a population parameter, such as proportion, as from a sample. Resampling provides estimates on how the point estimate may vary, given that each resample presents own properties [48]. In this study, the computing of the confidence intervals was done on 5,000 resamples from the sample, by considering the proportion of refactoring-inducing pull requests in each resample.

The examined refactoring types consist of the catalog of detectable refactoring types by RefactoringMiner in September 2019 that detects up to 40 different types of refactoring (Chapter 2, Subsection 2.1.3). Moreover, our classification of refactoring levels, presented in Table 4.12, is based on the three-level classification proposed by Murphy-Hill et al. [89]. In brief, a *low-level* refactoring restructures only code blocks, as in *Rename Variable*. A *medium-level* refactoring alters code blocks and the signature of classes, methods, or attributes, such as in *Rename Method*. A *high-level* refactoring modifies the signature of classes, methods, or attributes, as in *Rename Class*.

Table 4.12: Proposed mapping, adapted from [89], between levels of operation and refactoring types

<i>Level</i>	<i>Refactoring types</i>
High-level	Change Attribute Type
	Change Package
	Extract Attribute
	Extract Class
	Extract Interface
	Extract Subclass
	Extract Superclass
Continued on next page	

Table 4.12 – continued from previous page

<i>Level</i>	<i>Refactoring types</i>
	Merge Attribute Move and Rename Attribute Move and Rename Class Move Attribute Move Class Move Source Folder Pull Up Attribute Pull Up Method Push Down Attribute Push Down Method Rename Attribute Rename Class Replace Attribute Split Attribute
Medium-level	Change Parameter Type Change Return Type Extract Method Extract and Move Method Inline Method Merge Parameter Move Method Rename Method Split Parameter
Low-level	Change Variable Type Extract Variable Inline Variable Merge Variable Parameterize Variable Rename Parameter Rename Variable Replace Variable with Attribute Split Variable

4.1.5 Validity and Reliability

We propose the countermeasures listed in Table 4.13 to deal with the issues of **validity and reliability**. To clarify, the elaboration of our research design follows well-defined guidelines [112; 113], and was regularly reviewed by the supervisors of this thesis proposal. A chain of evidence is established from a clear description of the sequence of results and conclusions [113]; in this sense, this thesis proposal makes an effort to systematically explain the performed procedures, taken decisions, and obtained results for each study’s step.

Table 4.13: Validity and reliability countermeasures for the characterization study of refactorings

<i>Type</i>	<i>Description</i>
Internal validity	Proposal for a research design to guide the data collection and data analysis procedures
Construct validity	Establishment of a chain of evidence
	Regular revision of the research design by supervisors
	Selection of a state-of-the-art tool for refactoring detection purposes
Reliability	Effort towards clarifying the data collection and data analysis procedures in order to enable replications

4.2 Results and Discussion

Following, we describe the results and discuss them as per the research questions.

4.2.1 How Common are Refactoring-Inducing Pull Requests?

We found 2,100 refactoring-inducing pull requests (46.4% of our sample’s pull requests). From these, 1,137 are squashed and merged pull requests (54.1%). Specifically, RefactoringMiner detected 106,704 refactoring edits, however, 90,583 as from the second commit (that is, 84.9% of the total number of detected refactorings). As shown in

Figure 4.3 (a), the histogram of refactoring edits is positively skewed, presenting outliers, so that a low number of refactorings is quite frequent. The number of refactoring edits by pull request is 43.1 (SD = 122.7) on average and 6 on median (IQR = 25), as displayed in Figure 4.3 (b). The presence of outliers can indicate scenarios scientifically relevant for exploration.

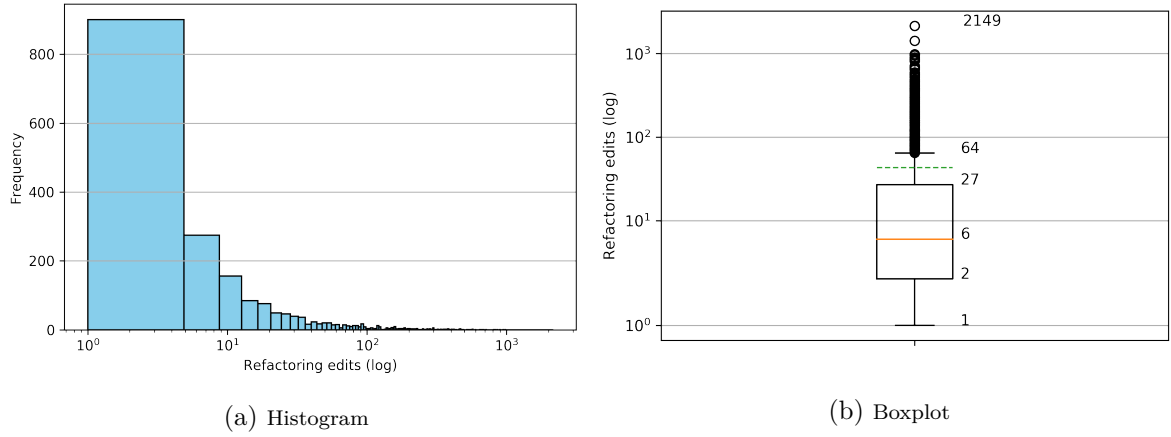


Figure 4.3: Refactoring edits in the refactoring-inducing pull requests

In addition, by using bootstrapping resampling and a 95% confidence level, we obtained a confidence interval ranging from 44.9% to 47.8% for the percentual of refactoring-inducing pull requests in the Apache's merged pull requests.

Those results reveals that there are concerns with refactoring in the pull requests context, independently of the type of merge, since 54.1% of the refactoring-inducing pull requests were squashed and merged. This is a motivating result for further investigation regarding how code review influences refactoring edits.

Finding 1. We found 46.4% of refactoring-inducing pull requests. The percentual of refactoring-inducing pull requests in the Apache's merged pull requests is in [44.9%, 47.8%], for a 95% confidence level.

4.2.2 What Refactoring Types often Take Place in Pull Requests?

Table 4.14 lists 39 distinct types of detected refactorings in sample's pull requests. In 345 refactoring-inducing pull requests, RefactoringMiner detected only one type of

refactoring. Thus, 83.6% of the refactoring-inducing pull requests have refactoring edits of different types (6.9 on average, and 5 on median, as shown in Figure 4.4). This finding is aligned with Brito and colleagues' findings, who identified that most refactorings are composed of more than one refactoring type over commits [37]. Accordingly, we understand that distinct subjects are addressed in the context of refactorings at pull request level.

Table 4.14: Refactoring types detected in the pull requests

<i>Refactoring type</i>	<i>Number of detections</i>	<i>Percentual (%)</i>
Change Variable Type	11,507	12.703
Change Parameter Type	8,298	9.161
Rename Attribute	8,116	8.960
Rename Method	7,145	7.888
Change Return Type	6,719	7.417
Extract Method	6,682	7.377
Rename Variable	6,498	7.173
Change Attribute Type	6,395	7.059
Rename Parameter	5,297	5.848
Move Class	3,772	4.164
Move Attribute	3,156	3.484
Move Method	2,548	2.813
Extract Variable	2,218	2.448
Extract and Move Method	1,988	2.195
Pull Up Method	1,632	1.801
Rename Class	1,538	1.698
Pull Up Attribute	1,018	1.124
Inline Method	723	0.798
Move and Rename Class	649	0.716
Extract Class	643	0.709
Replace Variable with Attribute	518	0.572
Push Down Method	511	0.564
Parameterize Variable	487	0.537

Continued on next page

Table 4.14 – continued from previous page

<i>Refactoring type</i>	<i>Number of detections</i>	<i>Percentual (%)</i>
Extract Attribute	406	0.448
Inline Variable	405	0.447
Move Source Folder	367	0.405
Extract Superclass	321	0.354
Push Down Attribute	248	0.274
Change Package	206	0.227
Extract Interface	137	0.151
Merge Attribute	87	0.096
Extract Subclass	86	0.095
Merge Parameter	60	0.066
Move and Rename Attribute	56	0.062
Split Attribute	49	0.054
Merge Variable	45	0.049
Replace Attribute	23	0.025
Split Parameter	19	0.021
Split Variable	10	0.011
<i>Total</i>	<i>90,583</i>	<i>100.0</i>

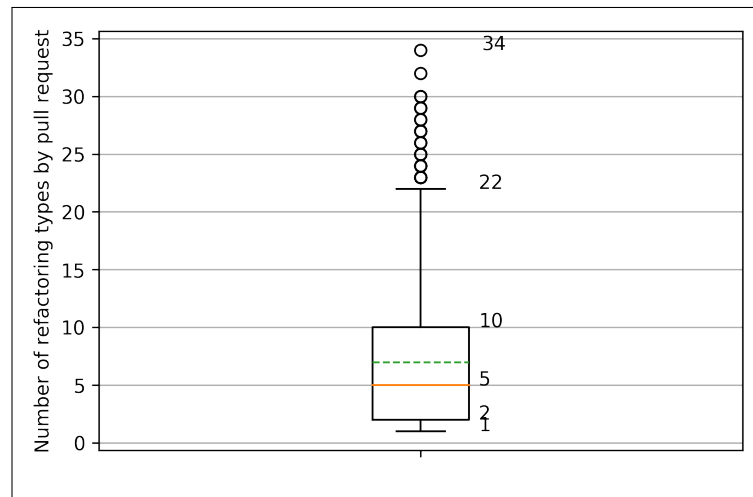


Figure 4.4: Number of detected refactoring types by refactoring-inducing pull request

The most and least refactorings are respectively of the *Change Variable Type* and *Split Variable* instances, a pattern that persists from second until sixth commit (which edits are of 39 distinct types), as illustrated in Figure 4.5. Over those commits, the refactorings present a gradual decrease in number of edits, except in the case of *Inline Method* and *Inline Variable* in the sixth and fifth commits, respectively.

Given that changes are driven by review comments [31] and refactorings are also triggered by discussion [98], we conjecture that code reviewing might influence the performing of refactoring edits, so producing that pattern, because commits in a pull request can express the implemented changes in response to reviewers' comments.

In terms of refactoring kinds, the most and least frequent edits are respectively of *Change Type* (36.34%) and *Split Variable* (0.08%) kinds, as shown in Table 4.15. In particular, *Change Type*, *Rename*, *Extract* and *Move* together encompass more than 90% of all detected refactorings, what suggests development efforts concerning code maintainability [125], readability [96], understandability [130], and cohesion and coupling [94].

To clarify, *Change Type* refactorings consist of changing a class's member type. For instance, suppose that a developer implemented the Java source code 4.1 and wants to change the type of the variable *age* to better accommodate input values. Then, he/she can refactor the code applying a *Change Variable Type* ($\text{int} \rightarrow \text{byte}$), as demonstrated in source code 4.2. *Change Type* refactorings are essential for class/library migration because it contributes to code maintainability [125]. This refactoring kind is not in the Fowler's catalog [53], but it is detectable by RefactoringMiner 2.0 [127].

```
1 public void simpleMethod() {  
2     int age;  
3     //do something  
4 }
```

Source Code 4.1: A simple Java method

```
1 public void simpleMethod() {  
2     byte age;  
3     //do something  
4 }
```

Source Code 4.2: A simple Java refactored method

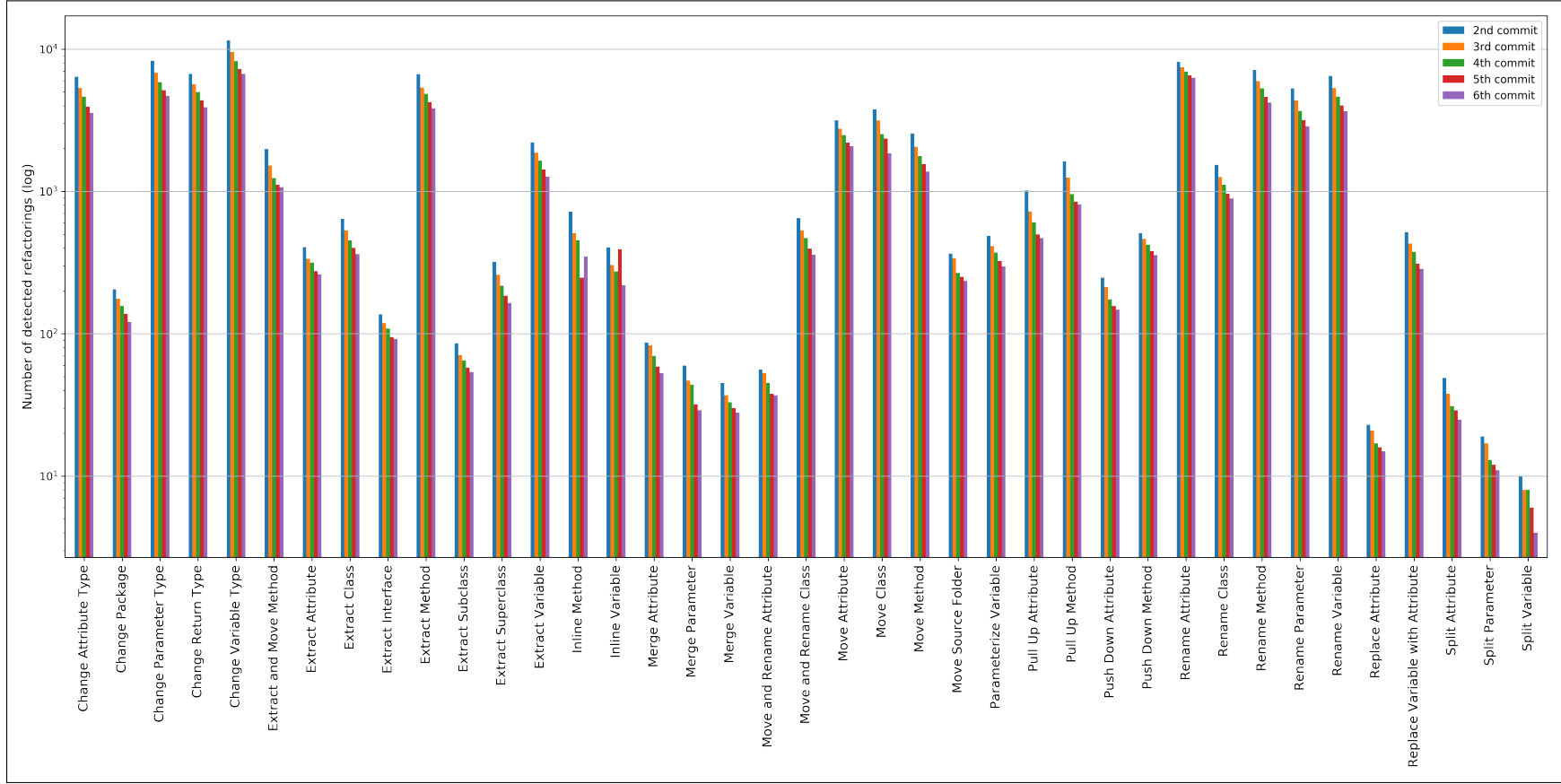


Figure 4.5: Distribution of refactoring types in the refactoring-inducing pull requests from the second until the sixth commit

Rename refactoring instances support the effort towards code readability [96] and understandability [130], whereas *Extract* and *Move* refactorings tackle concerns about cohesion and coupling [94].

Table 4.15: Number of detected refactoring kinds in the refactoring-inducing pull requests

<i>Refactoring kind</i>	<i>Number of detected refactorings</i>	<i>Percentual (%)</i>
Change Type	32,919	36.34
Rename	28,594	31.57
Extract	10,493	11.58
Move	9,843	10.86
Pull Up	2,650	2.92
Extract and Move	1,988	2.21
Inline	1,128	1.24
Push Down	759	0.84
Move and Rename	705	0.78
Replace	541	0.60
Parameterize	487	0.54
Change Package	206	0.23
Merge	192	0.21
Split	78	0.08
<i>Total</i>	<i>90,583</i>	<i>100.0</i>

This result differs from findings of Murphy-Hill and colleagues who found *Rename* refactorings as the most frequent when studying the refactoring practices in two other OSS projects [89], of Pantiuchina and colleagues who observed *Extract* refactorings as the most common when exploring motivations behind refactoring edits at pull request level in 150 projects in GitHub [98], and of Paixão and colleagues who identified *Extract Method* and *Move Method* as the most typical types when investigating refactorings over reviews on Gerrit [84]. It is worth mentioning that the catalog of refactorings, considered by those studies, differs from that detectable by RefactoringMiner 2.0 [127]; mainly, concerning *Change Type* refactorings.

Finding 2. We found 39 distinct refactoring types in the refactoring-inducing pull requests. The most common refactorings are *Change Type*, *Rename*, *Extract* and *Move* edits, so addressing code improvements concerning maintainability, readability, cohesion and coupling.

4.2.3 How are the Refactoring Edits Characterized?

We explored the number of detected refactorings through the commits and the refactoring levels in order to answer this question. As depicted in Figure 4.6 (a), the histogram of number of commits is positively skewed, thus a low number of commits is quite frequent. Refactoring-inducing pull requests have 11.0 commits on average ($SD = 25.9$) and 5 on median ($IQR = 6$), as displayed in Figure 4.6 (b).

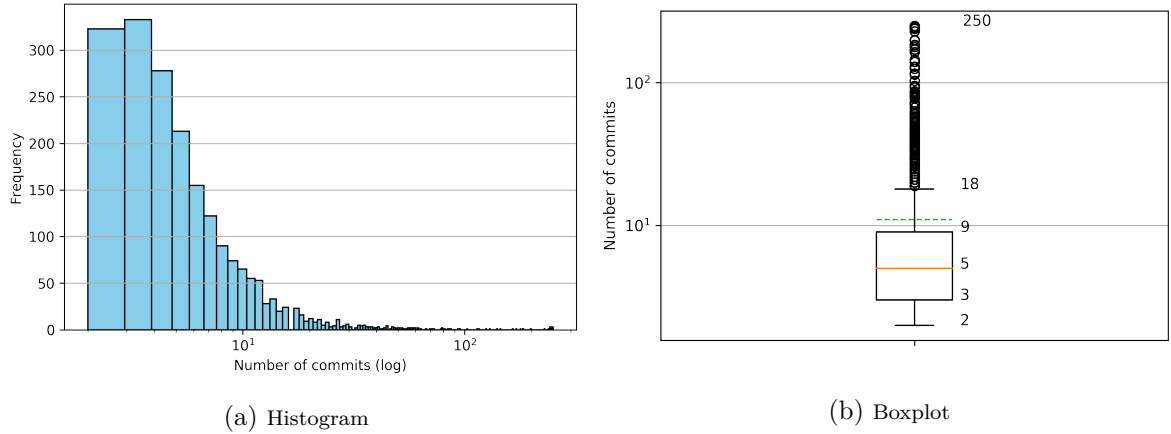


Figure 4.6: Commits in refactoring-inducing pull requests

We visually explored the relationship between number of commits and number of refactorings (Figure 4.7) and got no clear general trend. Then, we run Spearman's correlation test, considering 5-quantiles binning of data, and found a moderate and positive monotonic correlation between the two variables ($r_s = 0.502$, $n = 2,100$, $p < .05$). Further, by examining the number of refactorings detected from the second until the sixth commit (Table 4.16), we realized a decreasing number of edits over commits.

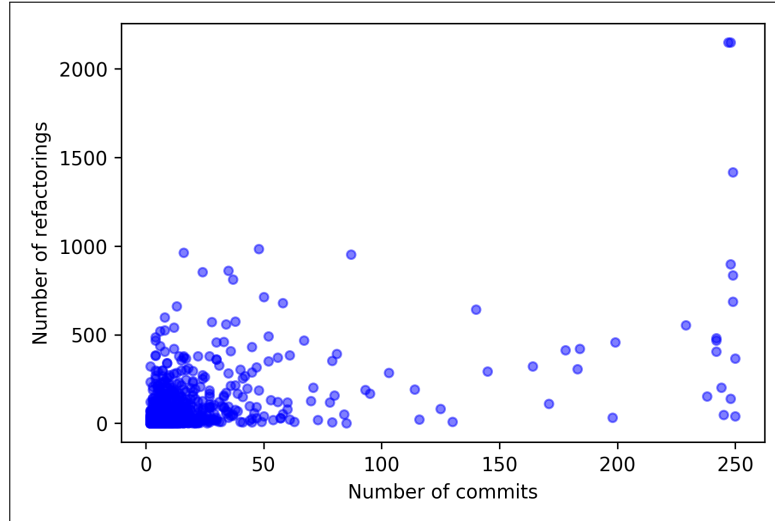


Figure 4.7: Scatterplot of number of refactorings in relation to number of commits

Table 4.16: Number of detected refactorings from the second until the sixth commit

<i>Commit</i>	<i>Number of detected refactorings</i>	<i>Percentual (%)</i>
2nd	15,094	16.6
3rd	9,529	10.5
4th	7,810	8.6
5th	5,029	5.5
6th	3,729	4.1

Finding 3. We found a moderate and positive monotonic correlation between the number of commits and number of refactorings in refactoring-inducing pull requests ($r_s = 0.502$, $n = 2,100$, $p < .05$). Also, the number of refactoring edits gradually decrease over commits of refactoring-inducing pull requests.

We register the results obtained from the classification of the operation levels of the detected refactorings in Table 4.17. Accordingly, there is no prevalence of a level concerning the others. In particular, almost 70% of refactorings are restructurings implemented at *medium* and *low* levels. This result corroborates with the conclusions of

Murphy-Hill and colleagues concerning the significant number of refactorings performed in *medium* and *low* levels [89], since those edits do not alter the interface of programs, constituting reduced impacts of change in comparison to high-level refactorings, which modify packages, classes, and member signatures.

Table 4.17: Levels of detected refactorings in the refactoring-inducing pull requests

<i>Level</i>	<i>Number of detected refactorings</i>	<i>Percentual (%)</i>
High-level	29,416	32.5
Medium-level	34,182	37.7
Low-level	26,985	29.8
<i>Total</i>	<i>90,583</i>	<i>100.0</i>

Finding 4. The refactorings detected in refactoring-inducing pull requests consist of high, medium and low impact restructurings.

4.3 Limitations

This characterization study presents a few limitations, despite the efforts in the purpose of countermeasures to deal with validity and reliability issues (Subsection 4.1.5).

By considering concerns on internal validity, it is worth pointing out that the research design was elaborated after conducting two brief case histories in order to get a better understanding of GitHub’s pull requests and the procedures of data collection and refactoring detection, as reported in Appendix A. Even so, there are risks of threats to internal validity due to any non-previously identified deficiencies in the research questions and procedures of the research design.

Despite the efforts towards the establishment of a chain of evidence for the data interpretation and description of taken decisions in the research design, the detected refactorings were not validated before data analysis (since a manual validation of edits means a time-consuming task [128]), so expressing a potential threat to construct validity. In order to overcome this issue, we selected the RefactoringMiner, a state-

of-the-art tool (precision of 99.6% and recall of 94%) [127], for refactoring detection purposes.

Although case studies enable analytical generalization, which refers to the generalization from empirical observation to theory rather than a population [113], it is worth indicating that Apache’s projects follow particular principles of software development, as argued in Subsection 4.1.1. Therefore, this is a threat to external validity, since the findings of the characterization study are exclusively extended to cases which have common characteristics with Apache’s projects.

4.4 Concluding Remarks

In this chapter, in addition to presenting the research design, we discuss the results and limitations of the characterization study of refactoring edits by considering 2,100 Apache’s refactoring-inducing pull requests. Accordingly, the refactorings comprise 39 distinct types, which gradually decrease from the second to sixth commits, and consist of a high, medium, and low impact restructurings.

Chapter 5

Comparing Refactoring-Inducing and non-Refactoring-Inducing Pull Requests

This chapter presents a comparative investigation between refactoring-inducing and non-refactoring-inducing pull requests based on the Apache’s sample obtained from the mining of raw code review data (Chapter 4, Subsection 4.1.3). It is worth mentioning that the research design proposed for this comparative study was also defined after preliminary motivating results from two ‘case histories’ considering Apache’s pull requests, as reported in Appendix A. In the following sections, we detail the research design (Section 5.1), the preliminary results and discuss them (Section 5.2), and the limitations of the study (Section 5.3).

5.1 Research Design

The goal of this study is to investigate code reviewing-related data in order to characterize refactoring-inducing pull requests in Apache’s repositories hosted in GitHub, from the reviewers’ perspective. In this regard, our main **research question** is:

- RQ₄ How do refactoring-inducing pull requests compare to non-refactoring-inducing ones?

In particular, we designed this question to better understand similarities and differences on pull-requests based on the refactoring edits detected. In this sense, we composed an empirical study, supported by [112] guidelines, that comprises three steps: clustering, ARL, and data analysis, as depicted in Figure 5.1. We describe the research steps in Subsections 5.1.1, 5.1.2, and 5.1.3, and argue a few issues concerning validity and reliability of the study in Subsection 5.1.4.

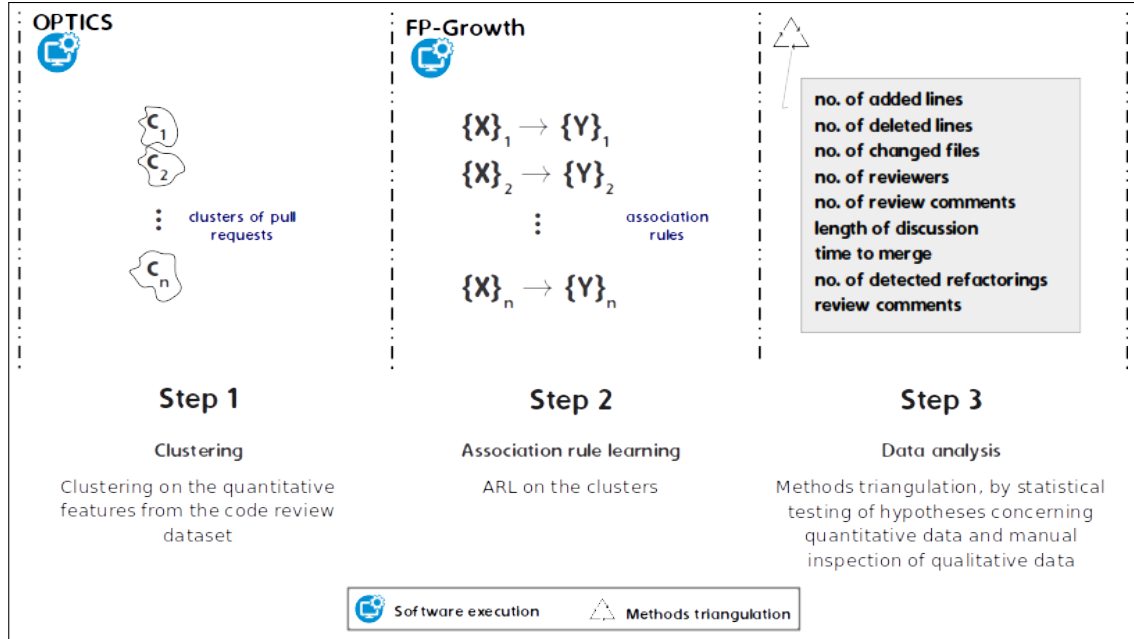


Figure 5.1: Design of the comparative study

It is worth clarifying that in order to explore the differences on refactoring-inducing pull requests and non-refactoring-inducing ones, we first applied unsupervised learning through clustering (Step 1) and ARL (Step 2) processes. This decision was taken because such strategies assist exploratory analysis by automatically identifying the data inherent structure derived from the relationships between distinct characteristics [33; 39]. Note that we sequentially applied clustering and ARL. In that setting, clustering worked as a preprocessing of data, while looking for similarities/dissimilarities between pull requests, intending to get meaningful associations, and hence, to support the formulation of more accurate hypotheses. We expect that clusters exclusively formed by either refactoring-inducing pull requests or non-refactoring-inducing pull requests might present interesting properties for exploration by the ARL technique instead of

considering non-clustered data.

The description of both processes is driven by this unsupervised learning workflow: selecting features (Phase 1), feature engineering (Phase 2), choosing and running an appropriate algorithm (Phase 3), and interpreting results (Phase 4).

5.1.1 Clustering

Phase 1. Selection of Features

For clustering, we selected a set of quantitative features concerning changes, code review, and refactorings, from the code review dataset. We considered a three-context perspective (changes, code review, and refactorings) because these contexts together may potentially support the identification of differences between refactoring-inducing and non-refactoring-inducing pull requests in the clustering process. These are the selected features for clustering:

- number of commits,
- number of changed files,
- number of added lines,
- number of deleted lines,
- number of reviewers,
- number of review comments,
- length of discussion,
- time to merge, and
- number of detected refactorings.

One may argue that other features (e.g., pull request number, title, label, date, non-review comments) could also be considered. However, (i) a pull request's number is only an identifier; (ii) a pull request's title is built on natural language, so it is subject to ambiguities; (iii) pull requests' labels are not mandatory, only 1,044 pull requests from our dataset have labels (23.1%); (iv) date and time of creation/merge are specific values, so we used the difference between them (time to merge) for exploration purposes;

and (v) the number of non-review comments of a pull request is part of its length of discussion.

Phase 2. Feature Engineering

This procedure encompasses the transformation of raw data into proper data for a machine learning algorithm execution [134]. In this sense, we previously checked the selected features' distribution (Appendix C¹) in order to identify specific requirements in direction to feature engineering. As a result, we applied quantile binning to all selected features since the data is non-normally distributed, and due to the presence of outliers, which might outweigh the similarity between data points. Quantile binning is a discretization technique that groups values into quantiles and discards the actual values, so mapping a continuous number to a discrete one [134]. Specifically, we scaled the raw data into quarters (four equal portions of data), so that the outliers were also taken into account, since, in the context of this thesis proposal, outliers do not represent experimental errors; consequently, they can indicate circumstances for further examination.

Phase 3. Selection and Execution of a Clustering Algorithm

We selected the OPTICS algorithm for clustering. This decision was taken by considering the comparative analysis of 46 clustering algorithms in light of the technical aspects investigated in a literature survey, conducted by Xu and Tian [131]: time complexity, adequacy to arbitrary shapes of data, sensibility to the sequence of inputting data, low sensibility to noise and outliers, and no requirement for presetting the number of clusters.

The OPTICS algorithm presents a time complexity $O(n \log n)$; adequacy to arbitrary shapes of data; ease discovering clusters of complex shapes; the ability to find tight and wide clusters; effectiveness for large-scale data; low sensibility to the sequence of input data, noise (which aid to restrict data overfitting [105]), and outliers; and mainly no requirement to input the number of clusters as a parameter. Therefore, for the

¹Available in <https://git.io/JtsRX>

context of this thesis proposal, those properties together express a positive differential of OPTICS.

We developed a Python script² for clustering our sample of pull requests, which reuses the OPTICS implementation available in the module `sklearn.cluster`³ [100]. The input parameters were determined in line with the following procedure: (1) inputting a range of values for three specific parameters, and (2) computing a DBCV index to each output obtained from (1) in order to select the best input values. In order to clarify, we present a fragment of the clustering script, which implements this procedure, in Listing 5.1.

The range of input values comprises, specifically:

- a range of integer values between 10 and 20 (line 2), as recommended by the OPTICS’s authors [24], for *min_samples* (minimum number of points in a neighborhood required to a point be considered a core-point),
- the values 0.01, 0.05, 0.10, 0.15 and 0.20 (line 3) for *min_cluster_size* (minimum number of points to form a cluster), expressed as a fraction of the number of points, and experimentally submitted, and
- the valid values available in the `sklearn.cluster` OPTICS implementation, ‘city-block’⁴, ‘cosine’⁵, and ‘euclidean’ (line 4), for *metric* (distance metric for measuring similarity).

The variation of input parameters occurs in a triple nested loop (lines 8–10). The OPTICS execution comprises an initialization of parameters (line 11), and a fit procedure that performs the clustering (line 12). Then, a DBCV index is computed for each output (line 13). DBCV assesses the clustering quality of density-based and arbitrarily

²Available in <https://git.io/JUgAm>

³<https://scikit-learn.org/stable/modules/generated/sklearn.cluster.OPTICS>

⁴*City-block distance* (or Manhattan distance) calculates the absolute differences between the coordinates of two points, x and y , in an n -dimensional space, by $d(x, y) = \sum_{i=1}^n |x_i - y_i|$ [74].

⁵*Cosine distance* determines the angular cosine distance between two vectors, u and v , in an n -dimensional space, by $d(u, v) = \frac{\sum_{i=1}^n (x_i \times y_i)}{(\sum_{i=1}^n x_i^2)^{\frac{1}{2}} \times (\sum_{i=1}^n y_i^2)^{\frac{1}{2}}}$ [51].

shaped clusters, resulting in a numeric value between -1 and 1, the higher the value the better the quality [88]. Thus, the comparison between DBCV indexes (line 14) aims to get the best input parameters by distance metric (lines 15–18).

```

1 def calculate_parameters_and_clusters(X):
2     min_samples_array = [10,11,12,13,14,15,16,17,18,19,20]
3     min_cluster_size_array = [.01, .05, .1, .15, .2]
4     metric_array = ['cityblock', 'cosine', 'euclidean']
5     parameters = [10, .01]
6     best_metric = "no metric"
7     score = -math.inf
8     for s in min_samples_array:
9         for c in min_cluster_size_array:
10             for m in metric_array:
11                 clust = OPTICS(min_samples = s,
12                               min_cluster_size = c, metric = str(m))
13                 clust.fit(X)
14                 optics_score = hdbscan.validity_index(X, clust.
15                                                         labels_[clust.ordering_], metric = m)
16                 if (optics_score > score):
17                     parameters = [s, c]
18                     score = optics_score
19                     best_clust = clust
20                     best_metric = m
21                     print('DBCV index:', score)
22                     print('\tOPTICS parameters:', parameters)
23                     print('\tMetric:', best_metric)
24     return parameters, best_clust

```

Source Code 5.1: Fragment of the Python script for clustering, highlighting the procedure to find the best input parameters to OPTICS execution

Then, for each pair of best input parameters and computed DBCV by metric, we registered the number of output clusters, as summarized in Table 5.1. It is worth noting that, although it is not the best result in terms of DBCV, 23.7% of the sample's pull requests were clustered when applying Euclidean distance as metric. We took into account this percentual of clustered pull requests for selecting the input parameters to OPTICS execution, because those clusters may promote a more significant ARL output.

Table 5.1: Output from the selection strategy of OPTICS's input parameters

<i>min_samples</i>	<i>min_cluster_size</i>	<i>metric</i>	<i>DBCV</i>	<i>Number of clusters</i>	<i>Percentual %</i>
16	0.01	'cityblock'	-0.013	2	2.5
19	0.01	'cosine'	-0.076	5	7.2
19	0.01	'euclidean'	-0.216	4	23.7

Phase 4. Interpretation of the results

At this point, we performed statistical tests, with a significance level of 0.05, in order to identify significant differences between clusters and noise by exploring the selected features. Thereby, each of them was investigated according to the procedure, described as follows. Checkings 1 and 2 are needed to examine the assumptions for parametric statistical tests. It is worth mentioning that the independence assumption is met, since all clusters are mutually exclusive, that is, a pull request belongs only to one cluster.

1. checking for data normality through Shapiro-Wilk test;
2. checking for homogeneity of variances via Levene's test;
3. performing of either parametric (ANOVA) or non-parametric (Kruskal-Wallis) test, in line with the outputs from 1 and 2; and
4. conducting of post-hoc tests, either Tukey Honestly Significant Difference (HSD) when applying ANOVA, or Dunn's test for Kruskal-Wallis, according to the output from 3.

5.1.2 Association Rule Learning

Phase 1. Selection of Features

For the ARL process, we selected the same quantitative features regarding changes, code review, and refactorings used in the clustering. We considered those features because the clusters are the input to the ARL process.

Phase 2. Feature Engineering

Since ARL algorithms typically require categorical data as input [54], the feature engineering consisted of one-hot encoding based on the quartiles of the selected features for the clustering, resulting in the binning of all features as defined in Table 5.2. The one-hot encoding was chosen for this purpose because of its simplicity, and linear time and space complexities [134].

Thus, we applied feature engineering to the following features: number of commits, number of changed files, number of added lines, number of deleted lines, number of reviewers, number of review comments, length of discussion, time to merge, and number of refactorings. At that point, we did not discard the outliers because they can potentially indicate special situations for further examination. Consequently, the *very high* category (fourth quartile) embraces outliers.

Table 5.2: Proposed one-hot encoding for binning of features

<i>Category</i>	<i>Range</i>
None	0
Low	$0 < quantile \leq 0.25$
Medium	$0.25 < quantile \leq 0.50$
High	$0.50 < quantile \leq 0.75$
Very high	$0.75 < quantile \leq 1.0$

Phase 3. Selection and Execution of an ARL Algorithm

We selected FP-Growth algorithm for rule-based learning purposes due to its performance, since it needs only two scans of the input data [62]. Then, we developed a Python script⁶ for the ARL execution on clusters, by using the FP-growth implementation available in the module *mlxtend.frequent_patterns*⁷ [104]. We experimentally set the FP-growth's support threshold minimum to 0.1, for avoiding the discard of likely associations for further analysis as argued by Coenen et al. [43]. In order to

⁶Available in <https://git.io/JUMcN>

⁷http://rasbt.github.io/mlxtend/api_subpackages/mlxtend.frequent_patterns/

get meaningful association rules, we set minimum thresholds to the following interest-
ingness metrics: confidence ≥ 0.8 , lift > 1 , and conviction > 1 . We considered that
confidence threshold minimum because lower values provide association rules that are
not very accurate [78]; whereas that lift threshold minimum reveals useful association
rules [32], and the conviction threshold minimum denotes the implication instead of
the co-occurrence of association rules [36].

Phase 4. Interpretation of Results

It is worth noting that we considered the features' levels (*none*, *low*, *medium*, *high*, and
very high), instead of absolute values, as items for composing association rules aiming
to identify relative associations among clusters for further investigation, for instance,
 $\{\text{high number of added lines}\} \rightarrow \{\text{high number of reviewers}\}$. In the case of presence
of only three levels to some feature, in any cluster, we assumed the median value to
determine the corresponding range, that is, assigning the *medium* range to the level
containing the median value. For example, since the number of reviewers is 3 on median
in Cluster 1, we considered the mapping *medium* (3), *high* (4), and *very high* (5 – 6).

The association rules work as basis for the hypotheses regarding differences/sim-
ilarities between refactoring-inducing and non-refactoring-inducing pull requests. In
this sense, we carried out the following procedure to deal with the association rules:

1. manual inspection of the association rules intending to recognize potential differ-
ences/similarities that support the formulation of hypotheses;
2. analysis of the pairwise association rules in order to support the rationale for the
formulation of hypotheses; and
3. formulation of hypotheses to quantitatively investigate the differences between
refactoring-inducing and non-refactoring-inducing pull requests.

5.1.3 Data Analysis

The **data analysis** (Step 3) consists of methods triangulation in the following setting:

- statistical testing of hypotheses, and

- manual inspection of a sample of review comments.

We used methods triangulation aiming to deal with constraints concerning the study's validity. It can reduce deficiencies from any single method by considering multiple methods of data analysis [112]. Next, we describe the design of the statistical testing of hypotheses. We propose a manual inspection of review comments based on *content analysis* because it supports the identification of purposes and effects of the content, allowing us to make qualitative inferences [75]. Specifically, we describe the design for the qualitative analysis as part of the next steps of our research in Chapter 6.

Statistical Testing of Hypotheses

The statistical analysis encompassed the testing of hypotheses formulated from the analysis of the ARL output (Step 2), driven by a comparison between two groups: 2,100 refactoring-inducing pull requests and 2,425 non-refactoring-inducing pull requests. We executed each hypothesis testing in line with the following workflow [38]:

1. Definition of null and alternative hypotheses.
2. Performing of statistical test. For that, we considered a statistical significance of 5%, and a substantive significance (effect size) for denoting the magnitude of the differences between refactoring-inducing and non-refactoring-inducing pull requests at the population level. First, we checked the assumptions for parametric statistical tests, (a) and (b). It is worth observing that the independence assumption is already met, since two groups are mutually exclusive, that is, a sample's pull request is a refactoring-inducing pull request or a non-refactoring-inducing pull request. For exploring differences between refactoring-inducing and non-refactoring-inducing pull requests, we computed a 95% confidence interval by bootstrapping resample for the difference, according to the output from (a) and (b), in mean or median (c). Then, we conducted a proper statistical test and calculated the effect size (d).

(a) checking for data normality through Shapiro-Wilk test;

- (b) checking for homogeneity of variances via Levene's test;
 - (c) computation of confidence interval for the difference in mean or median, based on the output from (a) and (b); and
 - (d) performing of either parametric independent t-test and Cohen's d , or non-parametric Mann Whitney U test and *Common-Language Effect Size* (CLES) in line with the output from (a) and (b). CLES is the probability, at the population level, that a randomly selected observation from a sample will be greater than a randomly selected observation from another sample [86].
3. Deciding whether the null hypothesis is supported or refused.

5.1.4 Validity and Reliability

We list in Table 5.3 the procedures proposed to deal with the issues of **validity and reliability** regarding data interpretation. In particular, we consider methods triangulation for the purpose of increasing the validity of the study [113].

Table 5.3: Validity and reliability procedures

<i>Type</i>	<i>Description</i>
Internal validity	Proposal of a research design to guide the data processing and data analysis procedures
Construct validity	Establishment of a chain of evidence based on the methods triangulation (quantitative and qualitative data analysis)
Reliability	Effort towards clarifying the data processing and data analysis procedures to enable replications

It is worth mentioning that in the clustering and ARL processes, we selected algorithms following previously established criteria. In the clustering process, we considered

a balance between the number of clustered pull requests and a quality index of clustering to select the best input parameters. We propose a manual inspection of review comments by using context analysis, which is intrinsically structured as a systematic procedure that can be replicated [75], intending more reliability. In order to reduce research bias, the coding step of the manual inspection of review comments will be conducted by two other coders in a way that the agreement assessment will be properly reported.

5.2 Results and Discussion

5.2.1 Clustering

After the selection of features, feature engineering, and computing the input parameters to OPTICS execution (`min_samples = 19`, `min_cluster_size = 0.01`, and `metric = 'euclidean'`), as explained in the research design (Subsection 5.1.1), we ran the clustering script and obtained the OPTICS's reachability plot displayed in Figure 5.2. The linear ordering of data points is on the x-axis, while the computed reachability distance is on the y-axis. Clusters are constituted of closest points with low reachability distance to their nearest neighbor. From that, it is possible identifying four clusters⁸ (colored points), labeled as Cluster 0, Cluster 1, Cluster 2 and Cluster 3, and noise (black points).

Table 5.4 registers the number of pull requests by cluster and respective percentual in relation to the total number of sample's pull requests (4,525). Thus, 76.3% (3,452) pull requests were not clustered. Clusters 0, 1, and 2 are composed of refactoring-inducing pull requests, whereas cluster 3 encompasses solely non-refactoring-inducing pull requests. In short, 8.6% (181) of the refactoring-inducing pull requests and 36.8% (892) of the non-refactoring-inducing pull requests were clustered.

⁸The complete clustering output is available in <https://git.io/JU6DS>

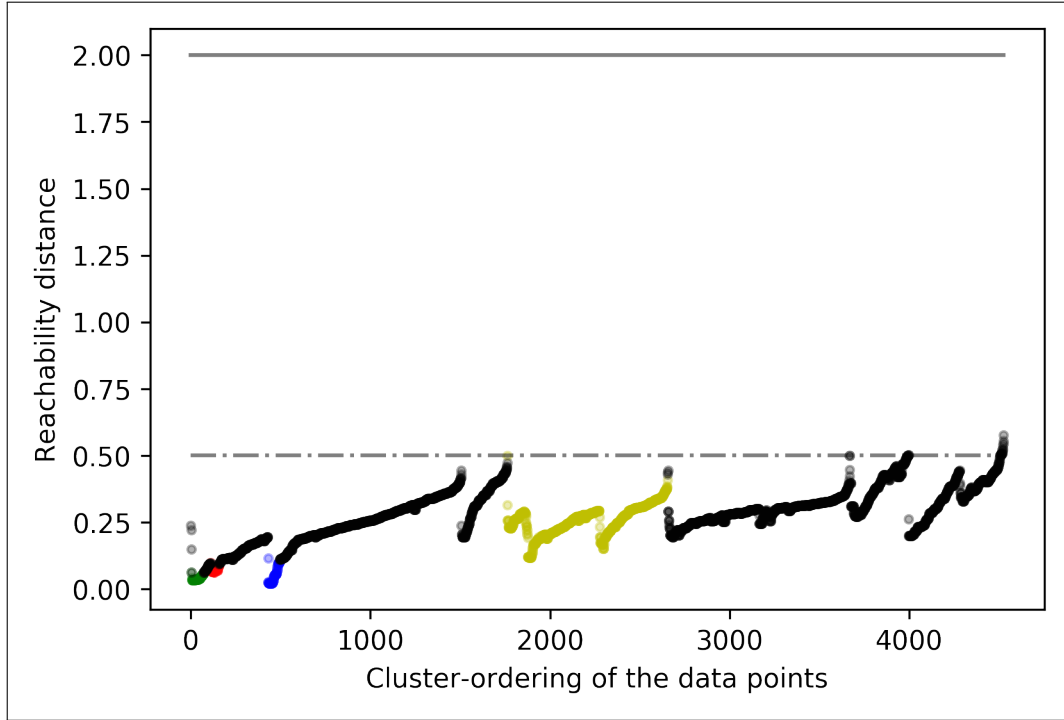


Figure 5.2: OPTICS's reachability plot

Table 5.4: Clustering output

<i>Cluster label</i>	<i>Number of pull requests</i>	<i>Percentual (%)</i>
Cluster 0	68	1.50
Cluster 1	47	1.04
Cluster 2	66	1.46
Cluster 3	892	19.70

Figure 5.3 displays the distribution of the features by cluster label, where -1 indicates noise. In an overview, the clusters exhibit differences between them, mainly by contrasting clusters consisting of refactoring-inducing pull requests (Cluster 0, Cluster 1, and Cluster 2) against Cluster 3, which comprises non-refactoring-inducing pull requests. Overall, the clusters exhibit differences between them, mainly when contrasting clusters consisting of refactoring-inducing pull requests (Cluster 0, Cluster 1, and Cluster 2) against Cluster 3, which comprises non-refactoring-inducing pull requests.

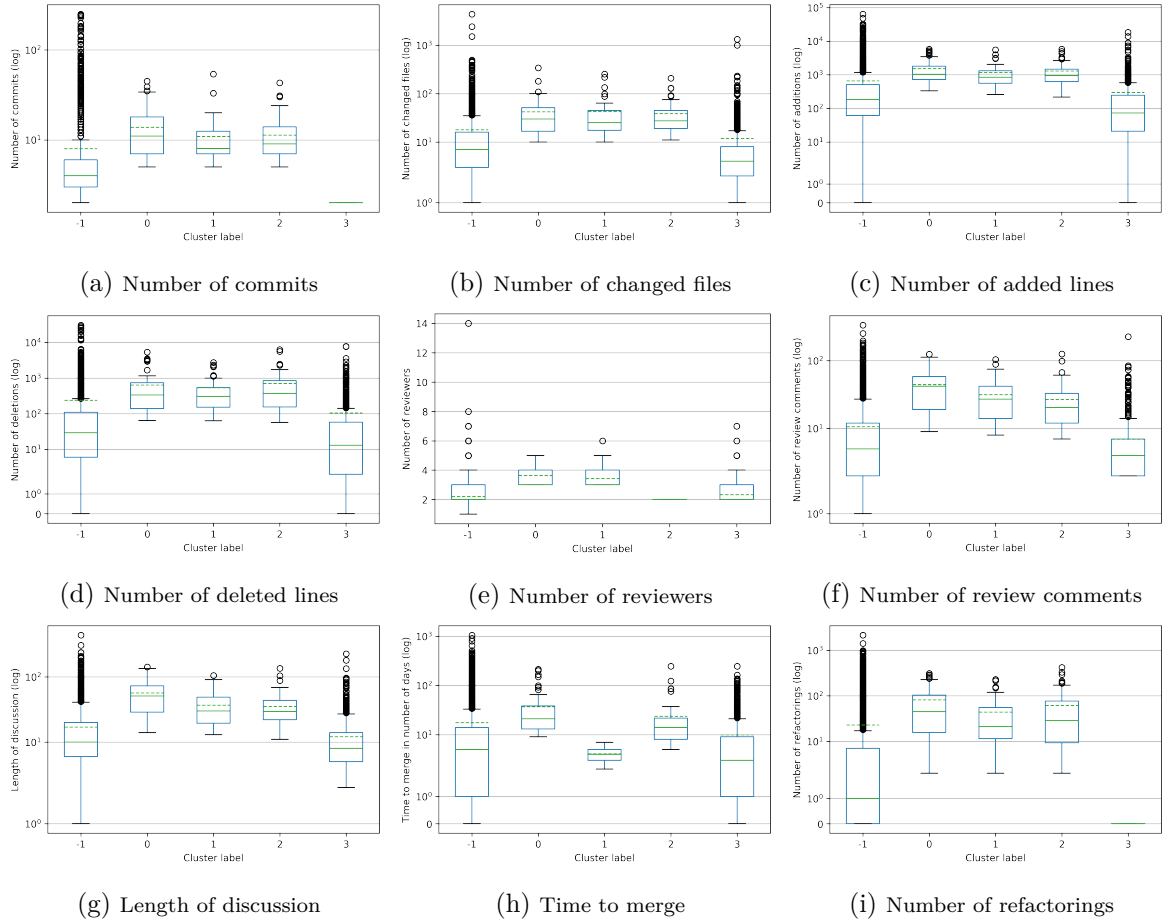


Figure 5.3: Distribution of selected features by cluster

Precisely, clusters have not met the parametric tests' assumptions for data normality and homogeneity of variances, so we applied Kruskal-Wallis test and post-hoc Dunn's test in all explorations. We detail the results of checking for parametric test assumptions and the performing of statistical tests in Appendix D⁹.

Accordingly, there is significant difference among clusters, as determined by Kruskal-Wallis tests. However, according to post-hoc Dunn's tests, there is no significant difference among Cluster 0, Cluster 1, and Cluster 2 that, in turn, are composed by refactoring-inducing pull requests. In particular, this non-significant difference raised in terms of changes (number of commits, number of changed files, number of added lines, and number of deleted lines), review (number of review comments, and length of discussion), and number of refactorings. This result suggests that number of reviewers

⁹Available in <https://git.io/JtsR5>

and time to merge are potentially the main aspects influencing the clustering.

Given that, we propose a high-level characterization of the clusters, defined in Table 5.5. For that, we examined the features in light of these perspectives: changes, code reviews, and refactorings with the basis on median values (hence, the terms more/less, larger/smaller, and higher/lower).

Table 5.5: High-level characterization of the clusters

<i>Cluster label</i>	<i>Number of pull requests</i>	<i>High-level characterization</i>
Cluster 0	68	Pull requests with more changes, larger proportion of review comments, higher number of reviewers, refactoring edits, and merged in a higher number of days
Cluster 1	47	Pull requests with more changes, larger proportion of review comments, higher number of reviewers, refactoring edits, and merged in a lower number of days
Cluster 2	66	Pull requests with more changes, larger proportion of review comments, refactoring edits, and merged in a higher number of days
Cluster 3	892	Pull requests with less changes, smaller proportion of review comments, and no refactoring edits

This high-level characterization of clusters denotes a difference in terms of the magnitude of changes and the proportion of review comments when comparing refactoring-inducing pull requests (Clusters 0, 1, and 2) and non-refactoring-inducing ones (Cluster 3). Thus, it can potentially support the elaboration of hypotheses regarding differences between refactoring-inducing and non-refactoring-inducing pull requests.

5.2.2 Association Rule Learning

We obtained the number of association rules by cluster listed in Table 5.6 from the FP-growth execution¹⁰.

Table 5.6: Number of associations obtained by cluster

<i>Cluster label</i>	<i>Number of association rules</i>
Cluster 0	18
Cluster 1	80
Cluster 2	24
Cluster 3	207
<i>Total</i>	<i>329</i>

Then, we manually inspected the association rules generated as from clusters containing entirely refactoring-inducing pull requests (Cluster 0, Cluster 1, and Cluster 2) and non-refactoring-inducing pull requests (Cluster 3), searching for pairwise association rules. As result, we found 15 association rules from Cluster 0, Cluster 1, and Cluster 2, and 13 association rules from Cluster 3 registered, in decreasing order of conviction, in Table 5.7.

Table 5.7: Association rules selected by manual inspection of ARL output (A1–A15 from refactoring-inducing pull requests and A16–128 from non-refactoring-inducing pull requests)

<i>Id</i>	<i>Association rule</i>	<i>supp</i>	<i>conf</i>	<i>lift</i>	<i>conv</i>
A1	{high no. of reviewers, high no. of review comments} → {high length of discussion}	0.18	1.0	3.78	∞
A2	{high no. of reviewers, very high no. of commits, high no. of review comments} → {high length of discussion}	0.10	1.0	3.78	∞
A3	{high no. of added lines, high no. of commits} → {high no. of reviewers}	0.11	1.0	1.47	∞

Continued on next page

¹⁰The complete ARL output is available in <https://git.io/JUyzD>

Table 5.7 – continued from previous page

<i>Id</i>	<i>Association rule</i>	<i>supp</i>	<i>conf</i>	<i>lift</i>	<i>conv</i>
A4	{high length of discussion, high no. of commits} → {high no. of reviewers}	0.11	1.0	1.47	∞
A5	{very high time to merge, very high no. of review comments} → {very high length of discussion}	0.13	1.0	3.61	∞
A6	{high no. of changed files, high no. of commits} → {high no. of reviewers}	0.10	1.0	1.47	∞
A7	{very high length of discussion} → {very high no. of review comments}	0.25	0.92	3.34	9.40
A8	{very high no. of commits, high no. of review comments} → {high length of discussion}	0.12	0.89	3.36	6.62
A9	{high length of discussion} → {high no. of reviewers}	0.23	0.89	1.73	4.37
A10	{very high time to merge, high no. of refactorings} → {high no. of commits}	0.13	0.86	2.24	4.32
A11	{very high time to merge, high length of discussion} → {high no. of reviewers}	0.10	0.87	1.70	3.89
A12	{high no. of added lines} → {high no. of reviewers}	0.23	0.92	1.35	3.83
A13	{medium length of discussion, medium no. of review comments} → {high no. of reviewers}	0.13	0.86	1.26	2.23
A14	{high no. of deleted lines, medium no. of additions} → {high no. of reviewers}	0.11	0.83	1.22	1.91
A15	{high no. of review comments} → {high no. of reviewers}	0.19	0.82	1.20	1.75
A16	{medium no. of review comments, low length of discussion} → {high no. of reviewers}	0.19	0.98	1.30	11.30
A17	{low length of discussion} → {high no. of reviewers}	0.22	0.97	1.30	9.94
A18	{medium no. of commits, low length of discussion} → {high no. of reviewers}	0.22	0.97	1.30	9.94
A19	{low length of discussion} → {medium no. of review comments}	0.20	0.91	2.24	6.53
A20	{low length of discussion} → {medium no. of review comments, medium no. of commits}	0.20	0.91	2.24	6.53

Continued on next page

Table 5.7 – continued from previous page

<i>Id</i>	<i>Association rule</i>	<i>supp</i>	<i>conf</i>	<i>lift</i>	<i>conv</i>
A21	{low length of discussion} → {high no. of reviewers, medium no. of review comments}	0.19	0.89	2.31	5.54
A22	{low length of discussion} → {high no. of reviewers, medium no. of review comments, medium no. of commits}	0.19	0.89	2.31	5.54
A23	{medium no. of review comments} → {high no. of reviewers}	0.38	0.95	1.26	4.78
A24	{medium no. of review comments, high time to merge, none refactorings, medium no. of commits} → {high no. of reviewers}	0.10	0.92	1.23	3.20
A25	{low no. of added lines, medium no. of deleted lines} → {high no. of reviewers}	0.11	0.85	1.14	1.73
A26	{low no. of added lines} → {high no. of reviewers}	0.21	0.85	1.14	1.70
A27	{low no. added lines, medium no. of commits} → {high no. of reviewers}	0.21	0.85	1.14	1.70
A28	{low no. of changed files, medium no. of commits} → {high no. of reviewers}	0.19	0.81	1.09	1.35

We carried out an analysis of the pairs of association rules, in light of the high-level classification of clusters (Table 5.5), aiming to elaborate the rationale for the formulation of hypotheses on the differences/similarities between refactoring-inducing and non-refactoring-inducing pull requests. Accordingly, we propose the following hypotheses.

H₁ Refactoring-inducing pull requests are more likely to have a higher number of added lines than non-refactoring-inducing pull requests.

Refers to association rules: (A3, A27), (A12, A26), (A14, A25)

H₂ Refactoring-inducing pull requests are more likely to have a higher number of deleted lines than non-refactoring-inducing pull requests.

Refers to association rules: (A14, A25)

Rationale: Having as reference the same number of reviewers, those association

rules suggest that the changes in refactoring-inducing pull requests tend to encompass a higher number of added lines and a higher number of deleted lines than in non-refactoring-inducing pull requests.

H₃ Refactoring-inducing pull requests are more likely to have a higher number of changed files than non-refactoring-inducing pull requests.

Refers to association rules: (A6, A28)

Rationale: In case of the same number of reviewers, those association rules indicate that refactoring-inducing pull requests tend to comprise changes arranged across a higher number of files than non-refactoring-inducing ones.

H₄ Refactoring-inducing pull requests are more likely to have a higher number of commits than non-refactoring-inducing pull requests.

Refers to association rules: (A2, A22), (A3, A27), (A4, A18), (A6, A28)

Rationale: When regarding the same number of reviewers, those association rules suggest that refactoring-inducing pull requests tend to include a higher number of commits than in non-refactoring-inducing pull requests.

H₅ Refactoring-inducing pull requests are more likely to have a higher number of review comments than non-refactoring-inducing pull requests.

Refers to association rules: (A1, A21), (A2, A22), (A7, A19), (A8, A20), (A15, A23)

Rationale: When considering the same number of reviewers, in (A1, A21), (A2, A22), (A15, A23), and regarding the gradual increase in the number of review comments, in (A7, A19), (A8, A20), those association rules propose that code review in refactoring-inducing pull requests tend to encompass a higher number of review comments than in non-refactoring-inducing pull requests.

H₆ Refactoring-inducing pull requests are more likely to have a lengthier discussion than non-refactoring-inducing pull requests.

Refers to association rules: (A1, A21), (A2, A22), (A4, A18), (A7, A19), (A8, A20), (A9, A17), (A13, A16)

Rationale: In the case of the same number of reviewers, in (A1, A21), (A2, A22), (A4, A18), (A9, A17), (A13, A16), and considering the gradual increase in the length of discussion, in (A7, A19), (A8, A20), those association rules suggest that code review in refactoring-inducing pull requests tend to contain a lengthier discussion than in non-refactoring-inducing pull requests.

H₇ Refactoring-inducing and non-refactoring-inducing pull requests are equally likely to have a higher number of reviewers.

Refers to association rules: (A1, A21), (A2, A22), (A3, A27), (A4, A18), (A6, A28), (A9, A17), (A12, A26), (A14, A25), (A15, A23)

Rationale: Those association rules suggest no difference between refactoring-inducing and non-refactoring-inducing pull requests as for the number of reviewers.

H₈ Refactoring-inducing pull requests are more likely to take a higher time to merge than non-refactoring-inducing pull requests.

Refers to association rules: (A5, A10, A11, A24)

Rationale: Together, A5, A10, and A11, in comparison to A24 (by considering the number of review comments, reviewers, refactoring edits, and commits) suggest that refactoring-inducing pull requests take more time to merge than non-refactoring-inducing ones.

5.2.3 Statistical Testing of Hypotheses

We present the results of the testing of hypotheses and discuss them, follows. We also provide more details in Appendix E¹¹.

¹¹Available in <https://git.io/JtsRh>

H₁ Refactoring-inducing pull requests are more likely to have a higher number of added lines than non-refactoring-inducing pull requests

The number of added lines is 813.9 on average (SD = 2,117.1), 315 on median (IQR = 674) in refactoring-inducing pull requests, and 437.8 on average (SD = 2,073.6), 96 on median (IQR = 248) in non-refactoring-inducing pull requests.

Result. There is a statistically significant difference in the number of added lines between refactoring-inducing and non-refactoring-inducing pull requests, 95% CI [189.9, 240.5]. Refactoring-inducing pull requests are significantly more likely to have a higher number of added lines than non-refactoring-inducing pull requests ($U = 3.55 \times e^{+6}$, $p < .05$). There is a 69.7% probability that a random observation of the number of added lines from refactoring-inducing pull requests will be higher than one from non-refactoring-inducing pull requests, at the population level.

H₂ Refactoring-inducing pull requests are more likely to have a higher number of deleted lines than non-refactoring-inducing pull requests

The number of deleted lines is 299.6 on average (SD = 1,376.6), 53 on median (IQR = 176) in refactoring-inducing pull requests, and 163.5 on average (SD = 1,064.2), 15 on median (IQR = 60) in non-refactoring-inducing pull requests.

Result. There is a statistically significant difference in the number of deleted lines between refactoring-inducing and non-refactoring-inducing pull requests, 95% CI [31.0, 43.51]. Refactoring-inducing pull requests are significantly more likely to have a higher number of deleted lines than non-refactoring-inducing pull requests ($U = 3.32 \times e^{+6}$, $p < .05$). There is a 65.2% probability that a random observation of the number of deleted lines from refactoring-inducing pull requests will be higher than one from non-refactoring-inducing ones, at the population level.

Discussion (H₁ and H₂). According to those results, code reviewing in refactoring-inducing pull requests comprises more code churn than in non-refactoring-inducing pull requests. The median number of added lines in refactoring-inducing pull requests is 228.1% higher than in non-refactoring-inducing ones. When examining the number of deleted lines, the percentual is 253.3%. This is an expected result, if we consider the

findings from Hegedüs et al., since refactored code has significantly higher size-related metrics [63], when evaluating the effects of refactoring on software maintainability. Nevertheless, we speculate that reviewing larger code churn may potentially promote refactoring edits. This understanding is supported by findings from Rigby et al. who observed that the code churn magnitude influences code reviewing, when studying code review practices in OSS projects [109; 108], and Beller et al. who discovered that the larger the code churn, the more changes could follow, while analyzing code review practices of OSS projects in the Gerrit system [31].

Finding 1. Refactoring-inducing pull requests comprise significantly more code churn than non-refactoring-inducing pull requests.

H₃ Refactoring-inducing pull requests are more likely to have a higher number of changed files than non-refactoring-inducing pull requests

The number of changed files is 21.4 on average (SD = 49.9), 10 on median (IQR = 17.2) in refactoring-inducing pull requests, and 14.3 on average (SD = 109.4), 4 on median (IQR = 8) in non-refactoring-inducing pull requests.

Result. There is a statistically significant difference in the number of changed files between refactoring-inducing and non-refactoring-inducing pull requests, 95% CI [5.0, 6.0]. Refactoring-inducing pull requests are significantly more likely to have a higher number of changed files than non-refactoring-inducing pull requests ($U = 3.46 \times e^{+6}$, $p < .05$). There is a 67.9% probability that a random observation of the number of changed files from refactoring-inducing pull requests will be higher than one from non-refactoring-inducing pull requests, at the population level.

Discussion. The median number of changed files in refactoring-inducing pull requests is 150% higher than in non-refactoring-inducing ones. We conjecture that reviewing code arranged across files may motivate refactoring edits, an argument supported by the conclusions from Beller et al. regarding more changed files comprise more changes during code review [31].

Finding 2. Refactoring-inducing pull requests encompass significantly more changed files than non-refactoring-inducing pull requests.

H₄ Refactoring-inducing pull requests are more likely to have a higher number of commits than non-refactoring-inducing pull requests

The number of commits is 11.1 on average (SD = 25.9), 5 on median (IQR = 6) in refactoring-inducing pull requests, and 3.4 on average (SD = 2.9), 3 on median (IQR = 2) in non-refactoring-inducing pull requests.

Result. There is a statistically significant difference in the number of commits between refactoring-inducing and non-refactoring-inducing pull requests, 95% CI [2.0, 3.0]. Refactoring-inducing pull requests are significantly more likely to have a higher number of commits than non-refactoring-inducing pull requests ($U = 3.86 \times e^{+6}$, $p < .05$). There is a 75.7% probability that a random observation of the number of commits from refactoring-inducing pull requests will be higher than one from non-refactoring-inducing pull requests, at the population level.

Discussion. Based on our previous findings on the magnitude of code churn and file changes in refactoring-inducing pull requests, that result is expected and aligned to Beller et al. concerning the impacts of larger code churn and wide-spread changes across files on consequent changes [31]. Accordingly, we speculate that reviewing refactoring-inducing pull requests might require more changes, in turn, denoted by more commits in comparison with non-refactoring-inducing pull requests.

Finding 3. Refactoring-inducing pull requests comprise significantly more commits than non-refactoring-inducing pull requests.

H₅ Refactoring-inducing pull requests are more likely to have a higher number of review comments than non-refactoring-inducing pull requests

The number of review comments is 15.1 on average (SD = 22.1), 8 on median (IQR = 14) in refactoring-inducing pull requests, and 7.1 on average (SD = 10.8), 4 on median

(IQR = 6) in non-refactoring-inducing pull requests.

Result. There is a statistically significant difference in the number of review comments between refactoring-inducing and non-refactoring-inducing pull requests, 95% CI [3.0, 4.0]. Refactoring-inducing pull requests are significantly more likely to have a higher number of review comments than non-refactoring-inducing pull requests ($U = 3.34 \times 10^6$, $p < .05$). There is a 65.6% probability that a random observation of the number of review comments from refactoring-inducing pull requests will be higher than one from non-refactoring-inducing pull requests, at the population level.

Discussion. This result differs of the findings from Paixao et al. that identified less feedback of reviewing in the presence of refactorings, when studying the intents behind those refactorings during code review, in projects using Gerrit [95]. On the other hand, Beller et al. found that the most of changes during code review is driven by review comments [31]. Given that, we speculate that the GitHub pull requests can comprise a peculiar structure of code review, in which review comments influence the occurrence of refactorings, therefore explaining our result. This argument originates from the fact that GitHub pull-based collaboration workflow provides reviewing resources [7], such as a proper code reviewing *User Interface* (UI), for developers to channel efforts for improving/fixing the code within a pull request, while having access to the history of commits, review comments, and discussion. In addition, our result provides insight for further content analysis of review comments aiming to identify evidence of refactoring-inducement in refactoring-inducing pull requests.

Finding 4. Refactoring-inducing pull requests embrace significantly more review comments than non-refactoring-inducing pull requests.

H₆ Refactoring-inducing pull requests are more likely to have a lengthier discussion than non-refactoring-inducing pull requests

The length of discussion is 22.1 on average (SD = 27.3), 13 on median (IQR = 20) in refactoring-inducing pull requests, and 12.6 on average (SD = 15.2), 8 on median (IQR = 10) in non-refactoring-inducing pull requests.

Result. There is statistically significant difference in the length of discussion between refactoring-inducing and non-refactoring-inducing pull requests, 95% CI [5.0, 6.0]. Refactoring-inducing pull requests are significantly more likely to have a larger length of discussion than non-refactoring-inducing pull requests ($U = 3.26 \times e^{+6}$, $p < .05$). There is a 62.3% probability that a random observation of the length of discussion from refactoring-inducing pull requests will be larger than one from non-refactoring-inducing pull requests, at the population level.

Discussion. Apache’s refactoring-inducing pull requests from our sample present a length of discussion superior in relation to the result found by Gousios et al., when considering average and median [59]. Although our result does not include the intents behind refactorings, it corroborates with Paixão et al. in the context of that refactorings with non-explicit intent demand more discussion [84]. Given that, we conjecture that more discussion expresses consideration to valuable review comments that, in turn, may potentially induce refactorings. Our argument is mainly based on the findings from Lee and Cole, when studying the Linux kernel development, acknowledged that the amount of discussion is a code quality indicator [79].

Finding 5. Refactoring-inducing pull requests enclose significantly more discussion than non-refactoring-inducing pull requests.

H₇ Refactoring-inducing and non-refactoring-inducing pull requests are equally likely to have a higher number of reviewers

The number of reviewers is 2.4 on average (SD = 1.1), 2 on median (IQR = 1) in refactoring-inducing pull requests, and 2.1 on average (SD = 0.8), 2 on median (IQR = 0) in non-refactoring-inducing pull requests.

Result. There is no statistically significant difference in the number of reviewers between refactoring-inducing and non-refactoring-inducing pull requests (95% CI [0.0, 0.0]). refactoring-inducing and non-refactoring-inducing pull requests are significantly likely to have a higher number of reviewers ($U = 2.95 \times e^{+6}$, $p < .05$). There is a 57.9% probability that a random observation of the number of reviewers from refactoring-

inducing pull requests will be as high as one from non-refactoring-inducing pull requests, at the population level.

Discussion. There is no statistical evidence of a difference in terms of the number of reviewers, because both refactoring-inducing and non-refactoring-inducing pull requests present two reviewers as median – the same result found by Rigby et al. [106] in the OSS scenario. There are outliers (Appendix E¹²) that, in turn, can be justified by other technical factors, such as complexity of changes, as argued in [107]. Nevertheless, this thesis proposal does not address that scope.

Finding 6. We found no statistical evidence that the number of reviewers is related to refactoring-inducement.

H₈ Refactoring-inducing pull requests are more likely to take a higher time to merge than non-refactoring-inducing pull requests

The time to merge is 19.6 on average (SD = 49.1), 6 on median (IQR = 15) days in refactoring-inducing pull requests, and 13.1 on average (SD = 42.5), 4 on median (IQR = 9) days in non-refactoring-inducing pull requests.

Result. There is statistically significant difference in time to merge between refactoring-inducing and non-refactoring-inducing pull requests (95% CI [2.0, 3.0]). Refactoring-inducing pull requests are significantly more likely to take a larger time to merge than non-refactoring-inducing pull requests ($U = 3.02 \times e^{+6}$, $p < .05$). There is a 59.4% probability that a random observation of time to merge from refactoring-inducing pull requests will be larger than one from non-refactoring-inducing pull requests, at the population level.

Discussion. Our sample's pull requests take 5 days, on median, to merge, differing from the discoveries about the brief duration of reviews (from 19.8 hours to 2.7 days, in 75% of cases) accomplished by Rigby et al., when studying OSS peer review [108]. We understand that difference is because the projects, studied by Rigby

¹²Available in <https://git.io/JtsRh>

and colleagues, were based on patch-based contribution tools (mailing lists) instead of a pull-based contribution strategy of code reviewing. We realize the impact of refactoring edits on the time to merge of refactoring-inducing pull requests, concluding that both time of code reviewing and of performing refactoring edits impact the time to merge a refactoring-inducing pull request. In particular, this conclusion is aligned to the findings of Szoke et al., who observed a correlation between implementing refactorings and time [121], and from Gousios et al., who found that review comments and discussion affect time to merge a pull request [59]. As argued by Kononenko et al., size-related factors influence the reviewing time [72; 73]. Accordingly, we also consider the impact of greater code churn and changed files, as it occurs in refactoring-inducing pull requests, on time to merge.

Finding 7. Refactoring-inducing pull requests take significantly more time to merge than non-refactoring-inducing pull requests.

Concluding Discussion

By observing change-related aspects (churn and changed files), our findings confirm previous conclusions concerning the influence of the amount and magnitude of changes on code review [29; 31; 72; 108; 109]. Peculiarly, most of our sample's pull requests have a larger code churn and a higher number of changed files than the pull requests investigated by Gousios et al. [59], when exploring the pull-based software development model (less than 20 changed files on average and two on median, and less than 500 lines of churn and 20 on median). When analyzing the changes and refactorings, our findings support prior conclusions about refactored software present significantly higher size-related metrics (as to the number of code lines and changed files) [63], and larger changes promote refactoring edits [95].

By addressing points concerned to code review and refactorings, the occurrence of more review comments and discussion in refactoring-inducing pull requests corroborates with findings about the impact of review comments on changes [31], and the importance of discussion to the code quality [109; 79]. Precisely, the number of review comments left in refactoring-inducing pull requests (15.1 on average) is superior to the value

found by Gousios et al. [59], that is, 12 review comments or less in 95% of pull requests from GHTorrent¹³. Also, by comparing the number of review comments left in refactoring-inducing and non-refactoring-inducing pull requests, it is possible to identify a divergence relative to the trend of less feedback in the presence of refactoring edits, noticed by Paixão et al. when exploring OSS projects on Gerrit [95].

Therefore, the statistical analysis of code review data revealed significant differences between refactoring-inducing and non-refactoring-inducing pull requests. Moreover, the divergences identified in relation to other studies on code review indicate pertinence of further investigation on patterns of code review practice that would qualitatively characterize the refactoring-inducing pull requests. In particular, the results concerning the number of reviewers suggest an in-depth study of review comments driven, in turn, by the following question: *How do review comments influence refactoring-inducing pull requests?* – to be explored in the next phase of the data analysis (manual inspection of review comments). In Chapter 6, we introduce some directions towards that phase.

5.3 Limitations

This comparative study presents a few limitations, despite the efforts in the sense of planning countermeasures to deal with validity and reliability issues, as proposed in Subsection 5.1.4.

Concerning internal validity, it is worth mentioning that the research design was elaborated after conducting pre-runings of the clustering and ARL processes, including the altering of processing order until we achieve the final design. We carried out a rigorous selection of algorithms, and input parameters for unsupervised learning. When specifically considering clustering, it is worth mentioning that there are several alternatives in terms of similarity measures besides City-block distance, Cosine distance and Euclidean distance [131]. Nevertheless, we took into account only the options available in the module `sklearn.cluster`. We carefully defined workflows for our research design procedures aiming to explain all the decisions taken. Even so, there are risks of threats to internal validity due to any non-previously identified deficiencies

¹³<https://ghtorrent.org/>

in our research design.

We endeavor in the sense of the establishment of a chain of evidence for the data interpretation and description of taken decisions in the research design. Moreover, we propose methods triangulation intending to reduce deficiencies from any single method and enhance our study's validity. For those methods, we are concerned with systematically structuring all procedures aiming at replicability, so aiming reliability.

In terms of manual inspection of review comments, there are a few limitations. First, the interpretation and analysis of the review comments are labor-intensive, so we will consider a small random sample. Despite the efforts towards a rigorous analysis, it is not suitable to generalize the conclusions, except when considering other OSS projects that also follows a geographically distributed development [135] and are aligned to the principle "the Apache way"¹⁴. Moreover, the interpretation and analysis are naturally subjective, since they can vary from distinct researchers' perspective [75]. Aiming to reduce researcher bias, the manual inspection will be also carried out by two other researchers.

Overall, it is appropriate considering that Apache's projects follow particular principles of software development, as explained in Chapter 4, Subsection 4.1.1. Therefore, this is a threat to external validity, since the findings of this comparative study are exclusively extended to cases which have common characteristics with Apache's projects.

5.4 Concluding Remarks

This study is essentially exploratory, as it is aimed to investigate how refactoring-inducing pull requests differ from non-refactoring-inducing ones. In this sense, we proposed a research design to conduct clustering and ARL process on features of a dataset containing 4,525 merged pull requests. Concisely, the preliminary results of the quantitative analysis are motivating, so directing efforts towards a qualitative analysis of review comments intending to achieve an in-depth understanding of refactoring-inducing pull requests, as argued in the next Chapter.

¹⁴<https://www.apache.org/theapacheway>

Chapter 6

Next Steps

In this chapter, we present a few preliminary conclusions (Section 6.1) towards characterizing refactoring-inducing pull requests based on the results obtained from the characterization study on refactorings in refactoring-inducing pull requests (Chapter 4) and comparative study between refactoring-inducing and non-refactoring-inducing pull requests (Chapter 5). Next, we propose a design for the manual inspection of review comments (Section 6.2) that constitute the qualitative domain of the data analysis (Step 3) in our comparative study design (Chapter 5, Subsection 5.1.3). Moreover, we introduce a few considerations on the “*strict*” *refactoring-inducing pull requests*, in turn, investigated in a supplementary study (Section 6.3).

6.1 Preliminary Conclusions

Aiming to address our research questions (Chapter 1, Section 1.4), we carried out two specific studies. In the first study, we identified the properties of refactoring edits performed in merged pull requests. We found 46.4% of refactoring-inducing pull requests in our sample, in which refactorings address diversified code improvements in high, medium, and low levels of operation, while their number of edits gradually decrease across commits in refactoring-inducing pull requests. From that, we conclude that those results are motivating, since they indicate concerns with code restructurings at the pull request level, while they provide a better understanding of how refactoring-inducing pull requests operate.

In the second study, we identified differences between refactoring-inducing and non-refactoring-inducing pull requests. Accordingly, we performed a quantitative analysis of code review data and found statistically significant differences between them, particularly in terms of changes (added lines, deleted lines, changed files, and commits), and code review (review comments, length of discussion, and time to merge). However, we found no significant difference in terms of number of reviewers. From those results, we hypothesize that the content of review comments might be an influencing factor in the refactoring-inducement. Thereby, the following research question drives our efforts of qualitative investigation.

- RQ₅ How do review comments influence refactoring-inducing pull requests?

Specifically, we propose a manual inspection of review comments aiming to get an in-depth understanding of the impact of review comments on refactoring-inducing pull requests, as previously specified in the research design (Chapter 5, Subsection 5.1.3).

6.2 Planning of the Manual Inspection of Review Comments

We suggest the exploration of review comments intending to identify and interpret patterns of code review practice in refactoring-inducing pull requests, since review comments provide assets for an in-depth understanding of code reviewing, as perceived in [26; 35; 49; 73; 80; 95; 84; 97; 99; 102; 108; 129]. In this sense, we propose a manual inspection of review comments supported by *content analysis* because it supports the identification of purposes and effects of the content, allowing us to make qualitative inferences [75]. By considering data from code review comments dataset¹, we will carry out a content analysis of review comments according to the following workflow:

1. **Selecting a random sample of review comments.** Since content analysis is a time-intensive task [75], we will analyze the content of review comments from 200 refactoring-inducing pull requests and 200 non-refactoring-inducing pull requests,

¹Available in <https://git.io/JU81G>

both randomly selected. We consider four categories for analyzing: pull requests containing a low, medium, high, and very high number of review comments, in line with the binning of features defined to the ARL process (Table 5.2). Thus, we propose a sample size (50 by category for each group of pull requests) according to guidance from Roscoe [111] for the minimum sample size of 30 for each category under analysis.

2. **Defining the unit of analysis.** The *unit of analysis* consists of the basic unit of text for categorization [75]. We will consider a review comment as the unit for analysis because we understand that it denotes a complete unit of meaning, with the purpose of recognizing patterns of code review practice at refactoring-inducing and non-refactoring-inducing pull requests.
3. **Coding the data.** *Coding* constitutes the interpretation of the units of analysis aiming at categorization [75]. This step will be performed based on a cross-reference between review comments and descriptions of the detected refactorings for each sample's refactoring-inducing pull request, aiming to validate connections between a review comment and one or more refactoring edits. As for the sample's non-refactoring-inducing pull requests, we will catalog the coding emergent patterns. Expecting to ensure the consistency of our coding scheme, we will develop specific rules for the assignment of codes, assisted by examples, as recommended in [133]. Then, we will conduct a pretest of our coding scheme, in which two coders will independently code the review comments from a refactoring-inducing pull request and a non-refactoring-inducing pull request, both randomly selected. After that, we will assess the *inter-coder agreement* (agreement between coders' responses) in order to validate and make improvements in our coding scheme. For that assessment, we will compute Krippendorff's *alpha* [75] and Cohen's *kappa* [44], as recommended by Lacy et al. [76], in order to deal with the controversies over the suitability of agreement coefficients. If needed, we will adjust our coding scheme, rerun the pretest on different refactoring-inducing and non-refactoring-inducing pull requests, and assess the inter-coder agreement until we achieve a high percent-

age of agreement in both coefficients. A high agreement denotes values greater than .80 for both Krippendorff’s *alpha* and Cohen’s *kappa*, as argued in [75; 77]. Then, we will tag the review comments with codes in line with our final coding scheme.

4. **Assessing coding consistency.** This step aims to register the reliability of our coding scheme of our sample’s review comments, though computing the inter-coder agreement using Krippendorff’s *alpha* and Cohen’s *kappa*, for each group of pull requests.
5. **Drawing inferences on the coding.** At this step, we will subjectively explore the emerging patterns intending to understand how review comments influence refactoring-inducing pull requests, by contrasting them against patterns that emerged from non-refactoring-inducing pull requests’ review comments. Moreover, we will relate those inferences to the results from our previous studies.

In short, we speculate that a qualitative analysis of review comments in refactoring-inducing and non-refactoring-inducing pull requests, supported by our refactorings dataset², may provide valuable evidence in direction to an in-depth understanding of code review practices in refactoring-inducing pull requests.

6.3 Towards “Strict” Refactoring-Inducing Pull Requests

Since code review efforts take place after the opening of pull requests, we also carried out a supplementary investigation³ on “strict” refactoring-inducing pull requests, so named in this thesis proposal, defined as follows.

Definition 2. *A pull request is refactoring-inducing if refactoring operations are performed in subsequent commits after the initial pull request commit(s), as a result of the reviewing process or spontaneous improvements by the pull request contributor. Let*

²Available in <https://git.io/JU1Dk>

³In collaboration with professor Nikolaos Tsantalis (Concordia University, Canada).

$U = \{u_1, u_2, \dots, u_w\}$, a set of repositories in GitHub. Each repository u_q , $1 \leq q \leq w$, has a set of pull requests $P(u_q) = \{p_1, p_2, \dots, p_m\}$ over time. Each pull request p_j , $1 \leq j \leq m$, has a set of commits $C(p_j) = \{c_1, c_2, \dots, c_n\}$, in which $I(p_j)$ is the set of initial commits included in the pull request when it is created, $I(p_j) \subseteq C(p_j)$. A refactoring-inducing pull request is that in which $\exists c_k \mid R(c_k) \neq \emptyset$, where $R(c_k)$ denotes the set of refactorings performed in commit c_k and $|I(p_j)| < k \leq n$.

In order to clarify our definition, Figure 6.1 illustrates a refactoring-inducing pull request consisting of three initial commits ($c_1 - c_3$) and six subsequent commits ($c_4 - c_9$), three of which include refactoring operations (c_5, c_7, c_8), e.g., commit c_7 has two *Rename Class* and three *Change Variable Type* refactoring instances. Accordingly, that study explored differences/similarities between pull requests based on the refactoring edits performed in pull request commits subsequent to the initial ones ($c_4 - c_9$). Note that, when considering our previous refactoring-inducing pull request definition (Definition 1), we would investigate such pull request based on the refactoring edits performed in commits subsequent to the first one ($c_2 - c_9$).

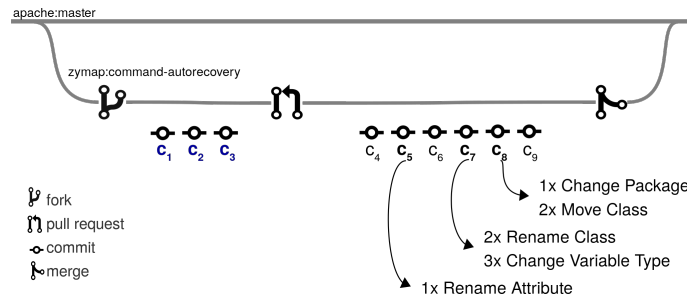


Figure 6.1: A refactoring-inducing pull request (PR #2010 from Apache’s bookkeeper repository), illustrating initial commits ($c_1 - c_3$) and subsequent commits ($c_4 - c_9$).

In that supplementary study, we selected 3,888 pull requests, which have at least one subsequent commit, from our sample (4,525 Apache’s merged pull requests). Then, we explored the following research questions in an empirical study consisting of five steps, inspired on our previous studies (Chapters 4 and 5), as depicted in Figure 6.2.

- RQ₁: How common are “strict” refactoring-inducing pull requests?

- RQ₂: How do “strict” refactoring-inducing pull requests compare to non-refactoring-inducing ones?

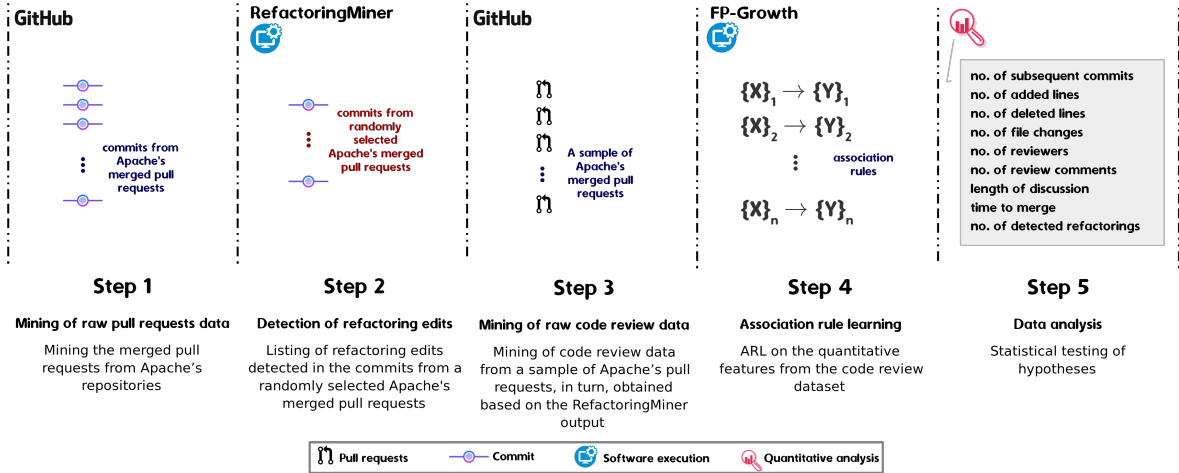


Figure 6.2: Empirical research design

Overall, the study's findings (omitted in this document) are motivating in two dimensions: practical and methodological. Practical, because they drive us regarding the pertinence of the differences between refactoring-inducing and non-refactoring-inducing pull requests even when considering changes and refactoring edits exclusively done after opening a pull request, regardless of the number of initial commits. To clarify, note that there is a subtle distinction between our original definition of the refactoring-inducing pull request (Definition 1) and the Definition 2. Originally, we consider a pull request as a unit, which we analyze as from the second commit because we understand that initial commit(s), regardless of amount, may also be reviewed after opening a pull request. In line with Definition 2, we exclusively analyze technical aspects as from the subsequent commits.

In the methodological dimension, we realized that performing clustering before ARL produces more accurate and meaningful association rules for manual inspection, even in presence of high noise, in our research context. Anyway, that extra study reinforces the suitability of ARL as a technique for supporting characterizing studies like what we propose.

Given this scenario, we suggest also contemplating those supplementary results when characterizing refactoring-inducing pull requests. Accordingly, another next step

consists of planning how to consider those results based on analyzing their impacts on our previous studies.

6.4 Concluding Remarks

The next steps of this thesis proposal concentrate efforts in the qualitative domain of the data analysis of the comparative study between refactoring-inducing and non-refactoring-inducing pull requests (Chapter 5). In this chapter, we proposed a workflow to manually inspect review comments based on the content analysis approach, so describing the phases and suggesting a few countermeasures in order to deal with reliability issues. We also argue an extra study on the strict refactoring-inducing pull requests, then offering supplementary results towards the characterization of refactoring-inducing pull requests. In the following chapter, we propose a work plan in order to address the next steps of this research.

Chapter 7






Work Plan

In this work plan, we register a complete time-bound and visual reference of the research phases (Section 7.1) to achieve the objectives of this thesis proposal (Introduction, Section 1.3), and discuss a few possible risks and respective strategies for mitigating them (Section 7.2).

7.1 Research Schedule

Table 7.1 presents the complete schedule of this thesis proposal, indicating the research phases over our timeline¹, and indicators of progress. The supervision meetings were not scheduled because they happen on demand. Moreover, a contingency buffer was programmed in order to afford a more flexible plan and accommodate unforeseen events.

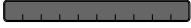

















Table 7.1: Research schedule (in quarters)

<i>Research phase</i>	<i>Period</i>	<i>Indicator of progress</i>
<i>Background research</i>		
Refactoring	Q1 2018 – Q3 2018	
Modern code review	Q4 2018 – Q1 2019	
Git-based development model	Q3 2019 – Q4 2019	
Unsupervised machine learning	Q1 2020 – Q2 2020	
<i>Research design planning</i>	Q3 2019	

Continued on next page

¹Expected thesis defense deadline: February 22, 2022.

Table 7.1 – continued from previous page

<i>Research phase</i>	<i>Period</i>	<i>Indicator of progress</i>
<i>Literature review</i>	Q4 2018, Q2 2020	
<i>Data collection</i>		
Mining of merged pull requests	Q3 2019	
Refactoring detection	Q3 2019	
Mining of code review data	Q1 2020	
<i>Quantitative analysis</i>		
Research question 1	Q4 2019	
Research question 2	Q4 2019	
Research question 3	Q3 2020	
<i>Research question 4</i>		
Clustering	Q1 2020	
Association rule learning	Q1 2020	
Statistical tests	Q3 2020	
<i>Writing (thesis proposal)</i>	Q2 2020 – Q4 2020	
<i>Impact analysis of “strict” refactoring-inducing pull requests</i>	Q1 2021	
<i>Qualitative analysis</i>		
Selection of sample	Q1 2021	
Training of coders	Q2 2021	
Data coding	Q2 2021	
Coding consistency assessment	Q2 2021	
Drawing of inferences	Q3 2021	
<i>Writing (thesis)</i>	Q2 2021 – Q3 2021	
<i>Contingency buffer</i>	Q4 2021	

Accordingly, the first two objectives of this thesis proposal (composing a dataset of code reviewing-related aspects and detected refactorings in pull requests; and identifying the differences between refactoring-inducing and non-refactoring-inducing pull requests) were achieved through mining data from GitHub and statistical data analysis. The third objective (identifying aspects of code review that influence refactoring-inducement in pull requests) is in progress, since the quantitative analysis has pointed the review comments for investigation in the qualitative analysis, whereas the fourth ob-

jective (characterizing refactoring-inducing pull requests) will be accomplished through examining, as a whole, the results from both quantitative and qualitative analysis of pull requests.

7.2 Risks

We identified two risks related to the qualitative analysis of our thesis proposal, ranked according to their potential negative impact on our work plan as follows. Firstly, the training of coders depends on the recruitment of individuals with skills and experiences, such as experience in refactoring and code review, needed to properly perform the coding of review comments. Thus, this step may require more time than planned.

Secondly, the steps of data coding and coding consistency assessment may require repetitions until we reach an appropriate coding scheme. In order to mitigate the related risks, we will conduct a practical training, supported by examples from our review comments data. Moreover, the data code will be regularly supervised to deal with unforeseen events.

7.3 Concluding Remarks

The next steps of this thesis proposal consist of qualitative analysis and writing of the thesis, as we presented in our research schedule. The associated risks directly impact time, so we proposed a few countermeasures to deal with unexpected events.

Chapter 8

Concluding Remarks

This thesis proposal has addressed a knowledge gap in the characterization of refactoring-inducing pull requests (Chapter 1) and provided a background on the underlying subjects (Chapter 2), and a connection to related works (Chapter 3). Moreover, we have proposed the research design of two complementary studies: a characterization of refactorings in refactoring-inducing pull requests (Chapter 4) and a comparison between refactoring-inducing pull requests and non-refactoring-inducing ones (Chapter 5) in light of reviewing-related aspects. For both, we have discussed the preliminary results and limitations. We have also suggested the next steps of research (Chapter 6), a work plan and countermeasures to a few risks (Chapter 7).

To the best of our knowledge, this is the first research work exploring aspects related to refactorings and code review in the context of refactoring-inducing pull requests (Definition 1). For that, we have investigated merged pull requests independently of the merge type applied since we implemented the strategy to miner squashed pull requests (Appendix B). Also, we have made available a complete kit of reproduction [18] consisting of the data mined and tools implemented intending to enable replications and future research.

Thus, we have presented the efforts leveraged to investigate technical aspects characterizing refactoring-inducing pull requests. By now, we have achieved two of our objectives:

- composing a dataset of changes, code review, and detected refactorings in pull requests; and

- identifying the differences between refactoring-inducing pull requests and non-refactoring-inducing ones (by considering quantitative data).

Those data were mined from 134 distinct Apache’s repositories in GitHub, totaling 4,525 merged pull requests, and 48,735 review comments, besides 90,583 detected refactorings by RefactoringMiner 2.0.

In short, we obtained motivating quantitative results. By investigating our sample, we found 46.4% of refactoring-inducing pull requests, comprising refactoring edits from 39 distinct types that gradually decrease over the commits, and operate in high, medium, and low levels. We discovered that refactoring-inducing pull requests significantly differ from non-refactoring-inducing ones in terms of code churn, number of changed files, number of commits, number of review comments, length of discussion, and time to merge. Particularly, we found no statistical evidence that the number of reviewers is related to refactoring-inducement.

Based on those results, we have suggested a further investigation of review comments, by using the content analysis approach, intending to identify patterns that could indicate the refactoring-inducement as a contribution of the code review process to the code submitted within pull requests. In practice, we conjecture that this research impacts researchers, practitioners, and tool builders regarding a novel view on pull requests, when recognizing that they may be refactoring-inducing.

Bibliography

- [1] Eclipse code review on Gerrit. <https://git.eclipse.org/r/>. Accessed on: June 2020.
- [2] Gerrit code review system. <https://www.gerritcodereview.com>. Accessed on: June 2020.
- [3] Git version control system. <https://git-scm.com/>. Accessed on: June 2020.
- [4] GitHub developer guide GraphQL API v4. <https://developer.github.com/v4/>. Accessed on: June 2020.
- [5] GitHub developer guide REST API v3. <https://developer.github.com/v3/>. Accessed on: June 2020.
- [6] GitHub platform. <https://github.com>. Accessed on: June 2020.
- [7] GitHub pull requests. <https://help.github.com/en/github/collaborating-with-issues-and-pull-requests>. Accessed on: June 2020.
- [8] Phabricator code review system. <https://www.phacility.com/phabricator/>. Accessed on: June 2020.
- [9] RefactoringMiner – a refactoring detection tool. <https://github.com/tsantalis/RefactoringMiner>. Accessed on: September 2019.
- [10] Review Board code review system. <https://www.reviewboard.org/>. Accessed on: June 2020.

-
- [11] Synopsys repositories comparison – report. <https://www.openhub.net/repositories/compare>. Accessed on: June 2020.
 - [12] Manifesto for agile software development. <https://agilemanifesto.org/>, February 2001. Accessed on: August 2020.
 - [13] The Apache® Software Foundation expands infrastructure with GitHub integration. <https://blogs.apache.org/foundation/entry/the-apache-software-foundation-expands>, April 2019. Accessed on: June 2020.
 - [14] Briefing: The Apache way. <https://www.apache.org/theapacheway/>, 2019. Accessed on: June 2020.
 - [15] Preliminary investigations on refactorings and modern code review. <https://git.io/JTLum>, July 2019.
 - [16] The state of the Octoverse – GitHub 2019 report. <https://octoverse.github.com/>, September 2019. Accessed on: July 2020.
 - [17] The Apache® Software Foundation projects statistics. <https://projects.apache.org/statistics.html>, November 2020. Accessed on: December 2020.
 - [18] Characterizing refactoring-inducing pull requests. <https://git.io/JTbgN>, December 2020.
 - [19] Stack Overflow annual developer survey. <https://insights.stackoverflow.com/survey/2020#most-popular-technologies>, 2020. Accessed on: December 2020.
 - [20] Charu C. Aggarwal and Chandan K. Reddy. *Data Clustering: Algorithms and Applications*. Chapman Hall/CRC, first edition, 2013.
 - [21] Rakesh Agrawal, Tomasz Imieliński, and Arun Swami. Mining association rules between sets of items in large databases. *ACM SIGMOD Record*, 22(2):207–216, June 1993.

- [22] Eman A. AlOmar, Mohamed W. Mkaouer, and Ali Ouni. Can refactoring be self-affirmed? An exploratory study on how developers document their refactoring activities in commit messages. In *Proceedings of the 3rd International Workshop on Refactoring*, IWor'19, pages 51–58, Montreal, Canada, May 2019. IEEE Press.
- [23] Everton L. G. Alves, Myoungkyu Song, Tiago Massoni, Patricia D. L. Machado, and Miryung Kim. Refactoring inspection support for manual refactoring edits. *IEEE Transactions on Software Engineering*, 44(4):365–383, 2018.
- [24] Mihael Ankerst, Markus M. Breunig, Hans-Peter Kriegel, and Jörg Sander. OP-TICS: Ordering points to identify the clustering structure. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, SIGMOD'99, pages 49–60, Philadelphia, USA, 1999. ACM.
- [25] Howard Anton and Chris Rorres. *Elementary Linear Algebra: Applications Version*. Wiley publishing, eleventh edition, 2014.
- [26] Alberto Bacchelli and Christian Bird. Expectations, outcomes, and challenges of modern code review. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE'13, pages 712–721, San Francisco, USA, 2013. IEEE Press.
- [27] Mike Barnett, Christian Bird, João Brunet, and Shuvendu K. Lahiri. Helping developers help themselves: Automatic decomposition of code review change-sets. In *Proceedings of 37th International Conference on Software Engineering - Volume 1*, ICSE'15, pages 134–144, Florence, Italy, 2015. IEEE Press.
- [28] Tobias Baum and Kurt Schneider. On the need for a new generation of code review tools. In *Product-Focused Software Process Improvement*, pages 301–308. Springer International Publishing, 2016.
- [29] Olga Baysal, Oleksii Kononenko, Reid Holmes, and Michael W. Godfrey. Investigating technical and non-technical factors influencing modern code review. *Empirical Software Engineering*, 21(3):932–959, June 2016.

- [30] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Longman Publishing Co., Inc., USA, 1999.
- [31] Moritz Beller, Alberto Bacchelli, Andy Zaidman, and Elmar Juergens. Modern code reviews in open-source projects: Which problems do they fix? In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR'14, pages 202–ss211, Hyderabad, India, 2014. ACM.
- [32] Fernando Berzal, Ignacio Blanco, Daniel Sánchez, and María-Amparo Vila. Measuring the accuracy and interest of association rules: A new framework. *Intelligent Data Analysis*, 6(3):221–235, August 2002.
- [33] Giuseppe Bonaccorso. *Machine Learning Algorithms*. Packt Publishing, first edition, July 2017.
- [34] Amiangshu Bosu, Jeffrey C. Carver, Christian Bird, Jonathan Orbeck, and Christopher Chockley. Process aspects and social dynamics of contemporary code review: Insights from open source development and industrial practice at microsoft. *IEEE Trans. Softw. Eng.*, 43(1):56–75, January 2017.
- [35] Amiangshu Bosu, Michaela Greiler, and Christian Bird. Characteristics of useful code reviews: An empirical study at Microsoft. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, MSR'15, pages 146–156, Florence, Italy, 2015. IEEE Press.
- [36] Sergey Brin, Rajeev Motwani, Jeffrey D. Ullman, and Shalom Tsur. Dynamic itemset counting and implication rules for market basket data. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, SIGMOD'97, pages 255–264, New York, NY, USA, 1997. ACM.
- [37] Aline Brito, Andre Hora, and Marco T. Valente. Refactoring graphs: Assessing refactoring over time. In *Proceedings of IEEE 27th International Conference on Software Analysis, Evolution and Reengineering*, SANER'20, pages 367–377, 2020.
- [38] Neil Burdess. *Starting statistics: a short, clear guide*. SAGE Los Angeles, 2010.

- [39] M. Emre Celebi and Kemal Aydin. *Unsupervised Learning Algorithms*. Springer Publishing Company, Incorporated, first edition, 2016.
- [40] Scott Chacon and Ben Straub. *Pro Git*. Apress, USA, second edition, 2014.
- [41] Zhiyuan Chen, Young-Woo Kwon, and Myoungkyu Song. Clone refactoring inspection by summarizing clone refactorings and detecting inconsistent changes during software evolution. *Journal of Software: Evolution and Process*, 30(10):e1951, 2018.
- [42] Flavia Coelho, Tiago Massoni, and Everton L. G. Alves. Refactoring-aware code review: A systematic mapping study. In *Proceedings of the 3rd International Workshop on Refactoring, IWoR’19*, pages 63–66, Montreal, Canada, 2019. IEEE Press.
- [43] Frans Coenen, Graham Goulbourne, and Paul Leng. Tree structures for mining association rules. *Data Mining and Knowledge Discovery*, 8(1):25–51, January 2004.
- [44] Jacob Cohen. A coefficient of agreement for nominal scales. *Educational and Psychological Measurement*, 20(1):37–46, 1960.
- [45] L. Cohen, L. Manion, and K. Morrison. Research methods in education. *British Journal of Educational Studies*, 48(4):446–446, 2000.
- [46] Brett Daniel, Danny Dig, Kely Garcia, and Darko Marinov. Automated testing of refactoring engines. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering, ESEC/FSE’07*, pages 185–194, Dubrovnik, Croatia, 2007. ACM.
- [47] Marco di Biase, Magiel Bruntink, Arie van Deursen, and Alberto Bacchelli. The effects of change-decomposition on code review – A controlled experiment. *PeerJ Computer Science*, 5:e193, May 2018.

- [48] Bradley Efron and Robert J Tibshirani. *An introduction to the bootstrap*. Monographs on Statistics and Applied Probability. Chapman and Hall, London, England, 1993.
- [49] Vasiliki Efstathiou and Diomidis Spinellis. Code review comments: Language matters. In *Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results*, ICSE-NIER'18, pages 69–72, Gothenburg, Sweden, 2018. ACM.
- [50] Michael E. Fagan. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 15(3):182–211, 1976.
- [51] John W. Foreman. *Data Smart: Using Data Science to Transform Information into Insight*. Wiley Publishing, first edition, 2013.
- [52] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., USA, 1999.
- [53] Martin Fowler. *Refactoring: Improving the Design of Existing Code (second edition)*. Addison-Wesley Professional, USA, 2018.
- [54] Johannes Fürnkranz and Tomáš Kliegr. A brief overview of rule learning. In *Rule Technologies: Foundations, Tools, and Applications*, pages 54–69. Springer International Publishing, 2015.
- [55] Tate Galbraith. Which code merging method should I use in GitHub? <https://t.ly/BDL9>, May 2020. Accessed on June 2020.
- [56] Xi Ge, Saurabh Sarkar, Jim Witschey, and Emerson Murphy-Hill. Refactoring-aware code review. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing*, VL/HCC'17, pages 71–79, Raleigh, USA, 2017. IEEE Press.
- [57] Liqiang Geng and Howard J. Hamilton. Interestingness measures for data mining: A survey. *ACM Computing Surveys*, 38(3):9–es, September 2006.

-
- [58] Bart Goethals. *Frequent Set Mining*, pages 321–338. Springer US, Boston, USA, 2010.
 - [59] Georgios Gousios, Martin Pinzger, and Arie van Deursen. An exploratory study of the pull-based software development model. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE’14, pages 345–355, Hyderabad, India, 2014. ACM.
 - [60] Georgios Gousios, Margaret-Anne Storey, and Alberto Bacchelli. Work practices and challenges in pull-based development: The contributor’s perspective. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE’16, pages 285–296, Austin, USA, 2016. ACM.
 - [61] Bo Guo and Myoungkyu Song. Interactively decomposing composite changes to support code review and regression testing. In *Proceedings of the IEEE 41st Annual Computer Software and Applications Conference*, COMPSAC’17, pages 118–127, Turin, Italy, 2017. IEEE Press.
 - [62] Jiawei Han, Jian Pei, and Yiwen Yin. Mining frequent patterns without candidate generation. *SIGMOD Record*, 29(2):1–12, May 2000.
 - [63] Péter Hegedüs, István Kádár, Rudolf Ferenc, and Tibor Gyimóthy. Empirical evaluation of software maintainability based on a manually validated refactoring dataset. *Information and Software Technology*, 95:313–327, 2018.
 - [64] Jez Humble and David Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley Professional, first edition, 2010.
 - [65] Daniel Izquierdo-Cortazar, Lars Kurth, Jesus M. Gonzalez-Barahona, Santiago Dueñas, and Nelson Sekitoleko. Characterization of the xen project code review process: An experience report. In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR’16, pages 386–390, Austin, USA, 2016. ACM.

-
- [66] Anil K. Jain, Narasimha M. Murty, and Patrick J. Flynn. Data clustering: A review. *ACM Computing Surveys*, 31(3):264–323, September 1999.
 - [67] Yujuan Jiang, Bram Adams, and Daniel M. German. Will my patch make it? And how fast?: Case study on the linux kernel. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR’13, pages 101–110. IEEE Press, 2013.
 - [68] Philip M. Johnson. Reengineering inspection. *Communications of ACM*, 41(2):49–52, February 1998.
 - [69] Prateek Joshi. *Artificial Intelligence with Python*. Packt Publishing, first edition, January 2017.
 - [70] Yoshio Kataoka, Takeo Imai, Hiroki Andou, and Tetsuji Fukaya. A quantitative evaluation of maintainability enhancement by refactoring. In *Proceedings of the International Conference on Software Maintenance*, ICSM’02, pages 576–585, USA, 2002. IEEE Computer Society.
 - [71] Miryung Kim, Thomas Zimmermann, and Nachiappan Nagappan. An empirical study of refactoring challenges and benefits at Microsoft. *IEEE Transactions on Software Engineering*, 40(7):633–649, July 2014.
 - [72] Oleksii Kononenko, Olga Baysal, and Michael W. Godfrey. Code review quality: How developers see it. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE’16, pages 1028–1038, Austin, EUA, 2016. ACM.
 - [73] Oleksii Kononenko, Tresa Rose, Olga Baysal, Michael Godfrey, Dennis Theisen, and Bart de Water. Studying pull request merges: A case study of Shopify’s active merchant. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*, ICSE-SEIP’18, pages 124–133, Gothenburg, Sweden, 2018. ACM.
 - [74] Eugene F Krause. *Taxicab geometry: an adventure in non-Euclidean geometry*. Dover Publications, Inc., New York, 1987.

-
- [75] Klaus Krippendorff. *Content Analysis: An Introduction to Its Methodology (second edition)*. Sage Publications, 2004.
 - [76] Stephen Lacy, Brendan R. Watson, Daniel Riffe, and Jennette Lovejoy. Issues and best practices in content analysis. *Journalism & Mass Communication Quarterly*, 92(4):791–811, 2015.
 - [77] J. Richard Landis and Gary G. Koch. The measurement of observer agreement for categorical data. *Biometrics*, 33(1):159–174, 1977.
 - [78] Robert Layton. *Learning Data Mining with Python*. Packt Publishing, first edition, July 2015.
 - [79] Gwendolyn K. Lee and Robert E. Cole. From a firm-based to a community-based model of knowledge creation: The case of the linux kernel development. *Organization Science*, 14(6):633–649, 2003.
 - [80] Zhi-Xing Li, Yue Yu, Gang Yin, Tao Wang, and Huai-Min Wang. What are they talking about? analyzing code reviews in pull-based development model. *Journal of Computer Science and Technology*, 32(6):1060–1075, November 2017.
 - [81] Hiu Liu, Zhiyi Ma, Weizhong Shao, and Zhendong Niu. Schedule of bad smell detection and resolution: A new way to save effort. *IEEE Transactions on Software Engineering*, 38(1):220–235, 2012.
 - [82] Laura MacLeod, Michaela Greiler, Margaret-Anne Storey, Christian Bird, and Jacek Czerwinka. Code reviewing in the trenches: Challenges and best practices. *IEEE Software*, 35(04):34–42, July 2018.
 - [83] Mehran Mahmoudi, Sarah Nadi, and Nikolaos Tsantalis. Are refactorings to blame? an empirical study of refactorings in merge conflicts. In *Proceedings of the IEEE 26th International Conference on Software Analysis, Evolution and Reengineering*, SANER’19, pages 151–162, Hangzhou, China, 2019. IEEE Press.
 - [84] Matheus Paixão and Anderson Uchôa and Ana Bibiano and Daniel Oliveira and Alessandro Garcia and Jens Krinke and Emilio Arvonio. Behind the Intents:

- An In-depth Empirical Study on Software Refactoring in Modern Code Review (MSR 2020 - Technical Papers), 2020.
- [85] Jumpei Matsuda, Shinpei Hayashi, and Motoshi Saeki. Hierarchical categorization of edit operations for separately committing large refactoring results. In *Proceedings of the 14th International Workshop on Principles of Software Evolution*, IWPSE'15, pages 19–27, Bergamo, Italy, 2015. ACM.
- [86] Kenneth O. McGraw and Seok P. Wong. A common language effect size statistic. *Psychological Bulletin*, 111(2):361–365, 1992.
- [87] Shane McIntosh, Yasutaka Kamei, Bram Adams, and Ahmed E. Hassan. The impact of code review coverage and code review participation on software quality: A case study of the Qt, VTK, and ITK projects. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR'14, pages 192–201, Hyderabad, India, 2014. ACM.
- [88] Davoud Moulavi, Pablo A. Jaskowiak, Ricardo J. G. B. Campello, Arthur Zimek, and Jörg Sander. Density-based clustering validation. In *Proceedings of the International Conference in Data Mining*, pages 839–847, Philadelphia, USA, 2014. SIAM.
- [89] Emerson Murphy-Hill, Chris Parnin, and Andrew P. Black. How we refactor, and how we know it. *IEEE Transactions on Software Engineering*, 38(1):5–18, January 2012.
- [90] Stas Negara, Nicholas Chen, Mohsen Vakilian, Ralph E. Johnson, and Danny Dig. A comparative study of manual and automated refactorings. In *Proceedings of the 27th European Conference on Object-Oriented Programming*, ECOOP'13, pages 552–576, Montpellier, France, 2013. Springer-Verlag.
- [91] U.S. Department of Commerce, National Institute of Standards, and Technology. *Secure Hash Standard - SHS: Federal Information Processing Standards Publication 180-4*. CreateSpace Independent Publishing Platform, North Charleston, USA, 2012.

- [92] William F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 1992.
- [93] William F. Opdyke and Ralph E. Johnson. Refactoring: An aid in designing application frameworks and evolving object-oriented systems. In *Proceedings of the Symposium on Object Oriented Programming Emphasizing Practical Applications*, New York, USA, September 1990.
- [94] Ali Ouni, Marouane Kessentini, Houari Sahraoui, Katsuro Inoue, and Kalyanmoy Deb. Multi-criteria code refactoring using search-based software engineering: An industrial case study. *ACM Transactions on Software Engineering Methodology*, 25(3), June 2016.
- [95] Matheus Paixão, Jens Krinke, DongGyun Han, Chaoyong Ragkhitwetsagul, and Mark Harman. The impact of code review on architectural changes. *IEEE Transactions on Software Engineering*, pages 1–1, 2019.
- [96] Fabio Palomba, Andy Zaidman, Rocco Oliveto, and Andrea De Lucia. An exploratory study on the relationship between changes and refactoring. In *Proceedings of the 25th International Conference on Program Comprehension, ICPC’17*, pages 176–185, Buenos Aires, Argentina, 2017. IEEE Press.
- [97] Thai Pangsakulyanont, Patanamon Thongtanunam, Daniel Port, and Hajimu Iida. Assessing mcr discussion usefulness using semantic similarity. In *Proceedings of the 6th International Workshop on Empirical Software Engineering in Practice, IWESEP’14*, pages 49–54, Osaka, Japan, 2014. IEEE Press.
- [98] Jevgenija Pantiuchina, Fiorella Zampetti, Simone Scalabrino, Valentina Piantadosi, Rocco Oliveto, Gabriele Bavota, and Massimiliano Di Penta. Why developers refactor source code: A mining-based study. *ACM Transactions on Software Engineering Methodology*, 29(4), September 2020.
- [99] Luca Pascarella, Davide Spadini, Fabio Palomba, Magiel Bruntink, and Alberto Bacchelli. Information needs in contemporary code review. *Proceedings of the ACM on Human-Computer Interaction*, 2(CSCW), November 2018.

-
- [100] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *Journal of Machine Learning Research*, 12(Oct):2825–2830, 2011.
 - [101] Kyle Prete, Napol Rachatasumrit, Nikita Sudan, and Miryung Kim. Template-based reconstruction of complex refactorings. In *Proceedings of the 2010 IEEE International Conference on Software Maintenance*, ICSM’10, pages 1–10, Timisoara, Romania, 2010. IEEE Computer Society.
 - [102] Mohammad Masudur Rahman, Chanchal K. Roy, and Raula G. Kula. Predicting usefulness of code review comments using textual features and developer experience. In *Proceedings of the 14th International Conference on Mining Software Repositories*, MSR’17, pages 215–226, Buenos Aires, Argentina, 2017. IEEE Press.
 - [103] Achyudh Ram, Anand Ashok Sawant, Marco Castelluccio, and Alberto Bacchelli. What makes a code change easier to review: An empirical investigation on code change reviewability. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE’18, pages 201–212, Lake Buena Vista, USA, 2018. ACM.
 - [104] Sebastian Raschka. Mlxtend: Providing machine learning and data science utilities and extensions to python’s scientific computing stack. *Journal of Open Source Software*, 3(24):638, 2018.
 - [105] Hefin I. Rhys. *Machine Learning with R, the tidyverse, and mlr*. Manning Publications, first edition, 2020.
 - [106] Peter Rigby, Brendan Cleary, Frederic Painchaud, Margaret-Anne Storey, and Daniel German. Contemporary peer review in action: Lessons from open source development. *IEEE Software*, 29(6):56–61, November 2012.

-
- [107] Peter C. Rigby and Christian Bird. Convergent contemporary software peer review practices. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE'13, pages 202–212, Saint Petersburg, Russia, 2013. ACM.
 - [108] Peter C. Rigby, Daniel M. German, Laura Cowen, and Margaret-Anne Storey. Peer review on open-source software projects: Parameters, statistical models, and theory. *ACM Transactions on Software Engineering Methodology*, 23(4), September 2014.
 - [109] Peter C. Rigby, Daniel M. German, and Margaret-Anne Storey. Open source software peer review practices: A case study of the apache server. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE'08, pages 541–550, Leipzig, Germany, 2008. ACM.
 - [110] Romain Robbes and Michele Lanza. Characterizing and understanding development sessions. In *Proceedings of the 15th IEEE International Conference on Program Comprehension*, ICPC'07, pages 155–166, USA, 2007. IEEE Computer Society.
 - [111] John T. Roscoe. *Fundamental Research Statistics for the behavioral sciences*. International Series in decision processes. Holt, Rinehart Winston, 2. ed edition, 1975.
 - [112] Per Runeson and Martin Höst. Guidelines for conducting and reporting case study research in Software Engineering. *Empirical Software Engineering*, 14(2):131–164, April 2009.
 - [113] Per Runeson, Martin Höst, Austen Rainer, and Bjorn Regnell. *Case Study Research in Software Engineering: Guidelines and Examples*. Wiley Publishing, first edition, 2012.
 - [114] Caitlin Sadowski, Emma Söderberg, Luke Church, Michal Sipko, and Alberto Bacchelli. Modern code review: A case study at google. In *Proceedings of the*

- 40th International Conference on Software Engineering: Software Engineering in Practice*, ICSE-SEIP'18, pages 181–190, Gothenburg, Sweden, 2018. ACM.
- [115] Chris Sauer, D. Ross Jeffery, Lesley Land, and Philip Yetton. The effectiveness of software development technical reviews: A behaviorally motivated program of research. *IEEE Transactions on Software Engineering*, 26(1):1–14, January 2000.
- [116] Danilo Silva, Nikolaos Tsantalis, and Marco Tulio Valente. Why we refactor? confessions of GitHub contributors. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE'16, pages 858–870. ACM, 2016.
- [117] Danilo Silva and Marco Tulio Valente. Refdiff: Detecting refactorings in version histories. In *Proceedings of the 14th International Conference on Mining Software Repositories*, MSR'17, pages 269–279, Buenos Aires, Argentina, 2017. IEEE Press.
- [118] Gustavo Soares, Rohit Gheyi, and Tiago Massoni. Automated behavioral testing of refactoring engines. *IEEE Transactions on Software Engineering*, 39(2):147–162, February 2013.
- [119] Ian Sommerville. *Software Engineering*. Pearson, tenth edition, 2015.
- [120] Ramakrishnan Srikant and Jeffrey F. Naughton. *Fast Algorithms for Mining Association Rules and Sequential Patterns*. PhD thesis, 1996. AAI9708697.
- [121] Gábor Szőke, Csaba Nagy, Rudolf Ferenc, and Tibor Gyimóthy. A case study of refactoring large-scale industrial systems to efficiently improve source code quality. In *Proceedings of the International Conference on Computational Science and its Applications*, ICCSA'14, pages 524–540, Guimarães, Portugal, 2014. Springer International Publishing.
- [122] Pang-Ning Tan, Michael Steinbach, Anuj Karpatne, and Vipin Kumar. *Introduction to Data Mining*. Pearson, second edition, 2018.

- [123] Yida Tao, Yingnong Dang, Tao Xie, Dongmei Zhang, and Sunghun 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*, FSE'12, pages 1–11, Cary, USA, 2012. ACM.
- [124] Yida Tao and Sunghun Kim. Partitioning composite code changes to facilitate code review. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, MSR'15, pages 180–190, Florence, Italy, 2015. IEEE Press.
- [125] Cédric Teyton, Jean-Rémy Falleri, Marc Palyart, and Xavier Blanc. A study of library migrations in java. *Journal of Software: Evolution and Process*, 26(11):1030–1052, November 2014.
- [126] Patanamon Thongtanunam, Shane McIntosh, Ahmed E. Hassan, and Hajimu Iida. Review participation in modern code review. *Empirical Software Engineering*, 22(2):768–817, April 2017.
- [127] Nikolaos Tsantalis, Ameya Ketkar, and Danny Dig. RefactoringMiner 2.0. *IEEE Transactions on Software Engineering*, 2020.
- [128] Nikolaos Tsantalis, Matin Mansouri, Laleh M. Eshkevari, Davood Mazinianian, and Danny Dig. Accurate and efficient refactoring detection in commit history. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE'18, pages 483–494, Gothenburg, Sweden, 2018. ACM.
- [129] Jason Tsay, Laura Dabbish, and James Herbsleb. Let's talk about it: Evaluating contributions through discussion in GitHub. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE'14, pages 144–154, Hong Kong, China, 2014. ACM.
- [130] Carmine Vassallo, Giovanni Grano, Fabio Palomba, Harald C. Gall, and Alberto Bacchelli. A large-scale empirical exploration on refactoring activities in open source software projects. *Science of Computer Programming*, 180:1–15, 2019.

-
- [131] Dongkuan Xu and Yingjie Tian. A comprehensive survey of clustering algorithms. *Annals of Data Science*, 2:165–193, August 2015.
 - [132] Yue Yu, Huaimin Wang, Vladimir Filkov, Premkumar Devanbu, and Bogdan Vasilescu. Wait for it: Determinants of pull request evaluation latency on GitHub. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, MSR’15, pages 367–371, Florence, Italy, 2015. IEEE Press.
 - [133] Yan Zhang and Barbara M. Wildemuth. Qualitative analysis of content by. *Human Brain Mapping*, 30(7):2197–2206, 2005.
 - [134] Alice Zheng and Amanda Casari. *Feature Engineering for Machine Learning: Principles and Techniques for Data Scientists*. O’Reilly Media, Inc., first edition, 2018.
 - [135] Hao Zhong, Ye Yang, and Jacky Keung. Assessing the representativeness of open source projects in empirical software engineering studies. In *Proceedings of the 19th Asia-Pacific Software Engineering Conference*, APSEC’12, pages 808–817, Hong Kong, China, 2012. IEEE Press.
 - [136] Jiaxin Zhu, Minghui Zhou, and Audris Mockus. Effectiveness of code contribution: From patch-based to pull-request-based tools. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE’16, pages 871–882, Seattle, USA, 2016. ACM.

Appendix A

Initial Investigations on Refactoring and Modern Code Review

In an effort to explore how refactorings impact code review, we carried out the following phases:

1. (Q2 2018 – Q3 2018) a replication of the empirical study, developed by Tsantalis and colleagues [128], on refactoring detection,
2. (Q3 2018) a study about multiple refactorings in a single commit,
3. (Q4 2018) a systematic literature mapping about refactoring-aware code review; presented as a position paper at the International Workshop on Software Refactoring 2019 [42]¹,
4. (Q1 2019) a pre-test of a quasi-experiment concerning refactoring-aware code review with members of the e-Pol, a Brazilian Federal Police's system, developed at SPLab/UFCG to support the process and access to information coming from investigations,
5. (Q2 2019) a brief "case history" about refactoring-aware code review on Gerrit [2],

¹The details and the reproduction kit of this systematic literature mapping study, conducted from September 2018 to January 2019, are available in <https://github.com/flaviacoelho/racr-sysmap>

6. (Q3 2019) a brief "case history" regarding refactoring-inducing code review in open pull requests on GitHub [6], and
7. (Q4 2019) a brief "case history" relating to refactoring-inducing code review in merged pull requests on GitHub [6].

Table A.1 succinctly provides the rationale, result, and respective impact of each phase fulfilled towards the thesis proposal. In order to obtain more details, consult [15].

It should be noted that phases 1, 2, and 3 were essential for understanding the fundamentals of refactorings and code review, selecting an accurate refactoring detection tool, and identifying research opportunities on refactoring-aware code review.

The results of phase 4 supported the decision to expand the research in terms of the number of reviewers, software complexity, and research method (from a quasi-experiment to a case study), aiming more relevant findings. For that, we selected the Gerrit code review system, because it is an open-source web-based tool, which provides repository management for the Git version control system [3], and it is utilized by large scale projects, such as Eclipse [1].

Despite the motivating results from phase 5, at that point, it was noticeable the growth of GitHub in terms of engaged organizations [16], such as the Apache Software Foundation, which has completely migrated its projects to GitHub, in February 2019 [13].

As a result, we conducted phases 6 and 7 to understand the GitHub pull-based development model and explore possibilities for the research design. Concurrently, a regular literature review and technical meetings with the participation of the professor Nikolaos Tsantalis (Concordia University, Canada) promoted a change in the Thesis proposal topic towards refactoring-inducing pull requests.

<i>Phase</i>	<i>Rationale</i>	<i>Result</i>	<i>Impact for the research</i>
1	Find an accurate refactoring detection tool for source code written in the Java language	RefactoringMiner [9]	Knowledge on the state-of-the-art in refactoring detection
2	Check the RefactoringMiner accuracy in detection of multiple refactorings in a single commit	Affirmative	Selection of the RefactoringMiner for the purpose of refactoring detection
3	Search for refactoring-aware solutions to support MCR	Systematic mapping	A few potential research directions
4	Verify the feasibility of an experiment about refactoring-aware code review in a midsize project	Infeasible	Selection of Gerrit for an in-depth exploration
5	Investigate a case study feasibility from Gerrit review data	Brief "case history"	Motivating results
6	Investigate a case study feasibility from GitHub open pull requests	Brief "case history"	Motivating results towards a case study on merged pull requests
7	Investigate a case study feasibility from GitHub merged pull requests	Brief "case history"	Decision for performing a case study on data from GitHub merged pull requests

Table A.1: A summary of the preliminary investigation results

Appendix B

Strategy for Mining Squashed and Merged Pull Requests from GitHub

In order to mine a pull request merged by *squash and merge* option, it is needed to recover its original commits. To clarify, consider the Apache’s pull request #1807¹, as illustrated in Figure B.1. Originally, this pull request had 12 commits (c_1 to c_{12}) that were squashed after a *force-pushed* event. Consequently, now, only one commit may be gathered from the pull request ($c_{afterCommit}$).

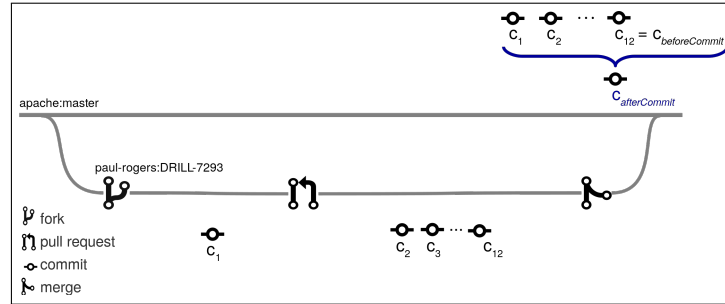


Figure B.1: An overview of the pull request #1807 events in GitHub

To deal with this scenario, we implemented the following two-steps strategy:

1. since the *HeadRefForcePushedEvent* is an object accessible through GitHub GraphQL API v4 [4], we recovered the after and before commits of the pull request #1807 (Table B.1); and

¹<https://github.com/apache/drill/pull/1807>

2. we rebuilt the original commits' history (Table B.2), through tracking the commits from $c_{beforeCommit}$, which has the same value of c_{12} , until reaching the same SHA of the $c_{afterCommit}$'s parent (77cf7e2ee61 fb40e7efd85148ac76947d13dda38), through the *compare* operation, accessible via GitHub REST API v3 [5].

Table B.1: GraphQL API's HeadRefForcePushedEvent fields for pull request #1807

$c_{afterCommit}$	ed9b9f2fe279bbafec1680b1229ecbaa40b3ed23
$c_{beforeCommit}$	b1a5446174485c55a162771b1e804a0587eef361

Table B.2: The recovered history of the pull request #1807's commits

Commit index	PR's original commits
c_1	21fc7b6d3e6064ff2c28bb1b9920487e7cf995ba
c_2	2041aca8887882b6f33a1a4366f44b5f2dac681c
c_3	0d521265e79ac05b33480dd3adb2a078ca28e54b
c_4	02fb0e9945353e187f5eaa8bea6a5763f3f9b9fb
c_5	ab512953b0e097b02fb25e33529bba0e27651fb7
c_6	a07936715378f41d7e375be53218d3dfc0a8e45e
c_7	6320a77caca59b886a50c5ef47cbb1d6461d98fb
c_8	6a712ddf7fe4693594805f64692c2677239fbe08
c_9	a14ebf31d6c924cbe877e1f1f672e211e3207e89
c_{10}	e816deba8dfbd937f89c216cc8c74aa6adf01aed
c_{11}	539bd0edd8348d03df6d17ae4ff2387c10dd10e9
c_{12}	b1a5446174485c55a162771b1e804a0587eef361

We executed strategy's Step 1 for gathering the after and before commits from 65,006 pull requests, obtaining 16,668 squashed commits². After collecting all $c_{afterCommit}$'s parent, we realized that some commits have two parents, i.e., they are the result of merging one branch to another by a commit merge that has both of them as its parents, a merging known as *true merge* [40]. Therefore, 112 commits were not considered squashed commits. We recovered 53,915 commits running strategy's Step 2.

²The complete output is available in <https://github.com/flaviacoelho/merged-prs-miner>