Universidade Federal de Campina Grande

Centro de Engenharia Elétrica e Informática

Coordenação de Pós-Graduação em Ciência da Computação

Flávia Estélia Silva Coelho

Characterizing Refactoring-Inducing Pull Requests

Campina Grande, Paraíba, Brasil

2022

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 Estélia Silva Coelho

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

C672c Coelho, Flávia Estélia Silva.

Characterizing refactoring-inducing pull requests / Flávia Estélia Silva Coelho. – Campina Grande, 2022.

210 f. : il. color.

Tese (Doutorado em Ciência da Computação) – Universidade Federal de Campina Grande, Centro de Engenharia Elétrica e Informática, 2022.

"Orientação: Prof. Dr. Tiago Lima Massoni, Prof. Dr. Everton Leandro Galdino Alves".

Referências.

1. Refactoring-Inducing Pull Request. 2. Code Review. 3. Empirical Study. 4. Computing Methodology and Techniques. 5. Software Engineering. I. Massoni, Tiago Lima. II. Alves, Everton Leandro Galdino. III. Título.

CDU 004.41(043)

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECÁRIA SEVERINA SUELI DA SILVA OLIVEIRA CRB-15/225

FOLHA DE ASSINATURA PARA TESES E DISSERTAÇÕES

FLÁVIA ESTÉLIA SILVA COELHO

CHARACTERIZING REFACTORING-INDUCING PULL REQUESTS

Tese apresentada ao Programa de Pós-Graduação em Ciência da Computação como pré-requisito para obtenção do título de Doutor em Ciência da Computação.

Aprovada em: 18/02/2022

Prof. Dr. TIAGO LIMA MASSONI, Orientador, UFCG

Prof. Dr. EVERTON LEANDRO GALDINO ALVES, Orientador, UFCG

Prof. Dr. ROHIT GHEYI, Examinador Interno, UFCG

Profa. Dra. MELINA MONGIOVI BRITO LIRA , Examinadora Interna, UFCG

Prof. Dr. UIRA KULESZA, Examinador Externo, UFRN

Prof. Dr. ALESSANDRO FABRICIO GARCIA, Examinador Externo, PUC-RIO conforme horário oficial de Brasília, com fundamento no art. 8º, caput, da Portaria SEI nº 002, de 25 de outubro de 2018.

Documento assinado eletronicamente por **MELINA MONGIOVI CUNHA LIMA SABINO**, **COORDENADOR DE POS-GRADUACAO**, em 18/02/2022, às 15:37, conforme horário oficial de Brasília, com fundamento no art. 8º, caput, da Portaria SEI nº 002, de 25 de outubro de 2018.

Documento assinado eletronicamente por **ROHIT GHEYI**, **PROFESSOR DO MAGISTERIO SUPERIOR**, em 18/02/2022, às 16:16, conforme horário oficial de Brasília, com fundamento no art. 8º, caput, da Portaria SEI nº 002, de 25 de outubro de 2018.

Documento assinado eletronicamente por **Uirá Kulesza**, **Usuário Externo**, em 18/02/2022, às 16:50, conforme horário oficial de Brasília, com fundamento no art. 8º, caput, da Portaria SEI nº 002, de 25 de outubro de 2018.

Documento assinado eletronicamente por **Alessandro Fabricio Garcia**, **Usuário Externo**, em 18/02/2022, às 17:27, conforme horário oficial de Brasília, com fundamento no art. 8º, caput, da Portaria SEI nº 002, de 25 de outubro de 2018.

A autenticidade deste documento pode ser conferida no site https://sei.ufcg.edu.br/autenticidade, informando o código verificador **2125607** e o código CRC **8CA353D6**.

Referência: Processo nº 23096.005705/2022-11 SEI nº 2125607

Resumo

O desenvolvimento baseado em pull adequa-se à prática da Revisão de Código Moderna (RCM), na qual os revisores podem sugerir melhorias de código, como refatoramentos, por meio de comentários e commits em Pull Requests (PRs). Estudos anteriores de RCM tratam todos os PRs como semelhantes, independentemente de induzirem refatoramento ou não. Definimos um PR como indutor de refatoramento quando as edições de refatoramento são realizadas após o(s) commit(s) inicial(is) como resultado de comentários de revisores ou ações espontâneas realizadas pelo autor do PR. Este trabalho explora aspectos relacionados à revisão de código com o objetivo de caracterizar PRs indutores de refatoramento. Para isso, extraímos edições de refatoramento e dados de revisão de código do GitHub. Em seguida, realizamos estudos empíricos para identificar similaridades/dissimilaridades entre PRs indutores de refatoramento e não indutores de refatoramento e caracterizar revisão de código e edições de refatoramento em PRs indutores de refatoramento. Encontramos diferenças significativas entre PRs indutores de refatoramento e não indutores de refatoramento e evidências empíricas sobre a relevância da revisão de código para edições de refatoramento no nível de PR. Observamos fatores motivadores por trás dos PRs indutores de refatoramento, identificamos aspectos estruturais dos comentários de revisão em PRs indutores e não indutores de refatoramento e propomos diretrizes para uma revisão de código mais produtiva. Também encontramos evidências empíricas sobre aspectos técnicos que caracterizam refatoramentos em PRs indutores de refatoramento. Nossas descobertas sugerem orientações para pesquisadores, profissionais e desenvolvedores de ferramentas para melhorar as práticas em torno da revisão de código baseada em pull.

Palavras-chaves: Solicitação Pull Indutora de Refatoramento; Revisão de Código; Estudo Empírico; Metodologia e Técnicas da Computação; Engenharia de Software.

Abstract

Pull-based development has shaped the practice of Modern Code Review (MCR), in which reviewers can suggest code improvements, such as refactorings, through comments and commits in Pull Requests (PRs). Past MCR studies treat all PRs as similar, regardless of whether they induce refactoring or not. We define a PR as refactoring-inducing when refactoring edits are performed after the initial commit(s) as either a result of reviewers' comments or spontaneous actions carried out by the PR author. This work explores code reviewing-related aspects intending to characterize refactoring-inducing PRs. For that, we mined refactoring edits and code review data from GitHub. Then, we carried out empirical studies to identify similarities/dissimilarities between refactoring-inducing and non-refactoring-inducing PRs and characterize code review and refactoring edits in refactoring-inducing PRs. We found significant differences between refactoring-inducing and non-refactoring-inducing PRs and empirical evidence on the relevance of code review to refactoring edits at the PR level. We observed motivating factors behind refactoring-inducing PRs, identified structural aspects of review comments in refactoring-inducing and non-refactoring-inducing PRs, and proposed guidelines for a more productive code review. We also found empirical evidence on technical aspects characterizing refactorings in refactoring-inducing PRs. Our findings suggest directions for researchers, practitioners, and tool builders to improve practices around pull-based code review.

Keywords: Refactoring-Inducing Pull Request; Code Review; Empirical Study; Computing Methodology and Techniques; Software Engineering.

To all those who taught me some lesson

and all my illiterate ancestors,

affectionately.

Contents

1 Introduction **1**

1.1 Motivation 1

1.2 Problem Statement 4

1.3 Motivating Example 6

1.4 Objectives 8

1.5 Research Questions 8

1.6 Main Results and Implications of Research 9

1.7 Document Structure 11

2 Background **12**

2.1 Refactoring 12

2.1.1 Identification of Candidates for Refactoring 14

2.1.2 Application of Refactoring 15

2.1.3 Refactoring Detection 16

2.2 Modern Code Review 18

2.3 Git-based Development and PRs 20

2.4 Association Rule Learning 25

2.5 Concluding Remarks 33

3 Mining Refactoring Edits and Code Review Data **34**

3.1 Data Mining Design 34

3.1.1 Mining Merged PRs 34

3.1.2 Detection of Refactoring Edits 39

3.1.3 Mining Code Review Data 45

3.2 Limitations 51
3.3 Concluding Remarks 54

4 Comparing Refactoring-Inducing and non-Refactoring-Inducing Pull Requests 55
4.1 Research Design 55
4.1.1 Association Rule Learning 56
4.1.2 Data Analysis 62
4.2 Results and Discussion 65
4.2.1 How Common are Refactoring-Inducing PRs? 65
4.2.2 How Do Refactoring-Inducing PRs Compare to non-Refactoring-Inducing Ones? 66
4.2.3 Is Refactoring Induced by Code Reviews? 72
4.3 Implications 74
4.4 Limitations 76
4.5 Concluding Remarks 77

5 Characterizing Code Review in Refactoring-Inducing Pull Requests 79
5.1 Research Design 79
5.1.1 Selection of a Purposive Sample 80
5.1.2 Sanity Check of Refactoring Edits and Manual Analysis of Review Comments 84
5.1.3 Merging of Results 85
5.2 Results and Discussion 86
5.2.1 Preliminary Results 87
5.2.2 How Are Review Comments Characterized in Refactoring-Inducing and non-Refactoring-Inducing PRs? 97
5.2.3 What Are the Differences Between Refactoring-Inducing and non-Refactoring-Inducing PRs, in Terms of Review Comments? 107
5.2.4 How Do Reviewers Suggest Refactorings in Refactoring-Inducing PRs? 112
5.2.5 Do Suggestions of Refactoring Justify the Reasons? 113

5.2.6 What Is the Relationship Between Suggestions and Actual Refactorings in Refactoring-Inducing PRs? 115
5.3 Implications and Guidelines 116
5.4 Limitations 120
5.5 Concluding Remarks 121

6 Characterizing Refactoring Edits in Refactoring-Inducing Pull Requests 122
6.1 Research Design 122
6.2 Results and Discussion 125
6.2.1 What Types of Refactoring Edits often Take Place in PRs? . . . 125
6.2.2 How are the Refactoring Edits Characterized? 133
6.3 Implications 140
6.4 Limitations 141
6.5 Concluding Remarks 142

7 Related Work 143
7.1 Characterization of Code Review 143
7.2 Characterization of Refactoring Edits throughout Code Evolution . . . 148
7.3 Concluding Remarks 151

8 Conclusions 153

References 157

A Initial Investigations on Refactoring and Modern Code Review 176

B Descriptive Statistics of pull requests data 179

C Checking for Parametric Tests Assumptions and Results of Statistical Testing of Hypotheses 186

D Complements to the Characterization of Code Review in Refactoring-Inducing Pull Requests 199

List of Symbols

API – *Application Programming Interface*

ASF – *Apache Software Foundation*

AST – *Abstract Syntax Tree*

ARL – *Association Rule Learning*

CLES – *Common-Language Effect Size*

DVCS – *Distributed Version Control System*

FP – *Frequent Pattern*

GUI – *Graphical User Interface*

IDE – *Integrated Development Environment*

IQR – *InterQuartile Range*

MCR – *Modern Code Review*

ML – *Machine Learning*

OSS – *Open-Source Software*

PMC – *Project Management Committee*

PR – *Pull Request*

p-value – *probability value*

REST – *REpresentational State Transfer*

SD – *Standard Deviation*

SHA-1 – *Secure Hash Algorithm 1*

SPLab/UFCG – *Software Practices Laboratory*

URL – *Uniform Resource Locator*

List of Figures

1.1 A refactoring-inducing PR (Apache Tinkerpop PR #1110), illustrating initial commits ($c_1 - c_2$) and subsequent commits ($c_3 - c_4$). 5
1.2 Examples of refactoring-inducing PRs from Apache Drill repository . . 7

2.1 The commit message #141112 of the Eclipse's repository Egit on Gerrit (a) and the respective RefactoringMiner's output (b) 17
2.2 A commit from Apache Hadoop-Ozone PR #1033 at GitHub 19
2.3 Examples of Git branches 21
2.4 An overview of a Git PR in GitHub 22
2.5 An overview of Git PR review in GitHub 22
2.6 A partial diff output (reviewing feature) of committed changes into the Apache Drill PR #1888 at GitHub 23
2.7 A partial diff output (differentiating feature) of committed changes into the Apache Drill PR #1888 at GitHub 24
2.8 Illustrating merge PR option (Apache Accumulo-Examples PR #19) . 25
2.9 Illustrating squash and merge option (Apache Accumulo PR #106) . . 25
2.10 Illustrating rebase and merge option (Apache Accumulo PR #190) . . 26
2.11 Example of one-hot encoding 28
2.12 Example of transactions 28
2.13 Example of the FP-tree formation 30
2.14 Example of the bottom-up approach for generation of frequent itemsets 32

3.1 Overview of the data mining design 35
3.2 An overview of Apache Drill PR #1807, illustrating squashed commits ($c_1 - c_{12}$) 38

3.3 A commit, from Apache Fluo PR #1077, consisting of four *Inline Method* instances as detected by RefactoringMiner 42
3.4 A commit, from Apache Kafka PR #4757, consisting of two *Pull Up Method* instances as detected by RefactoringMiner 43
3.5 A commit, from Apache Beam PR #6261, consisting of one *Extract Class*, three *Move Attribute* and on *Move Method* instances as detected by RefactoringMiner 44
3.6 Illustrating a PR's commit presenting two parents (Apache Avro PR #537) 49
3.7 Creation and merge dates of the final sample's PRs 50

4.1 An overview of the comparative study between refactoring-inducing and non-refactoring-inducing PRs 57
4.2 Distribution of refactoring edits in the refactoring-inducing PRs 65

5.1 An overview of round *i* of our characterization study of code review in refactoring-inducing PRs 81
5.2 A single review comment, from Apache Incubator-Pinot PR #479, which induced no refactoring edit 89
5.3 A review comment, from PR Apache Cloudstack #2071, which induced a *Rename Package* edit 89
5.4 Example of self-affirmed refactoring in five subsequent commits, from Apache Usergrid PR #102 90
5.5 Example of a GitHub profile (author of Apache Kafka PR #5784) . . . 95
5.6 An uncertain review comment Apache Flink PR #9143 102
5.7 A typical suggestion of *Rename* edit from Apache Samza PR #1051 . . 108
5.8 A review comment on a minor scope issue from Apache Beam PR #6050 108
5.9 A review comment on a major scope issue from Apache Tinkerpop PR #1110 109
5.10 A typical discussion in non-refactoring-inducing PRs, from Apache Brooklyn-Server PR #411 110

5.11 A direct review comment, from Apache Dubbo PR #3299, which induced a *Change Variable Type* edit 112
5.12 A review comment providing a reason (code conventions), from Apache Brooklyn-Server PR #1049, which induced a *Split Attribute* edit 112
5.13 A warning on a conditional to deal with enumerations, from Apache Cloudstack PR #2833, which induced a *Change Variable Type* and a *Rename Variable* edits 113
5.14 A review comment providing a reason (secure-related issues), from Apache Cloudstack PR #3454, which induced two *Push Down Attribute* and two *Push Down Method* edits 114
5.15 A warning on responsibility of methods, from Apache Incubator-Iceberg PR #119, which induced a *Change Return Type*, a *Rename Method*, a *Change Parameter Type*, and a *Rename Parameter* edits 115
5.16 A suggestion of a refactoring, from Apache Kafka PR #5946, which induced an *Extract Variable* edit 116
5.17 A review comment providing a reason (long class body), from Apache Flink PR #7945, which induced two *Move and Rename Class* edits . . 117
5.18 A warning on a class useless, from Apache Dubbo PR #2279, which induced a *Change Variable Type* and a *Rename Variable* edits 117
5.19 A review comment, from Apache Flink PR #8620, which induced a *Rename Method* edit 118
5.20 A review comment including an embedded code from Apache Dubbo PR #4870 118

6.1 An overview of the characterization study of refactoring edits in refactoring-inducing PRs 123
6.2 Number of types of refactoring detected by refactoring-inducing PR . . 131
6.3 A *Change Attribute Type* edit in Apache Flink PR #8222 132
6.4 A *Move Class* edit that induced a review comment in Apache Kafka PR #5194 134
6.5 Number of subsequent commits in refactoring-inducing PRs 135

6.6 Scatterplot of number of refactorings in relation to number of subsequent commits in 449 refactoring-inducing PRs 136

7.1 An overview of this thesis concerning other works 152

B.1 Number of subsequent commits in the sample's PRs 180
B.2 Number of added lines in the sample's PRs 180
B.3 Number of deleted lines in the sample's PRs 181
B.4 Number of file changes in the sample's PRs 182
B.5 Number of reviewers in the sample's PRs 182
B.6 Number of review comments in the sample's PRs 183
B.7 Length of discussion in the sample's PRs 184
B.8 Time to merge in the sample's PRs 184
B.9 Number of refactoring edits in the sample's PRs 185

C.1 Number of added lines by presence of refactorings 188
C.2 Number of deleted lines by presence of refactorings 189
C.3 Number of file changes by presence of refactorings 191
C.4 Number of subsequent commits by presence of refactorings 192
C.5 Number of review comments by presence of refactorings 193
C.6 Length of discussion by presence of refactorings 195
C.7 Number of reviewers by presence of refactorings 196
C.8 Time to merge by presence of refactorings 198

List of Tables

2.1 A summary of a few code review aspects raised from empirical studies . 20
2.2 Relationship between antecedent and consequent according to the lift value 27
2.3 Example of conditional pattern base, conditional FP-tree, and the respective generated frequent pattern rules 32

3.1 Search results for merged PRs from Apache's non-archived Java repositories in GitHub 36
3.2 Top 5 repositories containing PRs merged by *merge pull request* option 37
3.3 Top 5 repositories containing PRs merged by *squash and merge* or *rebase and merge* options 37
3.4 A RefactoringMiner output example 40
3.5 Types of refactoring edits detectable by RefactoringMiner 1.0 in September 2019, classified in line with technical descriptions by Fowler 40
3.6 Output of RefactoringMiner execution in Apache's repositories 45
3.7 RefactoringMiner output, at commit and PR levels, for Apache's repositories 45
3.8 Top 5 repositories containing PRs merged by *merge pull request* option in sample 1 46
3.9 Top 5 repositories containing PRs merged by *squash and merge* or *rebase and merge* options in sample 1 46
3.10 Selected PR attributes for mining 47
3.11 A summary of the final sample from mining Apache merged PRs in GitHub 49
3.12 Top 5 Apache's repositories in the final sample 50

3.13 Validity and reliability countermeasures for mining refactoring edits and code reviewing-related data 52

4.1 One-hot encoding for binning of features 59
4.2 ARL output by experimenting minimum support from 0.01 to 0.1 by steps of 0.01, and confidence of 0.5 60
4.3 Association rules selected by manual inspection (AR_1–AR_4 for refactoring-inducing PRs, AR_5–AR_8 for non-refactoring-inducing PRs) . 66
4.4 Descriptive statistics of PR attributes 67
4.5 Results of the manual validation of refactoring edits mined by RefactoringMiner 1.0 73
4.6 Refactoring-inducing PRs, in which refactoring edits were induced by code review, by level of refactoring. 73
4.7 Validity and reliability countermeasures for comparing refactoring-inducing and non-refactoring-inducing PRs 76

5.1 A summary of the main objective of each round of qualitative analysis . 81
5.2 Summary of the purposive samples 84
5.3 Results of the manual validation of refactoring edits mined by RefactoringMiner 1.0 84
5.4 Inducement by code review in refactoring-inducing PRs 87
5.5 Refactoring-inducing PRs containing self-affirmed refactorings in their subsequent commits 91
5.6 Type of changes by category of PRs 92
5.7 Characteristics of review comments in refactoring-inducing PRs, with refactorings led by the authors 98
5.8 Characteristics of review comments in refactoring-inducing PRs, with refactorings induced by code review 100
5.9 Characteristics of review comments in refactoring-inducing PRs, with refactorings both led by the authors and induced by code review 102
5.10 Characteristics of review comments in non-refactoring-inducing PRs . . 105

5.11 Validity and reliability countermeasures for characterizing code review in refactoring-inducing PRs 120

6.1 Sample of refactoring-inducing PRs for the quantitative analysis 124
6.2 Sample of refactoring-inducing PRs for the qualitative analysis 124
6.3 Types of refactoring edits detected by RefactoringMiner 1.0 in the 449 refactoring-inducing PRs 125
6.4 Kinds of refactoring edits detected by RefactoringMiner 1.0 in the 449 refactoring-inducing PRs 127
6.5 Types of refactoring edits detected by RefactoringMiner 1.0 in 136 refactoring-inducing PRs consisting of a single type of refactoring . . . 128
6.6 Kinds of refactoring edits detected by RefactoringMiner 1.0 in 136 refactoring-inducing PRs consisting of a single type of refactoring . . . 129
6.7 Kinds of refactorings in 65 refactoring-inducing PRs (validated refactorings) 132
6.8 Number of detected refactorings from the first until the third subsequent commit in refactoring-inducing PRs 135
6.9 Low-level and high-level refactorings detected by RefactoringMiner in 449 refactoring-inducing PRs 137
6.10 Types of high-level refactorings detected by RefactoringMiner 1.0 in 449 refactoring-inducing PRs 137
6.11 Number of low-level and high-level refactorings in 65 refactoring-inducing PRs (validated refactorings) 138
6.12 Kinds of refactorings by type of change in 65 refactoring-inducing PRs (validated refactorings) 139
6.13 Validity and reliability countermeasures for the characterization study of refactoring edits in refactoring-inducing PRs 142

7.1 A summary of characterization studies on code review 143
7.2 A summary of characterization studies on refactorings over code evolution148

A.1 A summary of the preliminary investigation results 178

B.1 Descriptive statistics – number of subsequent commits 179
B.2 Descriptive statistics – number of added lines 180
B.3 Descriptive statistics – number of deleted lines 181
B.4 Descriptive statistics – number of file changes 181
B.5 Descriptive statistics – number of reviewers 182
B.6 Descriptive statistics – number of review comments 183
B.7 Descriptive statistics – length of discussion 183
B.8 Descriptive statistics – time to merge 184
B.9 Descriptive statistics – number of refactoring edits 185

C.1 Statistical tests output – number of added lines by presence of refactorings 188
C.2 Statistical tests output – number of deleted lines by presence of refactorings 189
C.3 Statistical tests output – number of file changes by presence of refactorings 190
C.4 Statistical tests output – number of subsequent commits by presence of refactorings 192
C.5 Statistical tests output – number of review comments by presence of refactorings 194
C.6 Statistical tests output – length of discussion by presence of refactorings 195
C.7 Statistical tests output – number of reviewers by presence of refactorings 196
C.8 Statistical tests output – time to merge by presence of refactorings . . . 197

D.1 Descriptive statistics (refactoring-inducing PRs) 199
D.2 Descriptive statistics (non-refactoring-inducing PRs) 200
D.3 Type of changes by category of PRs 202
D.4 Presence of self-affirmed minor PRs 203
D.5 Presence of self-affirmed minor review comments 204
D.6 Presence of code review bot in PRs 205
D.7 Descriptive statistics of participants of PRs by category 206
D.8 Apache roles of the participants of PRs by category 207
D.9 Descriptive statistics of participants by subgroup of refactoring-inducing PRs 208
D.10 Age of PRs (in number of years) 209

List of Source Codes

2.1 A simple Java source code 12

2.2 A simple refactored Java source code 13

2.3 A simple example of a Java source code with *Feature Envy* 14

2.4 A simple example of a Java source code with no *Feature Envy* after a *Move Method* refactoring 15

6.1 A simple Java method 127

6.2 A simple Java refactored method 128

Chapter 1

Introduction

1.1 Motivation

Modern Code Review (MCR) is lightweight, tool-assisted, asynchronous, and driven by reviewing code changes [33]. Through the years, we perceive the transition from formal software inspection to MCR, in both open source and industrial software development. Thus, finding defects, the main objective of software inspection [59], became a limited operation of inspection [76], whereas regular change-based reviewing, in which code improvements are embraced, became an essential practice in the MCR scenario [33; 120]. Code changes may comprise new features, bug fixes, or other maintenance tasks, providing potential opportunities for refactorings [110], which form a significant part of the changes [35; 141]. Given that the nature of changes significantly affects code review effectiveness, as it directly influences how reviewers perceive the changes [117], the provision of suitable resources for assisting code review is essential.

Characterization studies of MCR have been conducted to investigate technical aspects of reviewing [38; 44; 73; 120; 121; 122; 126], factors leading to useful code review [45], circumstances that contribute to code review quality [80], and general code review patterns in pull-based development [87]. Those studies are relevant because MCR is critical in repository-based software, driven by change and collaboration, especially in *Agile software development* and *continuous delivery* approaches. A change-driven and collaboration-based strategy guides the principles of Agile practices. [14]. 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 [72].

In that scenario, Git-based software repositories have increasingly been adopted [13]. In practice, Git *Pull Requests* (PRs) are suitable to MCR as they promote well-defined and collaborative reviewing. A PR is a commonly used way for submitting contributions to collaboration-based projects [7]. Through PRs, the code is subject to a review process in which reviewers may suggest improvements before merging the code to the main branch of a repository [50]. *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 integrate the code developed in a branch into the main development line [50]. Code improvements may take the form of refactorings, resulting from discussions among the PR author and reviewers on code quality issues, including spontaneous actions of the author aiming to refine the originally submitted code.

Concerning the characterization of refactorings performed throughout code evolution, research has addressed the peculiarities of manual and automated refactoring edits over time [101], the relationship between different categories of code changes and specific refactoring types [110], and the commit-level assessment of contemporary refactoring practices [147]. Studies have investigated how developers document refactorings during the code evolution [27], and the impact of refactorings on merge conflicts [89]. Recently, studies have explored the reason for systems' architecture degradation after refactoring edits [107], the commit-level description of refactorings over time [47], the motivations behind refactorings at the PR level [112], and the characterization of the intents and evolution of refactorings during code reviewing [109].

We hypothesize that refactoring-inducing PRs have different characteristics from non-refactoring-inducing ones, as refactoring may involve design and *Application Programming Interface* (API) changes that require more extensive effort and knowledge of the project. Accordingly, we realized there is a lack of knowledge regarding the characterization of refactoring edits and technical aspects of code reviewing in the pull-based development in line with our definition of *refactoring-inducing PR* (Section 1.2).

We decided to work at the PR level because we understand a PR 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 PR level, we can obtain a global comprehension of contributions to the original code, in terms of both commits and reviewing-related aspects (e.g., 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 [153], and by findings from Pantiuchina et al. that considered discussions at the PR level as one of the motivating factors to refactorings [112].

Therefore, by distinguishing refactoring-inducing from non-refactoring-inducing PRs, we can potentially advance the understanding of code reviewing at the PR level and assist researchers, practitioners, and tool builders in this context. To the best of our knowledge, no prior MCR studies made a distinction between refactoring-inducing and non-refactoring-inducing PRs, when analyzing their research questions, which might have affected their findings or discussions. For instance, if Gousios et al. [68] and Kononenko et al. [82] had considered refactoring-inducing PRs, they might have found different factors influencing the time to merge a PR; Li et al. [87] could have included refactoring concerns to the multilevel taxonomy for review comments in the pull-based development model; Pascarella et al. [113] could have identified further information to perform a proper code review in presence of refactorings; Paixão et al. [109] could have complemented the study on the reasons for refactorings during code review when analyzing projects in Gerrit. Whereas, Pantiuchina et al. [112] could have drown different conclusions on motivations for refactorings in PRs, since they analyzed PRs in which refactorings were detected even in the initial commit(s) (i.e., these refactorings were not induced from reviewer discussions). In practice, being unaware of refactoring-inducing PRs' characteristics, practitioners and tool builders might miss opportunities to better manage their resources and to assist developers in PRs, respectively (Section 5.3).

This thesis aims to provide a systematical and practical understanding of the refactoring-inducing code review process in pull-based development. In this direction, we mine merged PRs from Apache's Java repositories in GitHub (Chapter 3) and investigate how common are refactoring-inducing PRs and what are the main similarities/dissimilarities between refactoring-inducing and non-refactoring-inducing PRs, considering metrics related to changes (e.g., number of changed files), particular characteristics of reviewing (e.g., review comments), and refactoring edits (Chapter 4); what are the main properties of code review in refactoring-inducing PRs (Chapter 5); and what are the typical refactoring types and their respective characteristics (Chapter 6). Specifically, we propose empirical studies driven by quantitative and qualitative methods in order to produce a more accurate characterization. Those studies are complementary, intending to characterize refactorings and code reviewing-related aspects in the refactoring-inducement context (Section 1.2).

It is worth mentioning that this thesis has evolved from initial investigations regarding refactorings and MCR [18] (summarized in Appendix A), developed in order to get a better understanding of the topic and plan the research design. For that, 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 [124].

1.2 Problem Statement

Pull-based development, as implemented in GitHub, constitutes an useful strategy for MCR practice. Through PRs, developers can contribute to source code improvements, such as triggering refactoring edits [112]. This scenario leads to a question: what characterizes PRs that induce refactorings?

By filling this knowledge gap, we may provide an extension in the understanding of MCR in the pull-based development model and support practical recommendations and further research. In this sense, this thesis investigates the technical aspects of code review and refactoring edits in GitHub PRs, in light of the following definition of a refactoring-inducing PR[1]. Accordingly, we define a PR as refactoring-inducing

[1]This designation was proposed by professor Nikolaos Tsantalis (Concordia University, Canada) in one of our technical meetings, so initially inspiring this thesis.

when refactoring edits are performed after the initial commit(s), as either a result of discussion among reviewers or spontaneous actions carried out by the PR author. Thus, we consider the PRs that comprise at least one review comment.

Definition 1. *A pull request is refactoring-inducing if refactoring edits 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 author. Let* $U = \{u_1, u_2, ..., u_w\}$*, a set of repositories in GitHub. Each repository* u_q*,* $1 \leq q \leq w$*, has a set of pull requests* $P(u_q) = \{p_1, p_2, ..., p_m\}$ *over time. Each pull request* p_j*,* $1 \leq j \leq m$*, has a set of commits* $C(p_j) = \{c_1, c_2, ..., c_n\}$*, in which* $I(p_j)$ *is the set of initial commits included in the pull request when it is created and* $S(p_j)$ *is the set of subsequent commits incorporated into the pull request after its creation,* $I(p_j) \subseteq C(p_j)$ *and* $S(p_j) \subseteq C(p_j)$*. A refactoring-inducing pull request is that in which* $\exists\ c_k \mid R(c_k) \neq \varnothing$ *and* $\exists\ c_i \mid RC(c_i) \neq \varnothing$*, where* $R(c_k)$ *and* $RC(c_i)$ *respectively denote the set of refactorings performed in commit* c_k *and the set of review comments left in commit* c_i*,* $|I(p_j)| < k \leq n$ *and* $c_i \in I(p_j) \cup S(p_j)$*.*

To clarify our definition, Figure 1.1 depicts a refactoring-inducing PR consisting of two initial commits ($c_1 - c_2$) and two subsequent commits ($c_3 - c_4$), which include refactoring edits, e.g., commit c_3 has one *Extract Method* edit. The PR also comprises review comments in commits c_2 and c_3. Our studies explore PR commits subsequent to the initial ones ($c_1 - c_2$).

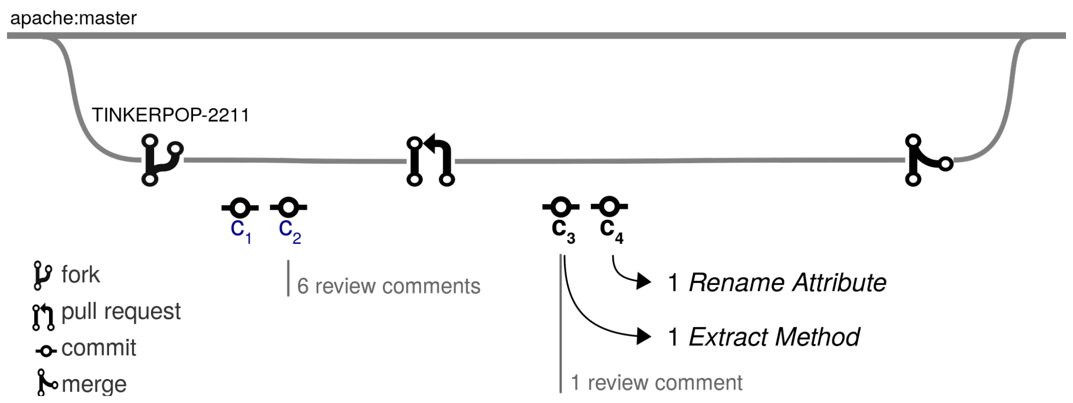


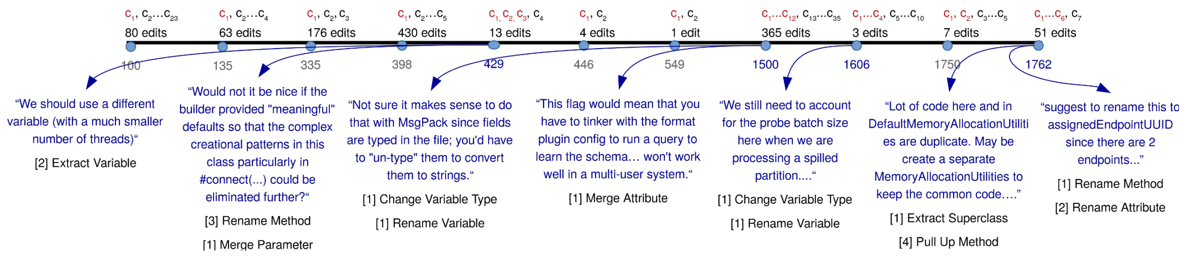
Figure 1.1: A refactoring-inducing PR (Apache Tinkerpop PR #1110), illustrating initial commits ($c_1 - c_2$) and subsequent commits ($c_3 - c_4$).

It is noteworthy that we recognize different schools leading the refactoring subject [135]. Thus, we consider refactoring an edit that changes the internal code structure while preserving its external behavior and as practiced in agile development (interchangeable). More specifically, we assume refactorings as mechanisms defined by Tsantalis et al. to implement *RefactoringMiner* [146] – a state-of-the-art tool used for refactoring detection in this thesis (Chapter 3, Subsection 3.1.2).

1.3 Motivating Example

There are empirical evidence on the benefits of code review and refactoring to concerns such as code quality [33; 37; 42; 51; 99]. Since code review and performing refactorings are time-consuming tasks [29; 68; 139], a detailed understanding of the practices around refactorings and code reviewing may contribute to precise directions towards more productivity for practitioners (e.g., reducing delays in merging code changes in pull-based development). Moreover, by identifying differences between refactoring-inducing and non-refactoring PRs, we can support researchers investigating MCR in pull-based projects in designing more suitable strategies for sampling PRs. We believe that research questions may have distinct findings if we interchangeably consider the characteristics of code changes in PRs. In MCR, a reviewer provides suggestions through comments in code snippets while manually examines changes that, in turn, may comprise refactoring edits. Given that, we firstly explored the interactions among authors and reviewers in a pull-based project intending to get an initial understanding of how review comments induce refactoring edits in code review time.

This thesis has evolved from results of preliminary investigations on refactorings and code reviews (Appendix A). From those, as a motivating example, we describe a case history, in which we explored the refactoring-inducement and code review aspects. We randomly selected 24 PRs from Apache's Drill repository. Then, we ran RefactoringMiner and obtained 11 (45.8%) refactoring-inducing PRs indicated in Figure 1.2, which shows the number of initial commits (in red) and refactorings by PR. Moreover, we show a few review comments that directly induced refactorings.



$c_1, c_2 \ldots c_{23}$
80 edits
100
$c_1, c_2 \ldots c_4$
63 edits
135
c_1, c_2, c_3
176 edits
335
$c_1, c_2 \ldots c_5$
430 edits
398
c_1, c_2, c_3, c_4
13 edits
429
c_1, c_2
4 edits
446
c_1, c_2
1 edit
549
$c_1 \ldots c_{12}, c_{13} \ldots c_{35}$
365 edits
1500
$c_1 \ldots c_4, c_5 \ldots c_{10}$
3 edits
1606
$c_1, c_2, c_3 \ldots c_5$
7 edits
1750
$c_1 \ldots c_6, c_7$
51 edits
1762
“We should use a different variable (with a much smaller number of threads)“
[2] Extract Variable
“Would not it be nice if the builder provided "meaningful" defaults so that the complex creational patterns in this class particularly in #connect(...) could be eliminated further?“
[3] Rename Method
[1] Merge Parameter
“Not sure it makes sense to do that with MsgPack since fields are typed in the file; you'd have to "un-type" them to convert them to strings.“
[1] Change Variable Type
[1] Rename Variable
“This flag would mean that you have to tinker with the format plugin config to run a query to learn the schema… won't work well in a multi-user system.“
[1] Merge Attribute
“We still need to account for the probe batch size here when we are processing a spilled partition....“
[1] Change Variable Type
[1] Rename Variable
“Lot of code here and in DefaultMemoryAllocationUtilities are duplicate. May be create a separate MemoryAllocationUtilities to keep the common code….”
[1] Extract Superclass
[4] Pull Up Method
“suggest to rename this to assignedEndpointUUID since there are 2 endpoints...”
[1] Rename Method
[2] Rename Attribute

Figure 1.2: Examples of refactoring-inducing PRs from Apache Drill repository

We compared refactoring-inducing and non-refactoring-inducing PRs concerning *code churn* (number of changed lines), and *length of discussion* (i.e., number of review and non-review comments). As a result, we identified that the refactoring-inducing PRs presented a higher code churn and discussion length than non-refactoring-inducing PRs. Note that we took into account one measure of each context under investigation: changes (code churn), code review (length of discussion), besides the number of refactoring edits.

Also, we manually analysed the refactoring-inducing PRs, by contrasting the descriptions of the detected refactorings by RefactoringMiner against review comments. Our strategy of analysis consisted of reading comments and searching for keywords (e.g., “*refac*”, “*mov*”, “*extract*”, and “*renam*”). We observed refactorings directly induced by review comments in four refactoring-inducing PRs. To exemplify, in the PR #1762[2], the review comment “*Lot of code here and in DefaultMemoryAllocationUtilities are duplicate. May be create a separate MemoryAllocationUtilities to keep the common code...*” motivated one *Extract Superclass* and four *Pull Up Method* refactorings.

In a nutshell, those results provided insights on the pertinence of (i) exploring technical aspects of changes, code review, and refactorings in the PR level, since we perceived differences between refactoring-inducing and non-refactoring-inducing PRs in terms of code churn and length of discussion; (ii) considering refactorings as part of contributions to the code improvement during code review, and (iii) investigating quantitatively and qualitatively technical aspects in light of our refactoring-inducing PR definition.

[2]Apache Drill PR #1762, available in `https://git.io/JczHh`.

1.4 Objectives

The main goal of this thesis is to establish an in-depth understanding of refactoring-inducing PRs. Given that, we define the following objectives:

- Build datasets of code review data and detected refactorings in merged PRs;
- Identify similarities/dissimilarities between refactoring-inducing and non-refactoring-inducing PRs;
- Characterize code review aspects in refactoring-inducing PRs;
- Characterize refactoring edits in refactoring-inducing PRs; and
- Provide directions for researchers, practitioners, and tool builders.

1.5 Research Questions

We designed the following studies and respective research questions to address our objectives. Aiming to identify similarities/dissimilarities between refactoring-inducing and non-refactoring-inducing PRs, we formulated these **research questions**:

- RQ_1: How common are refactoring-inducing PRs?
- RQ_2: How do refactoring-inducing PRs compare to non-refactoring-inducing ones?
- RQ_3: Are refactoring edits induced by code reviews?

To investigate code reviewing-related aspects intending to characterize code review in refactoring-inducing PR, we designed these research questions:

- RQ_1: How are review comments characterized in refactoring-inducing and non-refactoring-inducing PRs?
- RQ_2: What are the differences between refactoring-inducing and non-refactoring-inducing PRs, in terms of review comments?
- RQ_3: How do reviewers suggest refactorings in review comments in refactoring-inducing PRs?

- RQ_4: Do suggestions of refactoring justify the reasons?
- RQ_5: What is the relationship between suggestions of refactoring and actual refactorings in refactoring-inducing PRs?

Intending to characterize refactoring edits in refactoring-inducing PRs, we formulated the following research questions:

- RQ_1: What types of refactoring edits often take place in PRs?
- RQ_2: How are the refactoring edits characterized?

1.6 Main Results and Implications of Research

By characterizing refactoring-inducing PRs, we can potentially advance the understanding of code review process at the PR level and expand the research regarding contemporary code review. Our main results are:

- We found significant differences between refactoring-inducing and non-refactoring-inducing PRs (in terms of code churn, number of file changes, number of subsequent commits, number of review comments, length of discussion, and time to merge), and empirical evidence on the relevance of code review to refactoring edits at the PR level;
- We observed motivating factors behind refactoring-inducing PRs (e.g., their reviewers are usually more experienced than their authors), identified technical aspects of the structure of review comments in both refactoring-inducing and non-refactoring-inducing PRs (e.g., polite and precise review comments in refactoring-inducing PRs with refactorings induced by code review), and proposed guidelines for a more productive code reviewing; and
- We found empirical evidence regarding technical aspects characterizing refactorings (e.g., PR authors tend to perform refactoring edits in initial subsequent commits for addressing code review suggestions) in refactoring-inducing PRs.

Our findings provide actionable implications, summarized as follows. We give a complete description of them and other implications in the respective section of our studies (Chapters 4–6).

From our comparative study between refactoring-inducing and non-refactoring-inducing PRs, we suggest that:

- future experiment designs on MCR with PRs to consider the distinction between refactoring-inducing and non-refactoring-inducing PRs,
- PR managers to better manage the reviewers when a PR becomes refactoring-inducing (e.g., to share the knowledge of design changes caused by subsequent refactoring edits to more team members), and
- tool builders to develop tools for helping the developers to distinguish refactoring edits from non-refactoring edits directly in the review board of pull-based development platforms.

From our characterization study on code review in refactoring-inducing PRs, we propose that:

- future research should explore strategies to support an effective code review participation of authors and reviewers, and knowledge transfer at the PR level (e.g., by specifying requirements for reviewer recommendation in line with our findings).
- practitioners should follow our guidelines for articulating valuable review comments towards code refactoring.
- tool builders may implement checkers for review comments as a feature available at code review boards, guided by our guidelines for reviewers.

From our characterization study of refactoring edits in refactoring-inducing PRs, we recommend that:

- future research on the practice of refactoring at the PR level can investigate code repositories to identify patterns related to missing opportunities of implementing more complex refactorings at code review time.

- practitioners should pay more attention to resources for assisting the code review process (e.g, our guidelines for reviewers).
- tool builders can implement reviewer recommendation tools based on the experience of them against authors.

Moreover, as a general recommendation for researchers, we point out that researchers can use our mined data, developed tools, and research methods, publicly available [22], to further investigate code reviewing in pull-based development and perform replication studies.

1.7 Document Structure

We organize the following chapters. In Chapter 2, we provide a conceptual background regarding refactorings, MCR, Git-based development and PRs, and *Association Rule Learning* (ARL). We describe the procedures for mining merged PRs and refactoring edits for our empirical studies in Chapter 3. Then, we present the research design, results, discussion, and limitations of the studies carried out for characterizing refactoring-inducing PRs: a comparative study between refactoring-inducing and non-refactoring-inducing PRs (Chapter 4), a characterization study of code review in refactoring-inducing PRs (Chapter 5), and a characterization study of refactoring edits in refactoring-inducing PRs (Chapter 6). In Chapter 7, we discuss the characterization studies related to this thesis, followed by conclusions in Chapter 8, and appendices.

Chapter 2

Background

This chapter addresses the concepts that contextualize this thesis: the fundamentals of refactoring (Section 2.1), Modern Code Review (Section 2.2), Git-based development and PRs (Section 2.3), and *Association Rule 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 [77]. Design comprises the internal software structure, while quality addresses the functional requirements and the structural aspects such as maintainability [133].

For that, restructuring edits, originally named *refactorings* by Opdyke and Johnson [105] and popularized by Fowler [60], are performed to directly favor design and the quality of object-oriented software, whereas preserving its external behavior. To clarify, suppose that a developer (inspired by Fowler) implemented the Java code 2.1.

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

Listing 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. The developer restructures the code so that the external behavior (the program output) remains unchanged, as in listing 2.2.

```java
// A simple Hello World in Java!
public class SimpleHelloWorld {
  public static void main(String ... args) {
    String name = "developer.", hw = "Hello World!";
    printDetails(name, hw);
  }
  //this is a simple example of extract method
  static void printDetails(String name, String hw) {
    System.out.println("Hi, I'm " + name);
    System.out.println("This is my " + hw);
  }
}
```

Listing 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. This code transformation is a refactoring called *Extract Method*, which objective is to extract a method from a code snippet so that the method's name indicates its purpose. With this refactoring, developers often aim at improving code readability, achieved from an internal restructuring of the code, thereby supporting its maintainability. For further exploration, Fowler provides an updated catalog of more than 60 types of refactorings [61].

A general recommendation is that refactoring edits should be performed in a systematic and structured manner based on specific steps to (1) identify candidates for refactoring, and (2) perform the refactoring edits [60; 104], as described in Subsections 2.1.1 and 2.1.2. As a complement, developers may seek the applied refactoring edits 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

Refactoring is mainly motivated by changes in the code requirements (e.g., a new feature) or the presence of *bad smells*. For instance, an extracted code makes the original method easier to understand (as exemplified in Listing 2.2) and facilitates adding a new feature either in the extracted method or in the original one, so illustrating two of the reasons for such a refactoring: *improving readability* and *promoting extension*, respectively [129]. Bad smells are signs of deeper problematic situations [39], therefore potential candidates for refactoring in a code. For instance, consider one of the main problems for object-oriented design – *Feature Envy*, as exemplified in listing 2.3. This bad smell arises when a method (printEarthGreetings()) is a member of a class (HelloUniverse), however, its responsibilities are similar to the responsibilities of another class (HelloEarth).

```java
// A simple Hello Universe in Java!
class HelloUniverse{
  private String greeting = "Hello Universe!";
  public void printGreetings() {
    System.out.println(greeting);
    HelloEarth earth = new HelloEarth();
    printEarthGreetings(earth);
  }
  public void printEarthGreeting(HelloEarth earth) {
    System.out.println(earth.getGreeting());
  }
}
// A simple Hello Earth in Java!
class HelloEarth{
  private String greeting = "Hello Earth!";
  public String getGreeting() {
    return greeting;
  }
}
```

Listing 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 apply a *Move Method* [60] refactoring by moving printEarthGreeting() from class

HelloUniverse to class HelloEarth, as in listing 2.4.

```java
// A simple Hello Universe in Java!
class HelloUniverse{
  private String greeting = "Hello Universe!";
  public void printGreetings() {
    System.out.println(greeting);
    HelloEarth earth = new HelloEarth();
    earth.printEarthGreetings();
  }
}
// A simple Hello Earth in Java!
class HelloEarth{
  private String greeting = "Hello Earth!";
  public String getGreeting() {
    return greeting;
  }
  public void printEarthGreetings() {
    System.out.println(getGreeting());
  }
}
```

Listing 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 [88]. For this thesis, 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) [61; 100]. These edits can be performed manually or supported by tools.

When manual, the transformations are time-consuming and susceptible to errors [29; 56]. 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 [131].

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;

2. the engine checks whether the restructuring meets the preconditions for the type of refactoring (e.g., no conflicts with existing 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.

2.1.3 Refactoring Detection

Refactoring detection consists of the automatic identification of the refactoring edits applied in a given code. It is often used for assisting studies on code evolution [15; 36; 110; 148], understanding of changes [115], and code review [29; 65]. In practice, refactoring detection is supported by tools based on different techniques, such as predicate logic in RefFinder [115], heuristics based on static analysis and code similarity in RefDiff [130], and *Abstract Syntax Tree* (AST)-based associations in RefactoringMiner[4] [146].

To illustrate tool-supported refactoring detection, Figure 2.1 (a) shows a commit message[5] informing on *Rename* edits, 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 that a reviewer could run the refactoring detection tool for obtaining complementary information to the ones indicated in the commit message

[1]`https://eclipse.org`

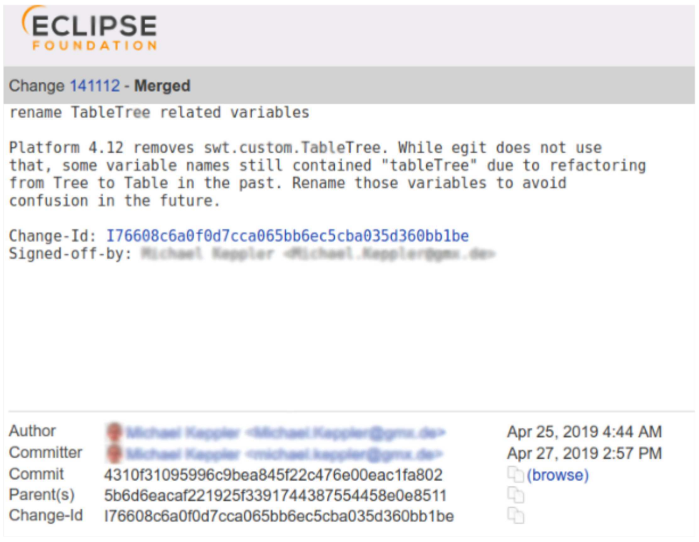
[2]`https://jetbrains.com/idea/`

[3]`https://netbeans.org`

[4]`https://github.com/tsantalis/RefactoringMiner`

[5]`https://git.eclipse.org/r/#/c/141112/`

in Figure 2.1 (a), so getting details regarding the refactoring edits applied in Figure 2.1 (b).

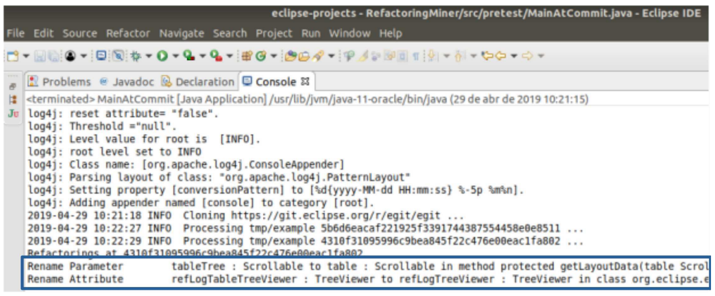


Change 141112 - **Merged**

```
rename TableTree related variables

Platform 4.12 removes swt.custom.TableTree. While egit does not use
that, some variable names still contained "tableTree" due to refactoring
from Tree to Table in the past. Rename those variables to avoid
confusion in the future.

Change-Id: I76608c6a0f0d7cca065bb6ec5cba035d360bb1be
Signed-off-by: [redacted]
```

Author	[redacted]	Apr 25, 2019 4:44 AM
Committer	[redacted]	Apr 27, 2019 2:57 PM
Commit	4310f31095996c9bea845f22c476e00eac1fa802	(browse)
Parent(s)	5b6d6eacaf221925f3391744387554458e0e8511	
Change-Id	I76608c6a0f0d7cca065bb6ec5cba035d360bb1be	

(a)



eclipse-projects - RefactoringMiner/src/pretest/MainAtCommit.java - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help
Problems Javadoc Declaration Console

```
<terminated> MainAtCommit [Java Application] /usr/lib/jvm/java-11-oracle/bin/java (29 de abr de 2019 10:21:15)
log4j: reset attribute= "false".
log4j: Threshold ="null".
log4j: Level value for root is  [INFO].
log4j: root level set to INFO
log4j: Class name: [org.apache.log4j.ConsoleAppender]
log4j: Parsing layout of class: "org.apache.log4j.PatternLayout"
log4j: Setting property [conversionPattern] to [%d{yyyy-MM-dd HH:mm:ss} %-5p %m%n].
log4j: Adding appender named [console] to category [root].
2019-04-29 10:21:18 INFO  Cloning https://git.eclipse.org/r/egit/egit ...
2019-04-29 10:22:27 INFO  Processing tmp/example 5b6d6eacaf221925f3391744387554458e0e8511 ...
2019-04-29 10:22:29 INFO  Processing tmp/example 4310f31095996c9bea845f22c476e00eac1fa802 ...
Refactorings at 4310f31095996c9bea845f22c476e00eac1fa802
Rename Parameter        tableTree : Scrollable to table : Scrollable in method protected getLayoutData(table Scrol
Rename Attribute        refLogTableTreeViewer : TreeViewer to refLogTreeViewer : TreeViewer in class org.eclipse.e
```

(b)

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

The effectiveness of those tools is often measured by *accuracy*, calculated from the *precision* and *recall*, according to Equations (2.1) and (2.2).

$$precision = \frac{|TP|}{|TP| + |FP|} \tag{2.1}$$

$$recall = \frac{|TP|}{|TP| + |FN|} \tag{2.2}$$

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 detection, 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) or 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 due to its high accuracy (precision of 97.9% and recall of 87.2%) [146]. It is presented as a Java API to detect refactorings applied to in the history of a Java project [11], which running may be simplified in Equation (2.3).

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

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 3.5.

For this thesis context, it is worth being aware that we investigate refactoring edits from the output of RefactoringMiner running, thus driven by its refactoring mechanisms.

2.2 Modern Code Review

MCR, as defined by Bacchelli and Bird [33], consists of a lightweight code review (in opposition to the formal code inspection specified by Fagan [59]), 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 [94]. This resource is supported by platforms such as GitHub [6], *Gerrit* [2], *Phabricator* [10], and *Review Board* [12].

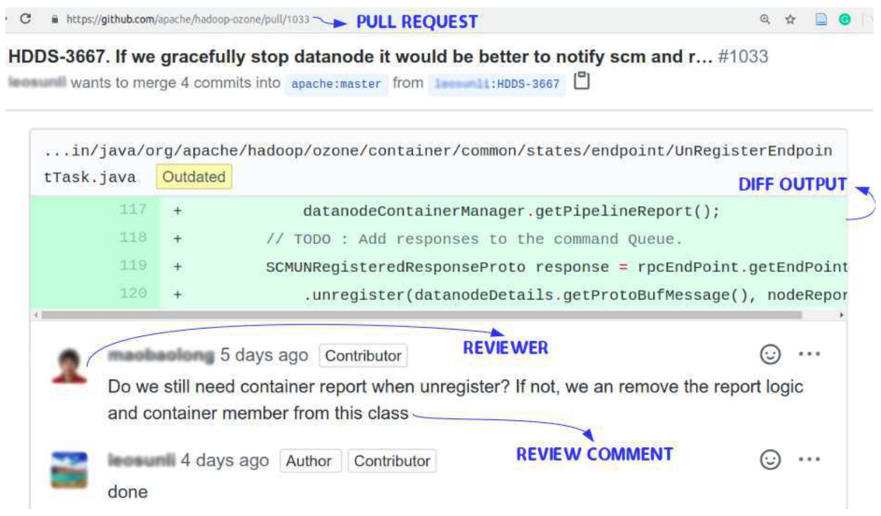
Essentially, the code review process is a manual examination of the code changes (*patch*), submitted by a developer (author), by another developer (reviewer) – or reviewers. The process usually proceeds the following workflow, as illustrated in Figure 2.2[6], until the changes are accepted or discarded:

1. an author submits code changes – the added lines 117–120 in the file UnRegisterEndpointTask, in turn, component of a PULL REQUEST's commit;

2. the reviewer(s) – REVIEWER – inspects the code changes highlighted by code differentiating tools (*diff tools*) – 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 [126] – e.g., the comment "done" indicates that the author agrees with the reviewer's suggestion.



https://github.com/apache/hadoop-ozone/pull/1033

HDDS-3667. If we gracefully stop datanode it would be better to notify scm and r... #1033

[redacted] wants to merge 4 commits into `apache:master` from `[redacted]:HDDS-3667`

```
...in/java/org/apache/hadoop/ozone/container/common/states/endpoint/UnRegisterEndpoin
tTask.java  Outdated
117  +        datanodeContainerManager.getPipelineReport();
118  +      // TODO : Add responses to the command Queue.
119  +      SCMUNRegisteredResponseProto response = rpcEndPoint.getEndPoint
120  +          .unregister(datanodeDetails.getProtoBufMessage(), nodeRepor
```

[redacted] 5 days ago Contributor

Do we still need container report when unregister? If not, we an remove the report logic and container member from this class

[redacted] 4 days ago Author Contributor

done

Figure 2.2: A commit from Apache Hadoop-Ozone PR #1033 at GitHub

[6]Example of a diff output and comments during the code review process of a GitHub's PR, available at `https://github.com/apache/hadoop-ozone/pull/1033`.

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, summarized in Table 2.1.

Table 2.1: A summary of a few code review aspects raised from empirical studies

Code review aspect	***Empirical study(ies)***
change description	[45; 141]
code churn (number of changed lines)	[38; 80; 120; 143]
length of discussion	[80; 94; 120; 143]
number of changed files	[45; 80]
number of commits	[94; 121]
number of people in the discussion	[80]
number of resubmissions	[80; 120]
number of review comments	[40; 94; 120]
number of reviewers	[120; 127]
time to merge	[68; 73]

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

2.3 Git-based Development and PRs

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. Results from the 2020 Stack Overflow development survey indicate that more than 82% of developers use GitHub [21] – 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 [19]. Given that, we consider the Git-based development as implemented in GitHub.

As a DVCS, Git supplies collaborative development by mirroring 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 [50]. This change history is structured as a linked-list of snapshots, denominated commits that, in turn, are uniquely identified by a hexadecimal characters (e.g., 0a66da598a986971a77900442e20025ebfc56e9d[7]) generated using the *Secure Hash Algorithm 1* (SHA-1) hash algorithm [102].

Git organizes commits into multiple lines of development, named branches. Each repository has a main (or master) branch. Each additional line of development consists of a copy of the repository, which can be modified without altering the main branch. Figure 2.3[8] illustrates the idea behind the Git branches, as implemented in GitHub, displaying two lines of development created after forking (copying) the master-branch. As result, a developer can alter the repository's content to submit a new feature, via the new-feature-branch, and to fix a bug, via the bug-fix-branch, without altering the repository's content in the master-branch.

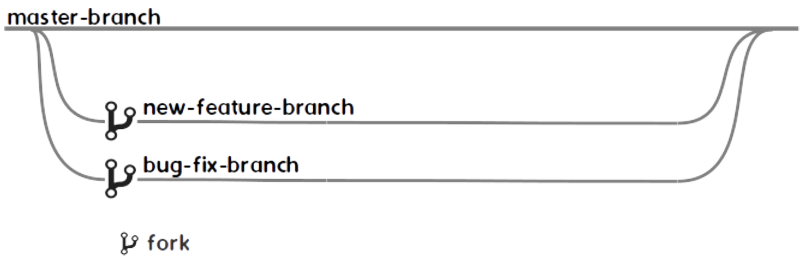


Figure 2.3: Examples of Git branches

After forking a Git branch, a developer can implement contributions (i.e., changes) through commits, and then, open a PR to submit them for discussion. At this moment, it occurs the code review process – the focus of investigations of this thesis. In this setting, as shown in Figures 2.4 and 2.5[9], a developer (author) forks the main repository branch (master-branch), creating a new branch (pr-branch), to make changes (a set of

[7]A commit of an Apache repository, available at `https://git.io/JUBG5`.

[8]Inspired by `https://guides.github.com/activities/hello-world/`.

[9]Both figures are inspired in `https://t.ly/EXH9`.

n commits, i.e., $c_1, c_2, ..., c_n$) and submit them through a PR (status *open pull request*). Next, one or more reviewers examine the submitted commits (CODE REVIEW), resulting in either adding new commits to the PR (CODE UPDATE) or closing the PR (status *closed pull request*) by the author.

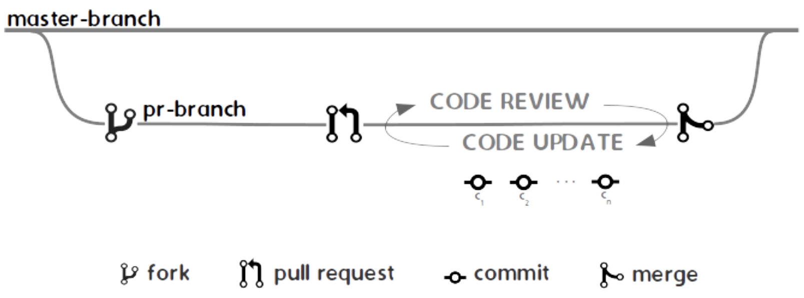


Figure 2.4: An overview of a Git PR in GitHub

Figure 2.5 emphasizes the MCR process inside an open PR. At that point, any developer with reading access can review the PR. 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 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.

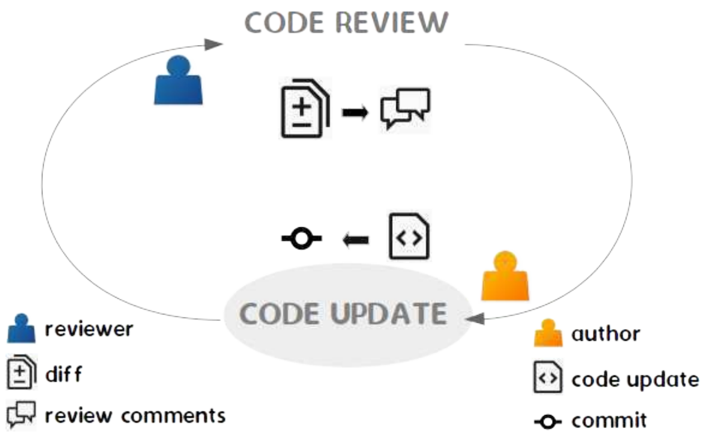
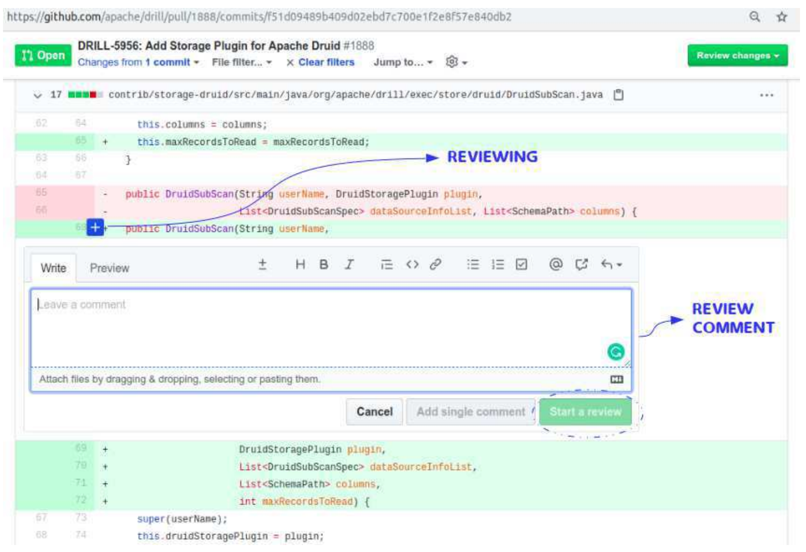


Figure 2.5: An overview of Git PR review in GitHub

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


https://github.com/apache/drill/pull/1888/commits/f51d09489b409d02ebd7c700e1f2e8f57e840db2
Open
DRILL-5956: Add Storage Plugin for Apache Druid #1888
Changes from 1 commit
File filter...
Clear filters
Jump to...
Review changes
17
contrib/storage-druid/src/main/java/org/apache/drill/exec/store/druid/DruidSubScan.java
REVIEWING
Write
Preview
Leave a comment
Attach files by dragging & dropping, selecting or pasting them.
REVIEW COMMENT
Cancel
Add single comment
Start a review

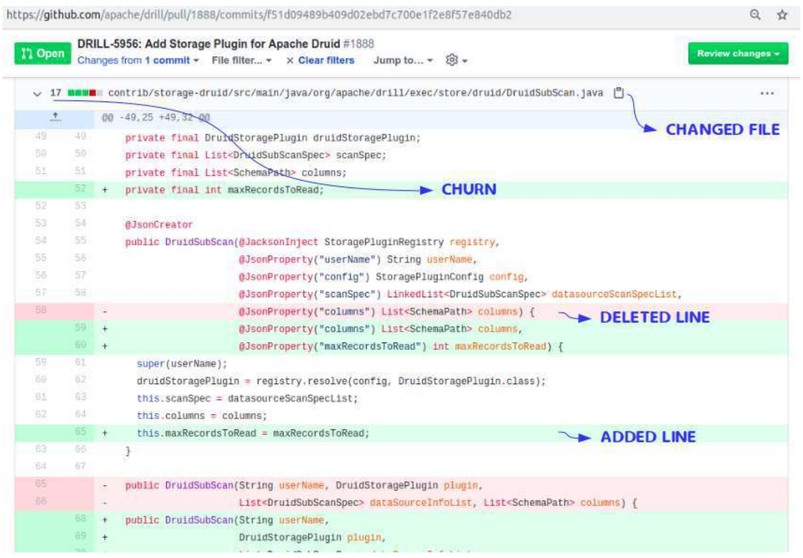
```
      this.columns = columns;
+     this.maxRecordsToRead = maxRecordsToRead;
    }

-   public DruidSubScan(String userName, DruidStoragePlugin plugin,
-                       List<DruidSubScanSpec> dataSourceInfoList, List<SchemaPath> columns) {
+   public DruidSubScan(String userName,
+                       DruidStoragePlugin plugin,
+                       List<DruidSubScanSpec> dataSourceInfoList,
+                       List<SchemaPath> columns,
+                       int maxRecordsToRead) {
      super(userName);
      this.druidStoragePlugin = plugin;
```

Figure 2.6: A partial diff output (reviewing feature) of committed changes into the Apache Drill PR #1888 at GitHub

merge; or *comment*, submitting comments. As a result, the author and other contributors can answer the review through general comments (also called PR comments). In this way, PRs provide a valuable environment for discussion.

After a review, the author can change code to 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 PR, 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* which merges the PR branch into the main branch, through a *merge commit*, chronologically ordered, as depicted in Figure 2.8. Note that the arrows indicate a commit's parent, and the *before* and *after* markers indicate the commits searchable in the PR, respectively, before and after merging;
- *Squash and merge* which squashes all PRs commits into a single commit and merges it into the main branch (Figure 2.9); and
- *Rebase and merge* which re-writes all commits from one branch onto another, by updating their SHA, in a manner that unwanted history can be discarded, as illustrated in Figure 2.10. In this case, commits *0be3d3f* and *66f02d3* received review


https://github.com/apache/drill/pull/1888/commits/f51d09489b409d02ebd7c700e1f2e8f57e840db2
Open
DRILL-5956: Add Storage Plugin for Apache Druid #1888
Changes from 1 commit
File filter...
Clear filters
Jump to...
Review changes
17
CHANGED FILE
CHURN
DELETED LINE
ADDED LINE

```
contrib/storage-druid/src/main/java/org/apache/drill/exec/store/druid/DruidSubScan.java
@@ -49,25 +49,32 @@
    private final DruidStoragePlugin druidStoragePlugin;
    private final List<DruidSubScanSpec> scanSpec;
    private final List<SchemaPath> columns;
+   private final int maxRecordsToRead;

    @JsonCreator
    public DruidSubScan(@JacksonInject StoragePluginRegistry registry,
                        @JsonProperty("userName") String userName,
                        @JsonProperty("config") StoragePluginConfig config,
                        @JsonProperty("scanSpec") LinkedList<DruidSubScanSpec> datasourceScanSpecList,
-                       @JsonProperty("columns") List<SchemaPath> columns) {
+                       @JsonProperty("columns") List<SchemaPath> columns,
+                       @JsonProperty("maxRecordsToRead") int maxRecordsToRead) {
      super(userName);
      druidStoragePlugin = registry.resolve(config, DruidStoragePlugin.class);
      this.scanSpec = datasourceScanSpecList;
      this.columns = columns;
+     this.maxRecordsToRead = maxRecordsToRead;
    }

-   public DruidSubScan(String userName, DruidStoragePlugin plugin,
-                       List<DruidSubScanSpec> dataSourceInfoList, List<SchemaPath> columns) {
+   public DruidSubScan(String userName,
+                       DruidStoragePlugin plugin,
```

Figure 2.7: A partial diff output (differentiating feature) of committed changes into the Apache Drill PR #1888 at GitHub

comments, but they are not accessible via PR. Hence, it is mandatory to recover the original commits when investigating reviewing-related aspects. Nonetheless, such a recovery is not trivial [74].

The first option is useful to deal with high commit volumes 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 [63]. Once merged (status *merged pull request*), the PRs are searchable, that is, supply the search for code changes history, except in case of the ”squash and merge” option.

In summary, PRs are a method for providing fast turnaround and decrease the time to integrate contributions into software development [68]. For this thesis, GitHub PRs denote a contemporary infrastructure from where it is feasible getting real-life review data.

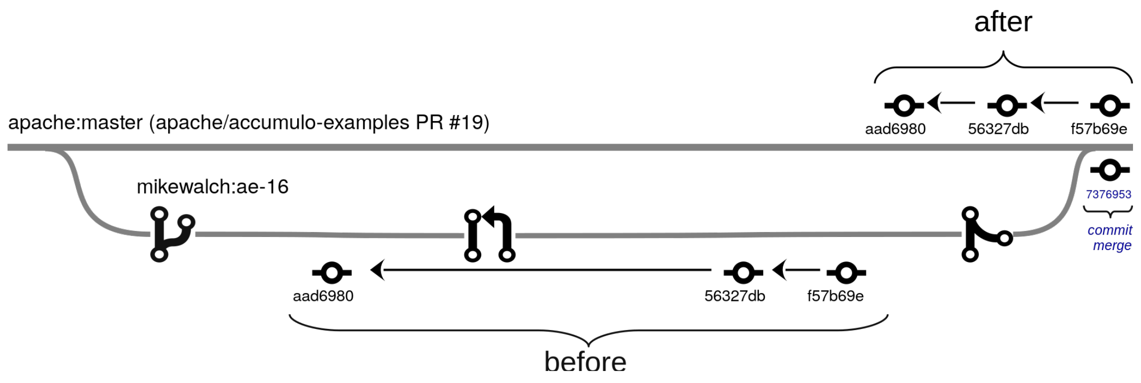


Figure 2.8: Illustrating merge PR option (Apache Accumulo-Examples PR #19)

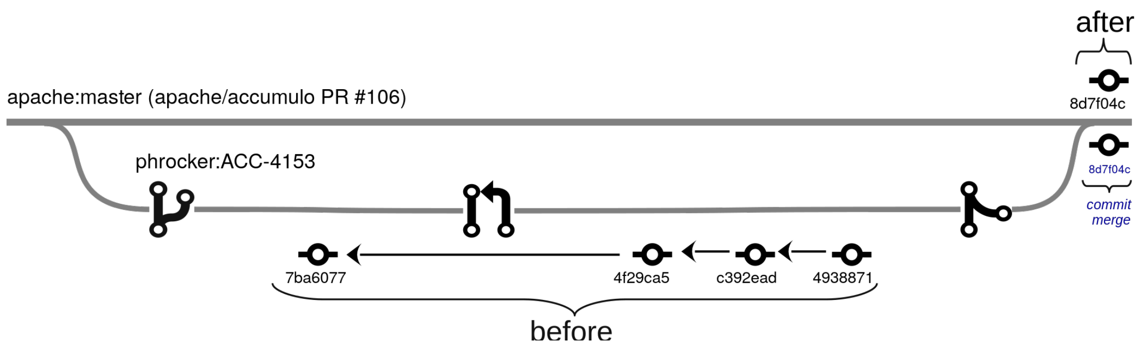


Figure 2.9: Illustrating squash and merge option (Apache Accumulo PR #106)

2.4 Association Rule Learning

Unsupervised learning is an approach for modeling the structure of a dataset by looking for non-obvious patterns without external guidance [43]. Usually, the process follows this workflow [149]:

- 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
- interpreting results, by analyzing the meaningfulness of the algorithm's output aligned with the research context.

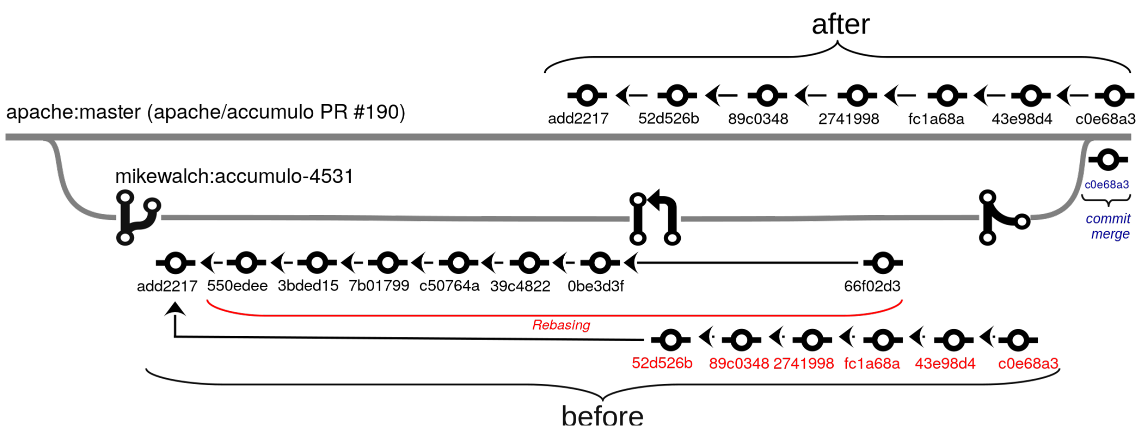


Figure 2.10: Illustrating rebase and merge option (Apache Accumulo PR #190)

Specifically, to the context of this thesis, unsupervised learning through *Association Rule Learning* (ARL) plays a role in the comparison between refactoring-inducing and non-refactoring-inducing PRs detailed in Chapter 3.

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 $I = \{i_1, i_2, ..., i_n\}$, a set of n binary attributes (*items*) and $D = \{t_1, t_2, ..., t_m\}$, 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 $\{X\}, \{Y\} \subseteq I$, $\{X\} \cap \{Y\} = \varnothing$ [25]. In this context, *support* (Equation 2.4) expresses the number of transactions in D that supports an association rule, so expressing its statistical significance.

$$supp(\{X\} \rightarrow \{Y\}) = \frac{frequency(\{X\}, \{Y\})}{n} \tag{2.4}$$

Given that, the following interestingness measures can indicate the strength of an association rule:

- *confidence* (Equation 2.5) means how likely $\{X\}$ and $\{Y\}$ will occur together;

$$conf(\{X\} \rightarrow \{Y\}) = \frac{supp(\{X\} \rightarrow \{Y\})}{supp(\{X\})} \tag{2.5}$$

- *lift* (Equation 2.6) reveals how X and Y are related to one another, as reported in Table 2.2 [66]. Particularly, lift overcomes the case in which the confidence, having a very frequent consequent, will always be high [41].

$$lift(\{X\} \rightarrow \{Y\}) = \frac{supp(\{X\} \rightarrow \{Y\})}{supp(\{X\}) \times supp(\{Y\})} \tag{2.6}$$

- *conviction* (Equation 2.7) provides the identification of the direction of an association rule. It is a measure of implication instead of co-occurrence, ranging 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 [46].

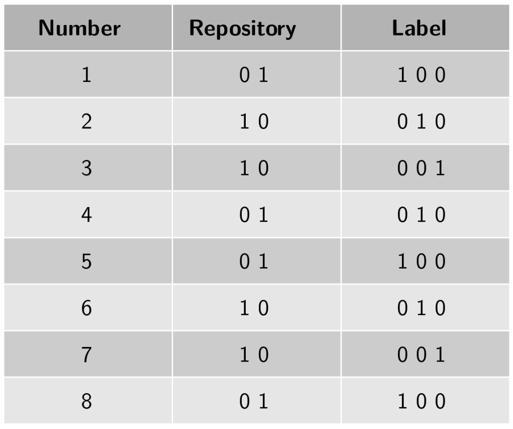
$$conv(\{X\} \rightarrow \{Y\}) = \frac{1 - supp(\{Y\})}{1 - conf(\{X\} \rightarrow \{Y\})} \tag{2.7}$$

Table 2.2: Relationship between antecedent and consequent according to the lift value

Lift	***Meaning***
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 enconding*, 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 [151]. For instance, consider a simple dataset of PRs composed of the following features: each PR 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

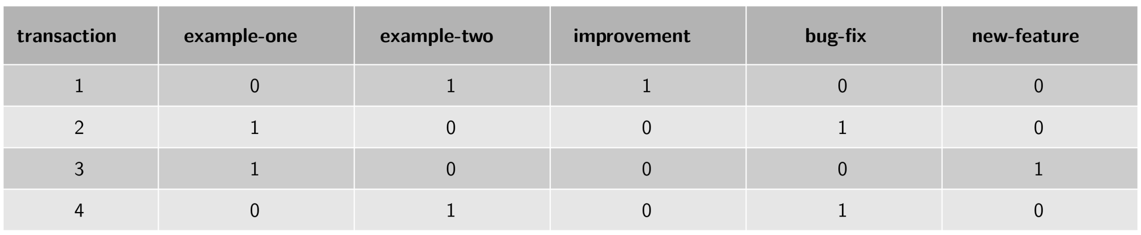
2.11, the dataset comprises eight PR numbers, two distinct repository names, and three labels. Especially, by using hot-encoding, the repository's names are identified by 0 1 (example-one) and 1 0 (example-two), while the labels are recognized by 0 0 1 (new-feature), 0 1 0 (bug-fix), and 1 0 0 (improvement). Thus, the PR number #5, labeled as "*improvement*", belongs to repository example-one.



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 2.11: Example of one-hot encoding

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 2.12, D = {{*example-two, improvement*}, {*example-one, bug-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 2.12: 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) [84]. In Phase 1, **ARL algorithms** require as input the minimum support level for an itemset be considered frequent. Phase 2 output constitutes 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 {*low number of review comments* $\rightarrow$ *low number of reviewers*} denotes that both itemsets {*number of review comments*} and {*number of reviewers*} occur at least 80 times in a dataset of reviewing-related attributes in GitHub PRs, whereas 80% of the PRs 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 [84].

Performance is the main issue surrounding ARL algorithms since Phase 1 requires searching $2^{|I|}$ sets [67]. Traditional algorithms, such as *Apriori* [134], 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* [70] 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 example[10], based on the data from Figures 2.11 and 2.12, that is, I = {*example-one, example-two, improvement, bug-fix, new-feature*}, D = {{*example-two, improvement*}, {*example-one, bug-fix*}, {*example-one, new-feature*}, {*example-two, 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,

[10]Inspired in [140] and `https://t.ly/AbFO`.

so that non-frequent items are dropped. In the example, the frequent itemsets and respective support count are {{*example-two*}, 4}, {{*example-one*}, 4}, {{*improvement*}, 3}, {{*bug-fix*}, 3}, {{*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 goal is to prevent repeated data scans), and mines the frequent itemsets directly from the FP-tree [140]. 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 2.13 illustrates the step-by-step of the FP-tree formation.

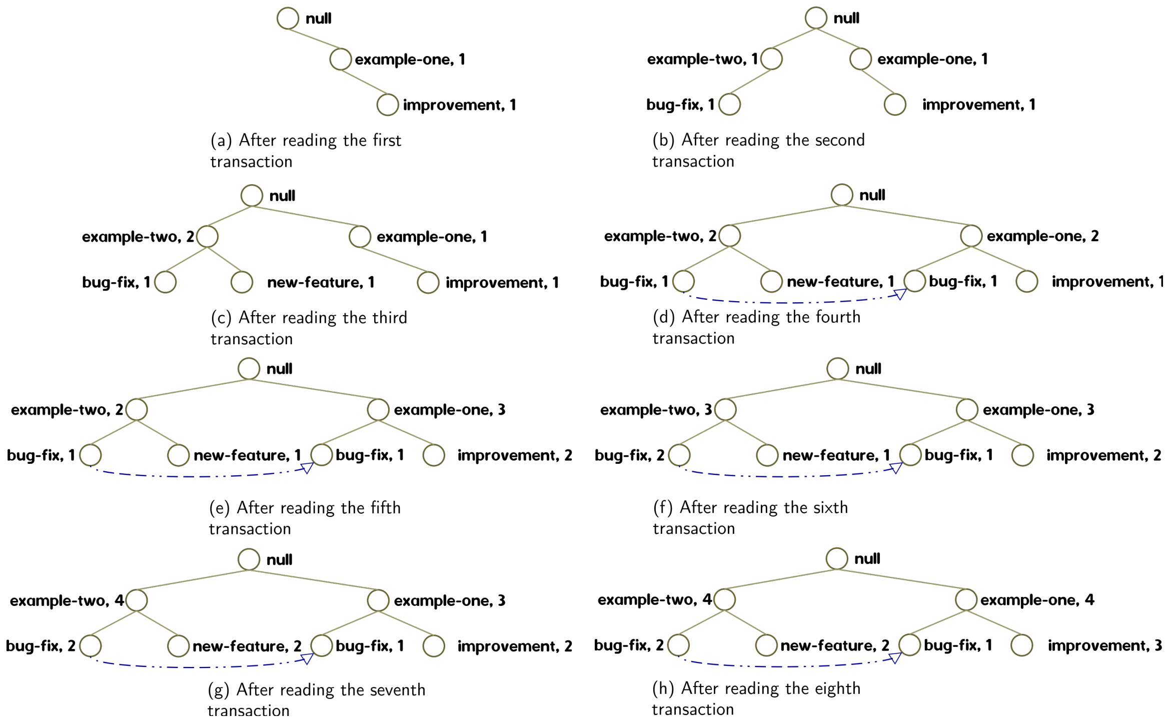


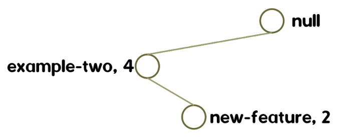
Figure 2.13: Example of the FP-tree formation

Figure 2.13 (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 2.13 (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 2.13 (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 2.13 (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 2.13 (d). The process continues until all transactions are scanned, as demonstrated in Figure 2.13 (e) – (h).

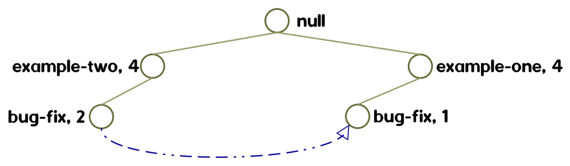
Next, the generation of frequent itemsets is performed by examining the FP-tree in a bottom-up manner. The key idea behind this strategy is to derive frequent itemsets ending with a specific item, by exploring only the branches carrying that item, as illustrated in Figure 2.14. The nodes are visited in descending order. Accordingly, after finding the frequent itemsets ending in {*new-feature*} (Figure 2.14 (a)), the algorithm sequentially search for frequent itemsets ending in {*bug-fix*} (Figure 2.14 (b)), {*improvement*} (Figure 2.14 (c)), {*example-one*} (Figure 2.14 (d)), and {*example-two*} (Figure 2.14 (e)).

It follows that a *conditional pattern base* is computed for each item, as registered in Table 2.3 (second column). A conditional pattern base is a set of frequent items co-occurring with a given item [70]. For instance, when analyzing the item {*new-feature*} (Figure 2.14 (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 2.14 (d), (e)).

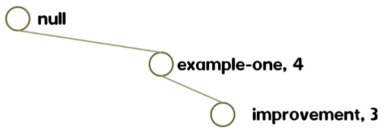
After that, a *conditional FP-tree* is built for each item by (i) considering the set of common items in all branches in the conditional pattern base of that item, and (ii) adding the frequency counts of all items of the branches in the conditional pattern base to compute the support count, as outlined in Table 2.3 (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.



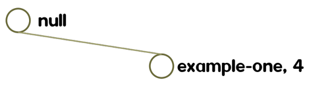
(a) Branches containing node {*new-feature*}



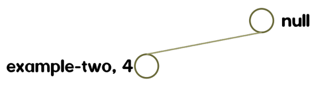
(b) Branches containing node {*bug-fix*}



(c) Branches containing node {improvement}



(d) Branches containing node {example-one}



(e) Branches containing node {example-two}

Figure 2.14: Example of the bottom-up approach for generation of frequent itemsets

Table 2.3: Example of conditional pattern base, conditional FP-tree, and the respective generated frequent pattern rules

Item	***Conditional pattern base***	***Conditional FP-tree***	***Frequent pattern rules***
{*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 2.3 (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, $conf(\{\textit{example-one}\} \rightarrow \{\textit{improvement}\}) = 0.75$ and $conf(\{\textit{improvement}\} \rightarrow \{\textit{example-one}\}) = 1$. Then, by considering the last one association rule, $lift(\{\textit{improvement}\} \rightarrow \{\textit{example-one}\}) = 1.25$ and $conv(\{\textit{improvement}\} \rightarrow \{\textit{example-one}\}) = \infty$. Accordingly, this association rule is useful, since the computed lift is greater than 1, and it is a logical implication because conviction is ∞.

For this thesis, we rely on ARL to investigate similarities and differences between refactoring-inducing and non-refactoring-inducing PRs.

2.5 Concluding Remarks

Refactoring, MCR, Git PRs, and ARL constitute the underlying concepts needed to understand the contextualization of this thesis.

In the next chapter, we describe the mining of refactoring edits and code reviewing-related data that provides the input datasets considered in the subsequent three empirical studies (Chapters 4–6).

Chapter 3

Mining Refactoring Edits and Code Review Data

In this chapter, we explain the procedures for collecting refactoring edits and code reviewing-related data (Subsections 3.1.1–3.1.3), as well as a few countermeasures to attend to validity and reliability issues (Subsection 3.2).

3.1 Data Mining Design

Our data collection consists of three steps: mining of raw PRs data, detection of refactoring edits, and mining of raw code reviewing-related data, as illustrated in Figure 3.1 and detailed in the following subsections.

3.1.1 Mining Merged PRs

We carried out the **mining of raw PRs data** (Step 1) through mining merged PRs from Apache's repositories at GitHub. We investigated only merged PRs because they reveal actions that were in fact finalized; thus, we could explore the refactoring edits performed and the review comments which constitute the actual history of PRs. We chose the GitHub platform due to its popularity [19] and to the mining resources available through public APIs – GitHub *REpresentational State Transfer* (REST) API v3 [5] and GitHub GraphQL API v4 [4]. We used GraphQL as an alternative to the

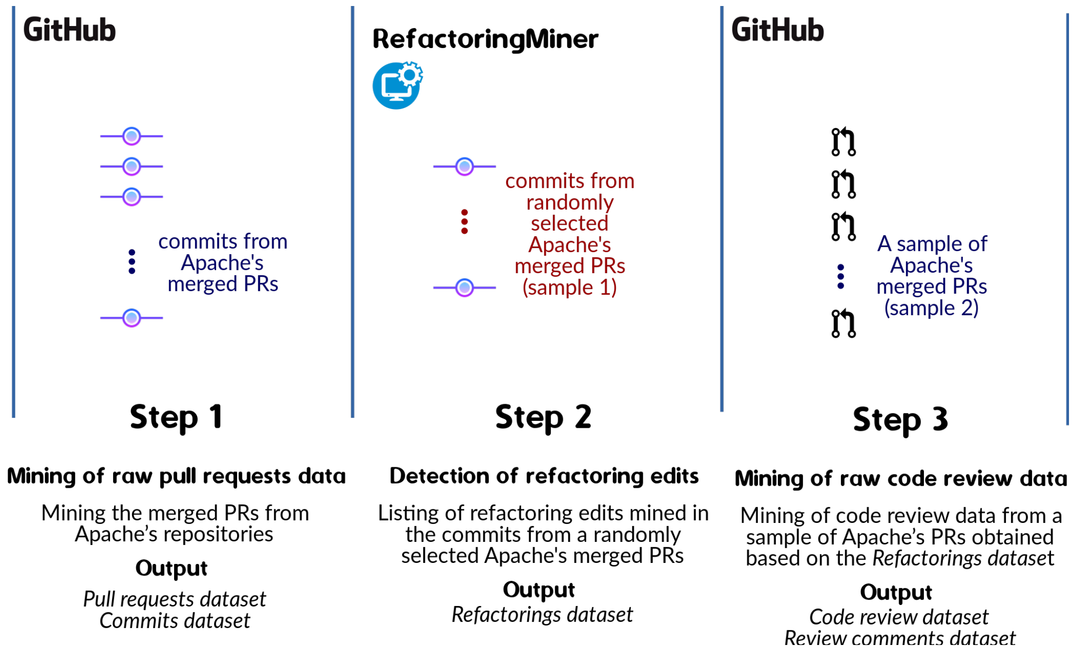

GitHub
commits from Apache's merged PRs
Step 1
Mining of raw pull requests data
Mining the merged PRs from Apache's repositories
Output
Pull requests dataset
Commits dataset
RefactoringMiner
commits from randomly selected Apache's merged PRs (sample 1)
Step 2
Detection of refactoring edits
Listing of refactoring edits mined in the commits from a randomly selected Apache's merged PRs
Output
Refactorings dataset
GitHub
A sample of Apache's merged PRs (sample 2)
Step 3
Mining of raw code review data
Mining of code review data from a sample of Apache's PRs obtained based on the Refactorings dataset
Output
Code review dataset
Review comments dataset

Figure 3.1: Overview of the data mining design

REST API, for instance, when we need to save bandwidth due to its well-structured query strategy [144].

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 [16]. Thus, we selected Apache given its popularity and relevance of contributions in the *Open-Source Software* (OSS) development context. In special, almost 43% of Apache's source code is developed in Java language [20]. Thus, since the version of the refactoring mining tool we selected (Refactoring-Miner, Subsection 3.1.3) only supports code developed in the Java, we considered Java repositories.

It is worth emphasizing that although Apache follows a geographically distributed development as well as other OSS projects [152], the Apache's development process obeys particular principles, named "the Apache way" [17]: 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 (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 inputting[1] the search string *user:apache language:java archived:false*, the results are indicated in Table 3.1. From this, 65,006 merged PRs were detected in 467 from a total of 956 repositories (48.85%).[2] We clarify that such a number denotes only the merged PRs searchable through the GitHub *Graphical User Interface* (GUI); thus, there is no guarantee that it includes the PRs merged through any other mean, such as command line.

Table 3.1: Search results for merged PRs from Apache's non-archived Java repositories in GitHub

Number of repositories	***Number of repositories with merged PRs***	***Number of merged PRs***
956	467	65,006

Then, we mined those merged PRs from August 26, 2019 to September 7, 2019. For that, we implemented a Python script[3], employing resources available at *PyGithub*[4] and *Requests*[5] libraries to use, respectively, GitHub REST API v3 and GraphQL API v4. This mining generated two datasets[6]: *pull requests dataset* and *commits dataset*. The *pull requests dataset* consists of 48,338 PRs, merged by *merge pull request* option (74.4% of the total number of Apache's merged PRs), from 453 distinct repositories (Table 3.2 displays the top 5 ones). The *commits dataset* contains 53,915 recovered commits from 16,668 PRs (25.6% of the total number of Apache's merged PRs), merged

[1]`https://github.com/search`

[2]The list of the number of merged PRs by Apache's repository is available at `https://git.io/J12mV`.

[3]Available at `https://git.io/J12Y0`.

[4]`https://pygithub.readthedocs.io`

[5]`https://requests.readthedocs.io`

[6]Available at `https://git.io/J12OD`.

by *squash and merge* or *rebase and merge* options, from 255 different repositories (Table 3.3 shows the top 5 ones).

Table 3.2: Top 5 repositories containing PRs merged by *merge pull request* option

Repository	***Number of merged PRs***
Incubator-Druid	3,702
Beam	2,838
Pulsar	2,805
Incubator-Pinot	2,673
Ambari	2,588

Table 3.3: Top 5 repositories containing PRs merged by *squash and merge* or *rebase and merge* options

Repository	***Number of merged PRs***
Beam	2,452
Incubator-Pinot	1,184
Incubator-Druid	1,081
Cloudstack	894
Geode	778

When the commits of a PR are squashed into a single commit (*squash and merge* option), in practice, the original commits become not directly accessible when searching the PR. To deal with this issue, we recovered the commits' history of each squashed and merged PR, before any exploration of its original commits, assisted by the *HeadRefForcePushedEvent* object accessible via GitHub GraphQL API [4]. To clarify, consider the Apache's PR #1807 (Figure 3.2) that, originally, had 12 commits ($c_1 - c_{12}$) that were squashed into single commit ($c_{afterCommit}$) after a *force–pushed* event. Consequently, only one commit may be gathered from the PR ($c_{afterCommit}$).

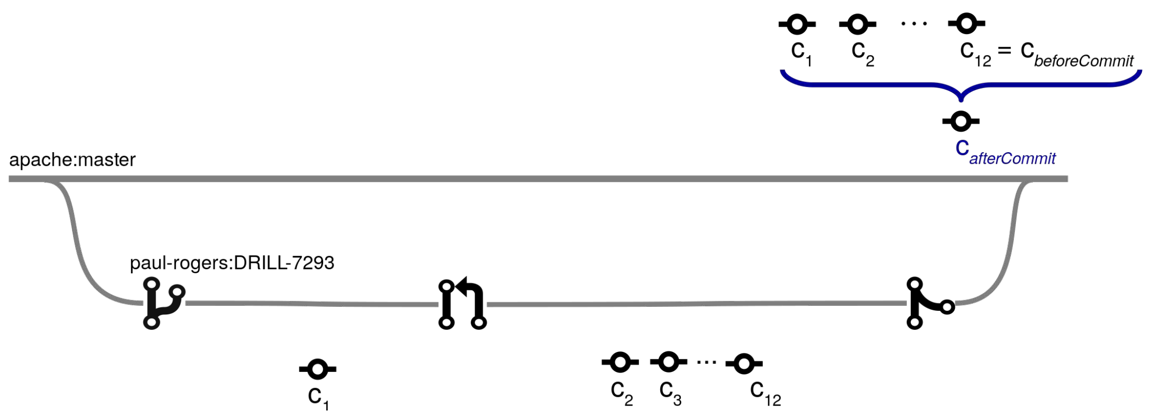


Figure 3.2: An overview of Apache Drill PR #1807, illustrating squashed commits ($c_1 - c_{12}$)

Our recovery strategy follows two steps: (1) we recover the commits $c_{afterCommit}$ and $c_{beforeCommit}$ through the *HeadRefForcePushedEvent* object; and (2) we rebuild the original commits' history by 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. For that, we use the *compare* operation, as available at GitHub REST API v3 [5]. We executed the strategy's Step 1 for gathering the after and before commits from 65,006 PRs, obtaining 53,915 commits after running the strategy's Step 2.

We implemented the recovery strategy of commits from GitHub and built it in the mining script[3] (lines 33–45, 126–130, 163–164). Aiming to assess the effectiveness of our recovery strategy, we followed Apache Drill PR #1807[7] up to the merge, then we (1) ran the RefactoringMiner to detect refactoring edits in the commits from open PRs; (2) executed the recovery strategy after the PRs merge; (3) ran the RefactoringMiner in the recovered commits; and (4) compared the outputs from steps (1) and (3).

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

- repository's *Uniform Resource Locator* (URL), PR number, and commit SHA (*commits dataset*)[8], and

[7]`https://github.com/apache/drill/pull/1807`

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

- repository's URL, and PR number (*pull requests dataset*).[9]

Therefore, Step 1 effected a pre-processing of Apache's merged PRs for detecting refactoring edits (Step 2), in turn, performed in two levels of running: at the *pull request level* for entries from the *pull requests 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 guidelines (by invocating the methods *detectAtCommit()* to commit level and *detectAtPullRequest()* to pull request level).[10]

3.1.2 Detection of Refactoring Edits

RefactoringMiner [11] is a state-of-the-art refactoring detection tool (precision of 97.9% and recall of 87.2%), which has been shown superior to competitive tools, to detect refactoring edits applied to Java source codes that follow pull-based development [146]. We considered RefactoringMiner 1.0 – a version closest to version 2.0 [145], available in September 2019, which supports up to 40 different types of refactoring, including low-level and high-level refactorings, allowing us to work with a comprehensive list of refactoring edits. For this reason, we chose RefactoringMiner 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* refactoring[11], as exposed in Table 3.4. 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.

[9] For example, (*https://www.github.com/apache/dubbo-admin.git, 479*).

[10] `https://github.com/tsantalis/RefactoringMiner#api-usage-guidelines`

[11] From the refactoring detection in Apache Kafka PR #6763; a commit available at `https://git.io/JUlDq`.

Table 3.4: A RefactoringMiner output example

Refactoring type	***Refactoring details***
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.Consumer-Coordinator & moved to class org.apache.kafka.clients. consumer.internals.PartitionAssignor

RefactoringMiner 1.0 detects up to 40 distinct types of refactoring edits (Table 3.5). Given that, we classified the types of refactoring into low-level and high-level categories in the light of their impact on code design, based on technical descriptions provided by Fowler to explain that high-level refactorings are structured in terms of low-level refactorings [60]. Accordingly, creating a class, changing the type of attributes, renaming a method, and moving methods between classes are examples of low-level refactorings. In contrast, high-level refactorings are more complex edits (e.g., creating a hierarchical inheritance and moving and renaming a class between packages) that either impact the code structure (*Extract Superclass*, *Extract Interface*, *Extract Class*, and *Extract Subclass*) or require a higher number of checking in order to preserve the code behavior (*Pull Up Method*, *Pull Up Attribute*, *Push Down Method*, *Push Down Attribute*, *Move Class*, *Move and Rename Class*, and *Move and Rename Attribute*).

Table 3.5: Types of refactoring edits detectable by RefactoringMiner 1.0 in September 2019, classified in line with technical descriptions by Fowler

Low-level refactorings	***High-level refactorings***
Extract Method	Pull Up Method
Inline Method	Pull Up Attribute
Rename Method	Push Down Method
Move Method	Push Down Attribute

Continued on next page

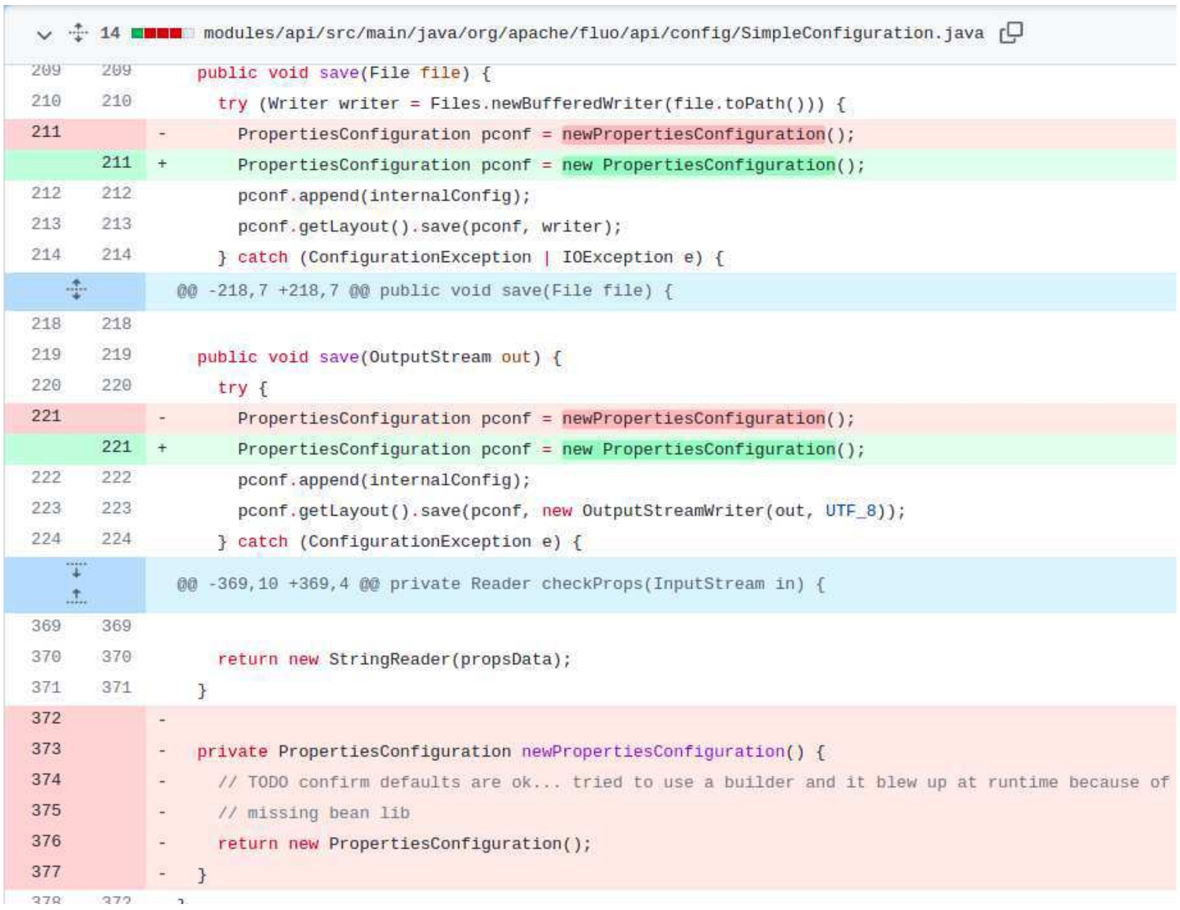
Table 3.5 – continued from previous page

Low-level refactorings	***High-level refactorings***
Move Attribute	Extract Superclass
Rename Class	Extract Interface
Extract and Move Method	Move Class
Rename Package	Move and Rename Class
Extract Variable	Extract Class
Inline Variable	Extract Subclass
Parameterize Variable	Move and Rename Attribute
Rename Variable	
Rename Parameter	
Rename Attribute	
Replace Variable with Attribute	
Replace Attribute (with Attribute)	
Merge Variable	
Merge Parameter	
Merge Attribute	
Split Variable	
Split Parameter	
Split Attribute	
Change Variable Type	
Change Parameter Type	
Change Return Type	
Change Attribute Type	
Extract Attribute	
Move and Rename Method	
Move and Inline Method	

Since we recognize that most of the practitioners and refactoring tool builders employ their experience to define refactoring mechanisms [103], it is worth mentioning that this thesis investigates refactoring in light of the mechanisms specified by RefactoringMiner's developers for each type of refactoring. Aware of challenges related to assurance of behavior preservation [132], we clarify that, to the best of our knowledge, RefactoringMiner does not check for behavior preservation when mining refactoring edits [145;

146]. To illustrate how RefactoringMiner determines refactoring mechanisms, we give a few examples as follows.

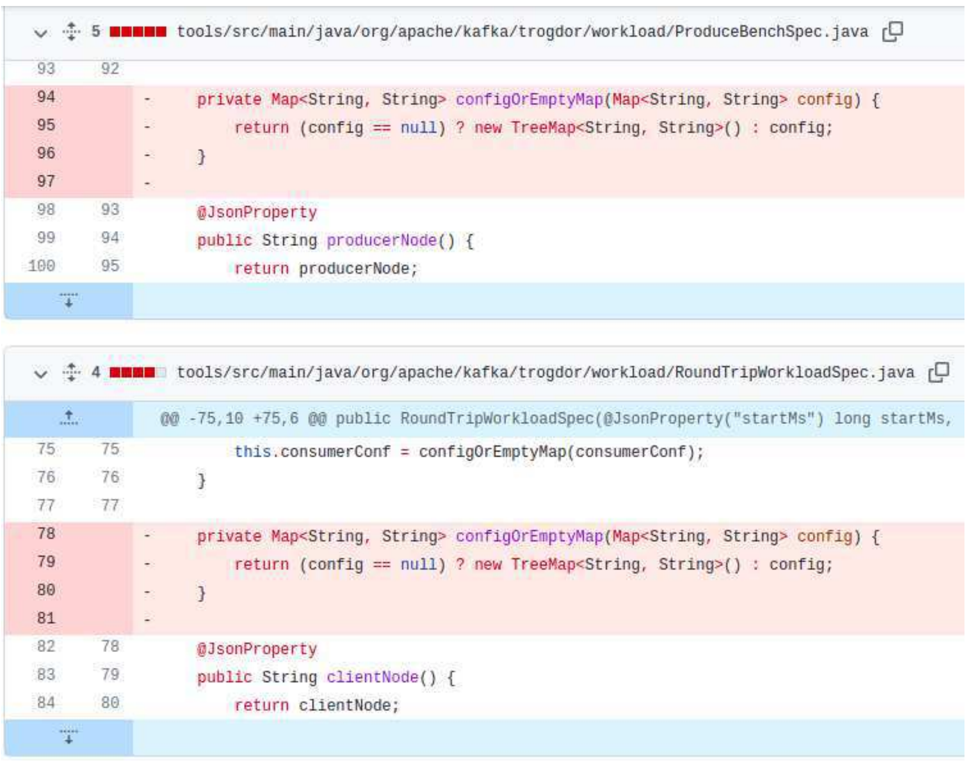
In Figure 3.3, we display a commit that comprises an *Inline Method* (lines 373–376). In this case, RefactoringMiner identifies four occurrences of *Inline Method* at SimpleConfiguration (lines 185, 201, 211, 221).



```
14  modules/api/src/main/java/org/apache/fluo/api/config/SimpleConfiguration.java
209  209      public void save(File file) {
210  210        try (Writer writer = Files.newBufferedWriter(file.toPath())) {
211       -       PropertiesConfiguration pconf = newPropertiesConfiguration();
     211  +       PropertiesConfiguration pconf = new PropertiesConfiguration();
212  212          pconf.append(internalConfig);
213  213          pconf.getLayout().save(pconf, writer);
214  214        } catch (ConfigurationException | IOException e) {
          @@ -218,7 +218,7 @@ public void save(File file) {
218  218
219  219      public void save(OutputStream out) {
220  220        try {
221       -       PropertiesConfiguration pconf = newPropertiesConfiguration();
     221  +       PropertiesConfiguration pconf = new PropertiesConfiguration();
222  222          pconf.append(internalConfig);
223  223          pconf.getLayout().save(pconf, new OutputStreamWriter(out, UTF_8));
224  224        } catch (ConfigurationException e) {
          @@ -369,10 +369,4 @@ private Reader checkProps(InputStream in) {
369  369
370  370        return new StringReader(propsData);
371  371      }
372       -
373       -   private PropertiesConfiguration newPropertiesConfiguration() {
374       -     // TODO confirm defaults are ok... tried to use a builder and it blew up at runtime because of
375       -     // missing bean lib
376       -     return new PropertiesConfiguration();
377       -   }
378  372  }
```

Figure 3.3: A commit, from Apache Fluo PR #1077, consisting of four *Inline Method* instances as detected by RefactoringMiner

When dealing with a *Pull Up Method*, RefactoringMiner identifies the occurrences of such a method in other classes. For instance, it detects two *Pull Up Method* instances of the method configOrEmptyMap (configMap<String, String>), lines 108–110 at TaskSpec, in the commit presented in Figure 3.4: lines 94–96 at ProduceBenchSpec and lines 78–80 at RoundTripWorkloadSpec.



```
5 tools/src/main/java/org/apache/kafka/trogdor/workload/ProduceBenchSpec.java

-    private Map<String, String> configOrEmptyMap(Map<String, String> config) {
-        return (config == null) ? new TreeMap<String, String>() : config;
-    }
-
     @JsonProperty
     public String producerNode() {
         return producerNode;
```

```
4 tools/src/main/java/org/apache/kafka/trogdor/workload/RoundTripWorkloadSpec.java

@@ -75,10 +75,6 @@ public RoundTripWorkloadSpec(@JsonProperty("startMs") long startMs,
         this.consumerConf = configOrEmptyMap(consumerConf);
     }

-    private Map<String, String> configOrEmptyMap(Map<String, String> config) {
-        return (config == null) ? new TreeMap<String, String>() : config;
-    }
-
     @JsonProperty
     public String clientNode() {
         return clientNode;
```

Figure 3.4: A commit, from Apache Kafka PR #4757, consisting of two *Pull Up Method* instances as detected by RefactoringMiner

Regarding an *Extract Class*, RefactoringMiner also detects the occurrences of *Move Attribute* and *Move Method*. To exemplify, it identifies three *Move Attribute* (int MAX_QUERY_RETRIES, Duration INITIAL_BACKOFF, FluentBackoff BACKOFF_FACTORY in lines 48, 51, 54), and one *Move Method* (*getDefaultCredentials()* in lines 59–73) instances at the *extracted class* BigqueryClient from BigqueryMatcher, in the commit shown in Figure 3.5.

In this step, we considered only merged PRs containing two or more commits intending to conform with our refactoring-inducing PR definition. After three weeks of RefactoringMiner running, we obtained a random sample (*sample 1*, Figure 3.1) of 225,127 detected refactorings in 8,761 merged PRs (13.5% of the total number of Apache's merged PRs) from 209 distinct repositories, embracing 68,209 commits, as summarized in Table 3.6. The source of randomness lies in the order in which the repositories were processed.

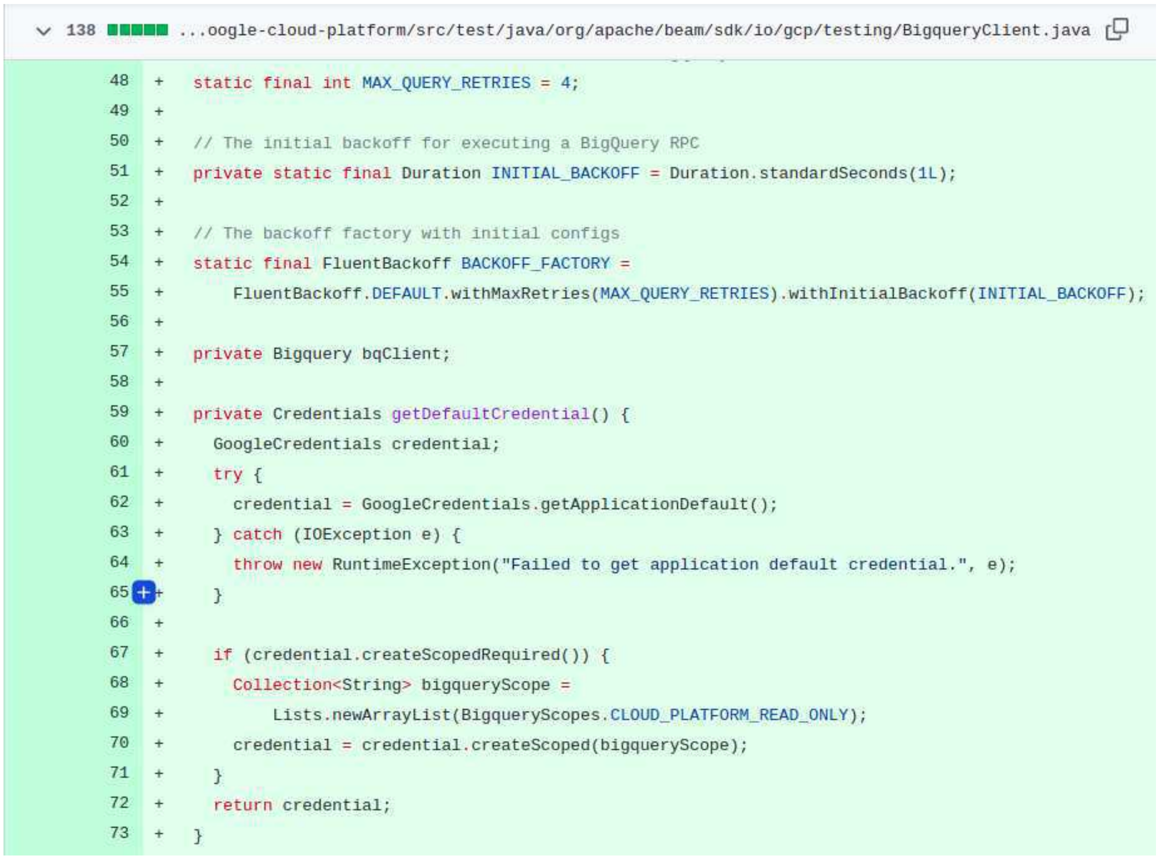

```
138 ...oogle-cloud-platform/src/test/java/org/apache/beam/sdk/io/gcp/testing/BigqueryClient.java
+  static final int MAX_QUERY_RETRIES = 4;
+
+  // The initial backoff for executing a BigQuery RPC
+  private static final Duration INITIAL_BACKOFF = Duration.standardSeconds(1L);
+
+  // The backoff factory with initial configs
+  static final FluentBackoff BACKOFF_FACTORY =
+      FluentBackoff.DEFAULT.withMaxRetries(MAX_QUERY_RETRIES).withInitialBackoff(INITIAL_BACKOFF);
+
+  private Bigquery bqClient;
+
+  private Credentials getDefaultCredential() {
+    GoogleCredentials credential;
+    try {
+      credential = GoogleCredentials.getApplicationDefault();
+    } catch (IOException e) {
+      throw new RuntimeException("Failed to get application default credential.", e);
+    }
+
+    if (credential.createScopedRequired()) {
+      Collection<String> bigqueryScope =
+          Lists.newArrayList(BigqueryScopes.CLOUD_PLATFORM_READ_ONLY);
+      credential = credential.createScoped(bigqueryScope);
+    }
+    return credential;
+  }
```

Figure 3.5: A commit, from Apache Beam PR #6261, consisting of one *Extract Class*, three *Move Attribute* and on *Move Method* instances as detected by RefactoringMiner

At that point, we checked the *commits' authored date* against the *PRs' opening date* in order to identify initial and subsequent commits for the sample's PRs. Therefore, the number of refactoring edits of a PR takes into account only subsequent commits.

We ran RefactoringMiner at the PR level for the *pull requests dataset*, and at the commit level for the *commits dataset*, which results are presented in Table 3.7. Accordingly, the sample is composed of 8,761 PRs, from which 61.5% are PRs merged by *merge pull request* option containing 54.6% of the total number of commits, and 61.6% of the detected refactorings. In specific, 5,391 PRs are from 116 repositories, while 3,370 ones belong to 146 repositories (top 5 repositories are respectively listed Tables 3.8 and 3.9).

Table 3.6: Output of RefactoringMiner execution in Apache's repositories

Number of repositories	***Number of PRs***	***Number of commits***	***Number of detected refactorings***
209	8,761	68,209	225,127

Table 3.7: RefactoringMiner output, at commit and PR levels, for Apache's repositories

Level	***Number of repositories***	***Number of PRs***	***Number of commits***	***Number of detected refactorings***
commit	146	3,370	30,955	86,414
pull request	116	5,391	37,254	138,713

The output dataset[12] after running Step 2 (*sample 1*) comprises the following variables: repository's name, PR number, commit(s) SHA, type(s) of refactoring edits, detail(s) on the refactoring edits, initial (a flag denoting if a commit is initial or subsequent commit), and level (a flag indicating the RefactoringMiner running level).[13]

3.1.3 Mining Code Review Data

Empirical studies have investigated code review efficiency and effectiveness to understand the practice, elaborate recommendations, and develop improvements. Together, these studies share a set of useful code review aspects for further investigation, such as code churn (number of added lines plus number of deleted lines) [38; 40; 80; 120; 121; 143], review comments [40; 68; 87; 109; 94; 107; 120; 112], length of discussion [68; 80; 82; 94; 120; 143], number of changed files [40; 45; 80], number of commits [94;

[12]Available at `https://git.io/J12Gn`.

[13]For 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, False, commit*).

Table 3.8: Top 5 repositories containing PRs merged by *merge pull request* option in sample 1

Repository	***Number of merged PRs***
Kafka	647
Dubbo	611
Beam	604
Tinkerpop	304
Cloudstack	280

Table 3.9: Top 5 repositories containing PRs merged by *squash and merge* or *rebase and merge* options in sample 1

Repository	***Number of merged PRs***
Flink	380
Beam	263
Cloudstack	242
Geode	237
Incubator-Pinot	226

121; 150], number of reviewers [75; 119; 120; 121; 127], and time to merge [68; 73; 82]. From those, we selected the code reviewing-related attributes, listed in (Table 3.10), for the **mining of raw code review data** (Step 3), considering the 8,761 PRs from Step 2 (*sample 2*, Figure 3.1).

The attributes number, title, labels, and repository's name are useful to uniquely identify a PR. We do not count the set of changed files, but the number of times the files changed (i.e., the list of file changes) over subsequent PR commits. Hence, the number of added lines and deleted lines denote the number of lines modified across file changes. Note that length of discussion and time to merge of a PR are respectively derived as follows:

- length of discussion = review comments + non-review comments (either response to the review comments or comments not-related to reviewing), and
- time to merge (in number of days) = merge date − creation date.

Table 3.10: Selected PR attributes for mining

Attribute	***Type***	***Description***
number	continuous	Numerical identifier of a PR
title	categorical	Title of a PR
repository	categorical	Repository's name of a PR
labels	categorical	Labels associated with a PR
commits	continuous	Number of subsequent commits in a PR
additions	continuous	Number of added lines in a PR
deletions	continuous	Number of deleted lines in a PR
changed files	continuous	Number of file changes a PR
creation date	continuous	Date and time of a PR creation
merge date	continuous	Date and time of a PR merge
reviewers	continuous	Number of reviewers in a PR
review comments	continuous	Number of review comments in a PR
non-review comments	continuous	Number of non-review comments in a PR
review comments text	text	The review comments of a PR

For mining the code review related-attributes from GitHub, we developed a Python script[14], employing resources available at the *PyGithub*[15] library to use the GitHub REST API v3 [5]. Specifically, in such step, we imposed one *precondition*: only merged PRs, comprising at least one review comment, should be mined aiming to explore refactoring-inducement and to collect review comments for further investigation. This mining occurred from February 20 to 26, 2020 and it generated two datasets[16], *code*

[14]Available at `https://git.io/J12Zt`.

[15]`https://pygithub.readthedocs.io`

[16]Available at `https://git.io/J12Zo`.

review dataset and *review comments dataset*, refined according to the following procedures:

- dropping of merged PRs with inconsistencies, such as zero file changes[17], and zero reviewers due to problems with their username[18];
- checking for duplicates; and
- mining from non-mirrored repositories[19].

Moveover, we discarded PRs merged by *rebase and merge* option since, in rebasing, some commits within the PR may be due to external changes (outside the scope of the code review sequence), conveying a threat to the validity, as argued in [108]. Accordingly, we considered the number of *HeadRefForcePushedEvent* events and PR commits to identify PRs merged by *squash and merge* option. In specific, PRs merged by *merge pull request* and *squash and merge* options present zero and one *HeadRefForcePushedEvent* event, respectively (squashed and merged PRs keep one commit). We also dropped all PRs containing at least one subsequent commit with two parents, because such commits may represent external changes rebased onto a branch, as depicted in Figure 3.6. Note that, once commit *ee88dea* has two parents, it integrates external changes, which were not reviewed in PR reviewing time. For that, we implemented a script[20] using resources available at *Requests*[21] library.

Then, we realize that even by performing those actions, a few PRs that suffered rebasing still remained. To deal with such an issue, we carried out a manual inspection of 1,845 merged PRs obtained so far, thus identifying rebasing in 206 PRs. Such a procedure consists of manually revisiting each PR commit searching for any signal of

[17]In Guacamole-Client PR #328, and Maven-Plugins PR #18.

[18]In Brooklyn-Server PR #570, Cloudstack PR #2346, Dubbo PRs #1607, #2662, #3326, Flink PRs #4551, #5561, Geode PRs #1017, #1776, Hadoop PR #663, Incubator-Iceberg PR #21, Incubator-Iotdb PR #203, Maven-Jxr PR #6, and Netbeans PR #594.

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

[20]Available at `https://git.io/J12wg`.

[21]`https://requests.readthedocs.io`

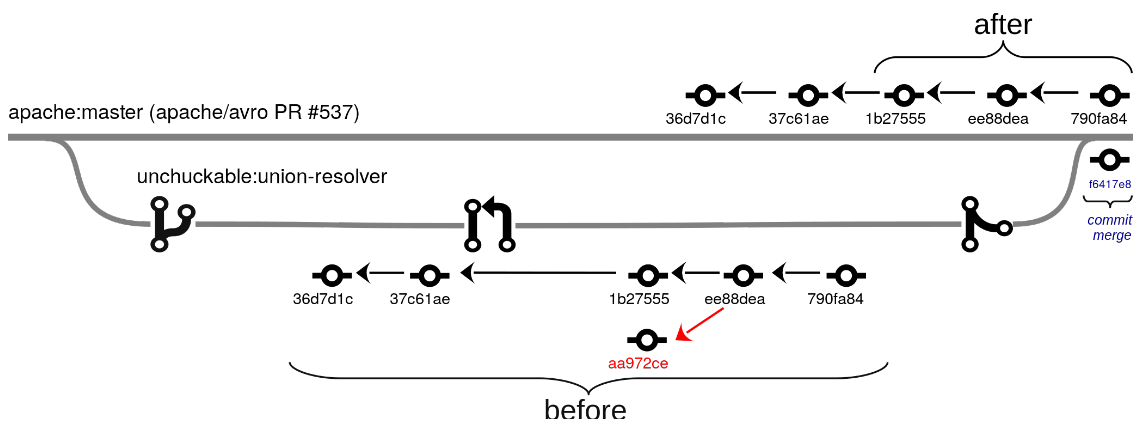


Figure 3.6: Illustrating a PR's commit presenting two parents (Apache Avro PR #537)

rebasing (e.g., a not previously detected commit having two parents) – a task that lasted almost 93 days (about two hours per day).

As a result, our final sample (*sample 2*, Figure 3.1) consists of code review data from 1,639 merged PRs (2.5% of the total number of Apache's merged PRs from Step 1 and 18.7% of the number of sample's PRs obtained from Step 2), encompassing 4,000 subsequent commits, 2,104 detected refactorings, and 10,647 review comments, mined from 73 distinct Apache's repositories (Table 3.11).

Table 3.11: A summary of the final sample from mining Apache merged PRs in GitHub

Number of repositories	***Number of PRs***	***Number of subsequent commits***	***Number of detected refactorings***	***Number of review comments***
73	1,639	4,000	2,104	10,647

Table 3.12 lists the top 5 repositories. The first three repositories are expressive in number of commits concerning the other Apache's repositories, because they together comprise more than 8% of the Apache's commits, in 2019 [20]. The final sample contains 1,580 PRs merged by *merge pull request* option (96.4%), and 59 PRs merged by *squash and merge* option (3.6%).

The final sample includes PRs created from March 18, 2014 to September 02, 2019, and merged from July 18, 2014 to September 05, 2019. As presented in Figure 3.7,

Table 3.12: Top 5 Apache's repositories in the final sample

Repository	***Number of merged PRs***
Kafka	338
Beam	248
Dubbo	145
Servicecomb-Java-Chassis	110
Cloudstack	98

most of these PRs were created and merged in 2018 and 2019, what corroborates with the progressive migration of Apache's projects to GitHub [16].

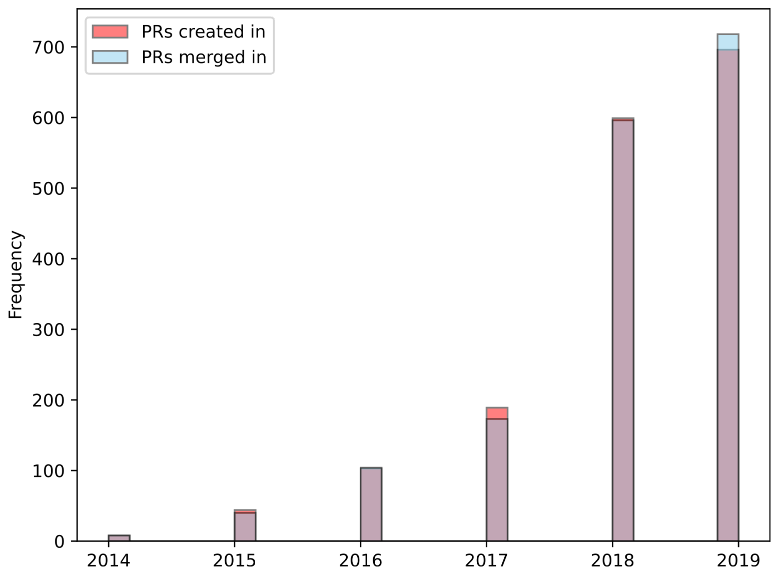


Figure 3.7: Creation and merge dates of the final sample's PRs

The *code review dataset* consists of the following variables by PR: repository's name, number, title, a list of labels, date and time of creation, date and time of merge, number of subsequent commits, number of file changes, number of added lines, number of deleted lines, number of reviewers, number of review comments, number of non-review 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 PRs registered in the *code review dataset*: the name of the repository that comment comes from, the PR 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 hunk[22] 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 PR creation.

3.2 Limitations

We propose the countermeasures listed in Table 3.13 to deal with the issues of **validity and reliability**. To clarify, our data mining design follows well-defined guidelines [124; 125], and was regularly reviewed by the supervisors of this thesis. We make an effort to systematically explain the performed procedures, taken decisions, and obtained results in each design step.

By considering concerns on internal validity, it is worth pointing out that we defined the mining design after conducting two brief case histories in order to get a better understanding of GitHub's PRs and the procedures of data mining and refactoring detection, as reported in Appendix A. Even so, there are risks of threats due to any non-previously identified deficiencies in the procedures of the mining design. Moreover, as we mentioned in Subsection 3.1.1, our mining design is restricted to take on only merged PRs searchable via the GitHub GUI.

We also recognize that the steps and a few procedures within steps could be performed in a distinct sequence, for instance, searching for rebasing in PRs before mining refactoring edits. Nevertheless, our mining design reveals an incremental and practical knowledge regarding how to mine data from GitHub. To exemplify, we performed an

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

Table 3.13: Validity and reliability countermeasures for mining refactoring edits and code reviewing-related data

Type	***Description***
Internal validity	A design proposal to guide the data collection
Construct validity	Regular revision of the mining design by supervisors
	Automatic and manual inspection of PRs to search for rebasing
	Selection of a state-of-the-art tool for refactoring detection purposes
External validity	Randomness and a confidence level of 99.9% and a margin of error of 5% for sampling a representative number of merged PRs
	A few directions on how to apply our mining design to mine data from other pull-based platforms
Reliability	Effort towards clarifying the data collection procedures in order to enable replications

automatic search for rebasing in PRs from our sample (*sample 2*, Figure 3.1), however by manually examining a few PRs, we realized that there were PRs presenting none *force-pushed* event while including commits that have two parents (e.g., Avro #537)[23], thus not detected by our automatic search. Accordingly, we carried out a manual inspection of PRs to identify undetected rebasing in our sample. As a result, we provide an accurate code reviewing-related dataset for further studies and replications.

Despite the efforts towards a more accurate data interpretation, we did not validate the mined refactorings for further studies (since a manual validation of refactoring edits is a time-consuming task [146]), so expressing a potential threat to construct validity. In order to overcome this issue, we selected RefactoringMiner, a state-of-the-art tool (precision of 97.9% and recall of 87.2%), for refactoring detection purposes.

Furthermore, it is noteworthy that, as we argued Subsection 3.1.2, we consider refactoring edits in light of the mechanisms defined by RefactoringMiner's developers – such

[23]More examples and details are available at `https://git.io/J1KB1`.

an issue is due how RefactoringMiner and other developments tools (and developers) define the mechanisms of refactoring [103]. In this context, we shed light on a potentially inflated number and types of refactoring edits, in our sample, due to how RefactoringMiner detects specific refactorings (e.g., an *Extract Class* implies *Move Attribute* and *Move Method* associated instances). We also remember that, to the best of our knowledge, it does not check for behavior preservation when mining refactoring edits [145; 146]. Thus, further studies and replications need to deal with such restriction when using our *refactorings dataset.*

Aiming to ensure a representative set of merged PRs for further research, we consider randomness (obtained through a random order of refactoring detection on the PRs), a confidence level of 99.9%, and a margin of error of 5% at least. Accordingly, regarding 65,006 merged PRs from 467 distinct repositories (Table 3.1), our final sample comprises 1,639 merged PRs, thus achieving a confidence level of 99.9% and a margin of error of 4.01% from 73 distinct repositories (15.6%).

To reduce threats to external validity, we elucidate that is needed customization of our scripts and dataset structures in Steps 1–3 to mine data in other Git-pull-based platforms, such as GitLab [9]. In specific, GitLab provides a GraphQL API for querying attributes related to PRs (coined as *merge requests*) [8]; however, there are a few distinctions, in terms of performance and attribute names, that should be taken into account when reproducing our data mining design. For instance, the complexity of queries is crucial to the response time in GitLab GraphQL API, and the name of attributes are distinct (e.g., *reviewcomments* in GitHub GraphQL API corresponds to *discussions* in GitLab GraphQL API). Even so, we speculate that efforts for such reproduction would be facilitated due to the use of GraphQL resources in both platforms.

Likewise, we clarify that it is not suitable to generalize the conclusions of studies based on our final sample, except when considering other OSS projects that follow a geographically distributed development, as it occurs in GitHub, and are aligned to "the Apache way" principles [17].

To deal with reliability issues, we describe details concerning the procedures and decisions taken in each design step, also provide a reproduction kit to enable replications, publicly available at [22].

3.3 Concluding Remarks

We present a rigorous design for mining refactoring edits and code review data on PRs from Apache's Java repositories in GitHub. In particular, we provide and discuss a few countermeasures to mitigate validity and reliability issues to support further studies and replications.

In the next chapter, we describe an empirical study that addresses a comparison between refactoring-inducing to non-refactoring-inducing PRs, based on the refactoring edits and code reviewing-related aspects mined as from our data mining design.

Chapter 4

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

This chapter presents a comparative investigation between refactoring-inducing and non-refactoring-inducing PRs based on the final sample obtained from mining refactoring edits and code review data in Apache's Java repositories in GitHub (Chapter 3). In the following sections, we detail the research design (Section 4.1), the results, and discuss them (Section 4.2), as well as their implications (Section 4.3) and limitations (Section 4.4).

A preliminary version of this study was published [53]. Since that, we performed the manual inspection of 1,845 merged PRs in order to deal with rebasing (Chapter 3, Subsection 3.1.3). Here, we present the updated version of the study considering 1,639 merged PRs.

4.1 Research Design

We hypothesize that refactoring-inducing PRs have distinct characteristics from non-refactoring-inducing PRs and thus deserve special attention and treatment from researchers, practitioners, and tool builders. Given that, this mixed study (quantitative and qualitative) explores code reviewing-related aspects intending to identify similarities/dissimilarities between refactoring-inducing and non-refactoring-inducing PRs. In this regard, we formulated these **research questions**:

- RQ_1: How common are refactoring-inducing PRs? We firstly explored PRs that met our refactoring-inducing PR definition.
- RQ_2: How do refactoring-inducing PRs compare to non-refactoring-inducing ones? We quantitatively investigated code reviewing-related aspects aiming to find out similarities/dissimilarities in PRs based on the refactoring edits performed.
- RQ_3: Are refactoring edits induced by code reviews? We qualitatively scrutinized a stratified sample of refactoring-inducing PRs to validate the occurrence of refactoring edits induced by code review by manually examining review comments while cross-referencing their detected refactoring edits.

Accordingly, we designed an empirical study that comprises two steps: ARL and data analysis, as depicted in Figure 4.1. We describe the research steps in Subsections 4.1.1 and 4.1.2, and argue a few countermeasures to deal with issues concerning validity and reliability in Section 4.4.

4.1.1 Association Rule Learning

To explore the similarities/dissimilarities between refactoring-inducing and non-refactoring-inducing PRs, we firstly ran ARL (Step 1) because it assists exploratory analysis by automatically identifying the data's inherent structure derived from the relationships between distinct characteristics [43; 49]. Accordingly, by considering ARL on refactoring-inducing and non-refactoring-inducing PRs, we can identify association rules that likely support us in the formulation of more accurate hypotheses concerning similarities/dissimilarities between those two groups.

One may argue that clustering is a better alternative than ARL to find groups of PRs with distinct characteristics. Nonetheless, we experimentally performed clustering in our sample of PRs, after conducting a rigorous selection of clustering algorithm, *Ordering Points To Identify the Clustering Structure (OPTICS)*[31], and input param-

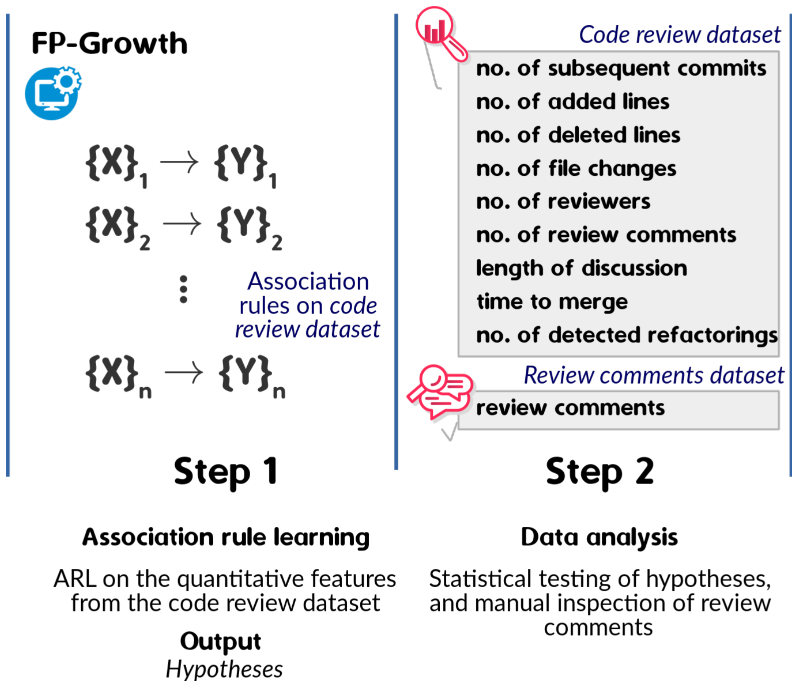

FP-Growth
$\{X\}_1 \rightarrow \{Y\}_1$
$\{X\}_2 \rightarrow \{Y\}_2$
$\{X\}_n \rightarrow \{Y\}_n$
Association rules on code review dataset
Step 1
Association rule learning
ARL on the quantitative features from the code review dataset
Output
Hypotheses
Code review dataset
no. of subsequent commits
no. of added lines
no. of deleted lines
no. of file changes
no. of reviewers
no. of review comments
length of discussion
time to merge
no. of detected refactorings
Review comments dataset
review comments
Step 2
Data analysis
Statistical testing of hypotheses, and manual inspection of review comments

Figure 4.1: An overview of the comparative study between refactoring-inducing and non-refactoring-inducing PRs

eters, *Euclidean distance* [32] as similarity metric, but we found a large noise ratio (76.3%).[1]

The description of ARL meets the following workflow: selecting features (Phase 1), feature engineering (Phase 2), choosing and running an appropriate algorithm (Phase 3), and interpreting results (Phase 4).

Phase 1. Selection of features

We selected all features that can be represented as a number regarding changes, code review, and refactorings, from the *code review dataset* (Chapter 3, Subsection 3.1.3). We considered a three-context perspective (changes, code review, and refactorings) because they together might potentially support the identification of similarities/dissimilarities between refactoring-inducing and non-refactoring-inducing PRs. These are the selected features:

[1]More details are available at `https://git.io/J1i24`.

- number of subsequent commits,
- number of added lines,
- number of deleted lines,
- number of file changes,
- number of reviewers,
- number of review comments,
- length of discussion,
- time to merge, and
- number of detected refactoring edits.

It is arguable that other features could also be considered; however, (i) the PR title is written using natural language, so it is subject to ambiguities; (ii) PR labels are not mandatory, only 312 PRs from our sample have labels (19%); (iii) date and time of creation/merge are specific values, so we used the difference between them (time to merge) for exploration; and (iv) the number of non-review comments of a PR is part of its length of discussion.

Phase 2. Feature engineering

Firstly, we checked the data distribution of the selected features (Appendix B) 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 also due to the presence of outliers. 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 [151]. Specifically, we scaled the raw data into quarters (four equal portions of data), so that the outliers were also taken into account, because in the context of this thesis, outliers do not represent experimental errors; consequently, they can indicate circumstances for further examination. Thus, the *Very high* category includes the outliers.

Then, we applied one-hot encoding based on the quantiles of the features since ARL algorithms typically require categorical data as input [62], resulting in the binning presented in Table 4.1. We chose such technique due to its simplicity and linear time and space complexities [151]. To exemplify, such encoding produces four codes for the number of reviewers in refactoring-inducing PRs in our sample: *n_reviewers_1_to_2*, *n_reviewers_2_to_2*, *n_reviewers_2_to_3*, and *n_reviewers_3_to_6* that correspond to *Low number of reviewers* (1 reviewer), *Medium number of reviewers* (2 reviewers), *High number of reviewers* (3 reviewers), and *Very high number of reviewers* (4–6 reviewers).

Table 4.1: One-hot encoding for binning of features

Category	***Range***
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 the FP-growth algorithm due to its performance, since it needs only two scans of the input data [70]. Then, we developed a Python script[2] for the ARL by using the FP-growth implementation available at the module *mlxtend.frequent_patterns*[3] [118].

We set the minimum support threshold to 0.1 to avoid discarding likely association rules for further analysis [54]. Aiming to get meaningful association rules, we considered minimum thresholds for confidence ≥ 0.5, lift > 1, and conviction > 1. We performed a prior experiment concerning values of minimum support and minimum confidence by taking the thresholds considered in [25] as a reference (support of 0.01, confidence of 0.5). We ran FP-growth considering support values ranging from 0.01 to 0.1 by steps of 0.01, and confidence 0.5 (Table 4.2). In all these settings, we found association rules that cover all input features. Since support is a statistical significance measure, we consider the last setting (minimum support of 0.1, confidence of 0.5) for purposes of

[2]Available at `https://git.io/JMYrj`.

[3]`http://rasbt.github.io/mlxtend/api_subpackages/mlxtend.frequent_patterns/`

FP-growth execution. A lift threshold > 1 reveals useful association rules [41], while a conviction threshold > 1 denotes association rules with logical implications [46].

Table 4.2: ARL output by experimenting minimum support from 0.01 to 0.1 by steps of 0.01, and confidence of 0.5

***Support* ≥**	***Number of association rules***
0.01	58,467
0.02	22,803
0.03	11,431
0.04	7,424
0.05	4,372
0.06	3,006
0.07	1,903
0.08	1,191
0.09	872
0.10	640

Phase 4. Interpretation of Results

We considered feature levels (*None*, *Low*, *Medium*, *High*, and *Very high*), instead of absolute values, as items for composing association rules aiming to identify relative associations among two groups for investigation, for instance, {*high number of review comments*} → {*high number of reviewers*}. In the case of the presence of only three levels to some feature, we assumed the *medium* range to the level containing the median value. We found such a setting for the number of subsequent commits and reviewers in non-refactoring-inducing PRs. To exemplify, the number of subsequent commits is 1 on median in our sample; then, we mapped *Medium* (1), *High* (2), and *Very high* (3 – 19).

The association rules work as basis for the formulation of hypotheses regarding the characterization of our sample's PRs. In this sense, we carried out the following procedure:

1. We pre-processed the association rules, by discarding in line with conviction, in both refactoring-inducing and non-refactoring-inducing PRs. For instance,

considering the following association rules, we dropped the first one due to the conviction values (1.52 < 1.96). In consequence, from 640 initial association rules, we obtained 222.

```
['n_reviewers_3_to_6'] -> ['n_review_comments_12_to_82']
support: 0.1759465478841871
confidence: 0.5163398692810458
lift: 1.9482067336738618
conviction: 1.519593089748992

['n_review_comments_12_to_82'] -> ['n_reviewers_3_to_6']
support: 0.1759465478841871
confidence: 0.6638655462184874
lift: 1.9482067336738615
conviction: 1.9612472160356347
```

2. We mapped the association rules to the corresponding feature levels (*Low*, *Medium*, *High*, and *Very High*). As an example, consider that *n_reviewers_1_to_2*, *n_reviewers_2_to_2*, *n_reviewers_2_to_3*, and *n_reviewers_3_to_6* are mapped respectively to *Low number of reviewers* (1 reviewer), *Medium number of reviewers* (2 reviewers), *High number of reviewers* (3 reviewers), and *Very High number of reviewers* (4–6 reviewers) in all association rules from refactoring-inducing PRs.
3. We dropped identical association rules in both refactoring-inducing and non-refactoring-inducing PRs. With this action, we filtered the more meaningful associations to discover similarities/dissimilarities. As a result, we obtained 191 association rules (54 from refactoring-inducing PRs and 137 from non-refactoring-inducing PRs).
4. We ordered the association rules by conviction in both refactoring-inducing and non-refactoring-inducing PRs.
5. We searched for and analyzed pairwise association rules to assist the rationale for the hypotheses formulation. Accordingly, we found and analyzed eight pairs

of association rules. To exemplify, we analyzed the first following association rule (from refactoring-inducing PRs) in contrast to a pair, that is, the second association rule (from non-refactoring-inducing PRs).

```
['very high reviewers', 'very high file changes'] ->
['very high subsequent commits']
support: 0.10690423162583519
confidence: 0.7868852459016393
lift: 2.2081967213114755
conviction: 3.0202158643138595

['very high reviewers', 'medium file changes'] ->
['medium number of subsequent commits']
support: 0.2848739495798319
confidence: 1.0
lift: 1.8085106382978722
conviction: inf
```

6. We formulated hypotheses, in light of the pairwise association rules, to quantitatively investigate similarities/dissimilarities between refactoring-inducing and non-refactoring-inducing PRs.

4.1.2 Data Analysis

Quantitative data analysis

To answer RQ_1, we analyzed our *code review dataset* (Chapter 3, Subsection 3.1.3) by exploring the detected refactorings by PR. The number of refactorings indicates the edits detected as in the PR *subsequent* commit(s). As a complement, we computed a 95% confidence interval for the percentual (proportion) of refactoring-inducing PRs in Apache's merged PRs, by performing *bootstrap resampling* [58].

We applied statistical testing of hypotheses to answer RQ_2. That analysis encompassed the testing of eight hypotheses formulated from the analysis of the ARL output

(Step 1), driven by a comparison between refactoring-inducing and non-refactoring-inducing PRs. We executed each hypothesis testing in line with this workflow, guided by [48]:

1. Definition of null and alternative hypotheses.
2. Run the statistical test, as follows:
   (a) checking for data normality by using the *Shapiro-Wilk* test;
   (b) checking for homogeneity of variances via *Levene's* test;
   (c) computation of confidence interval for the difference in average or median aligned to output from Steps *(a)* and *(b)*;
   (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 Steps *(a)* and *(b)*. CLES is the probability, at the population level, that a randomly selected observation from a sample will be higher/greater than a randomly selected observation from another sample [93].

   We considered a significance level of 5%, and a substantive significance (*effect size*) for denoting the magnitude of the differences between refactoring-inducing and non-refactoring-inducing PRs at the population level. First, we checked the assumptions for parametric statistical tests (Steps *a* and *b*), since the independence assumption is already met (i.e., a PR is either a refactoring-inducing or not). For exploring the difference between refactoring-inducing and non-refactoring-inducing PRs, we computed a 95% confidence interval by bootstrapping resample according to the output from Steps *a* and *b*, in average or median (Step *c*). Then, we conducted a proper statistical test and calculated the effect size (Step *d*).
3. Deciding if the null hypothesis is supported or refused.

Qualitative data analysis

To answer RQ_3, three developers (intending to mitigate researcher bias) manually examined review comments and validated the detected refactorings from a subset of refactoring-inducing PRs from our sample. We adopted a stratified random sampling to select refactoring-inducing PRs for an in-depth investigation of review comments while cross-referencing detected refactoring edits. Moreover, we validated these refactorings by checking for false positives. As a whole, this analysis lasted 30 days (about four hours per day).

We chose that sampling strategy because it provides a mean to sample non-overlapping subgroups based on specific characteristics [91], (e.g. number of refactorings), where each subgroup (*stratum*) can be sampled using another sampling method – a setting that quite fits to further investigation of categories of refactoring-inducing PRs containing a *Low*, *Medium*, *High*, and *Very high* number of refactorings (Table 4.1). To define the sample size, we considered a confidence level of at least 95% and a margin of error of 4.5%. Accordingly, we analyzed 228 PRs, being 57 refactoring-inducing PRs randomly selected from each category. We split the samples into four categories based on the numbers of refactorings to check if there is a difference in the effect of code review refactoring requests/inducement between PRs with massive refactoring efforts versus PRs with small/focused refactoring efforts.

In the analysis, firstly, we conducted a calibration in which one of the analysts followed up ten analyses performed by the others. Next, each analyst apart examined 40.3%, 38.2%, and 21.5% of the data. In such subjective decision-making, we considered the refactoring-inducement in settings where review comments either explicitly suggested refactoring edits (e.g., *"How about renaming to ...?"*[4]) or left any actionable recommendation that induced refactoring (e.g., *"avoid multiple booleans"* induced a *Merge Parameter* instance[5]).

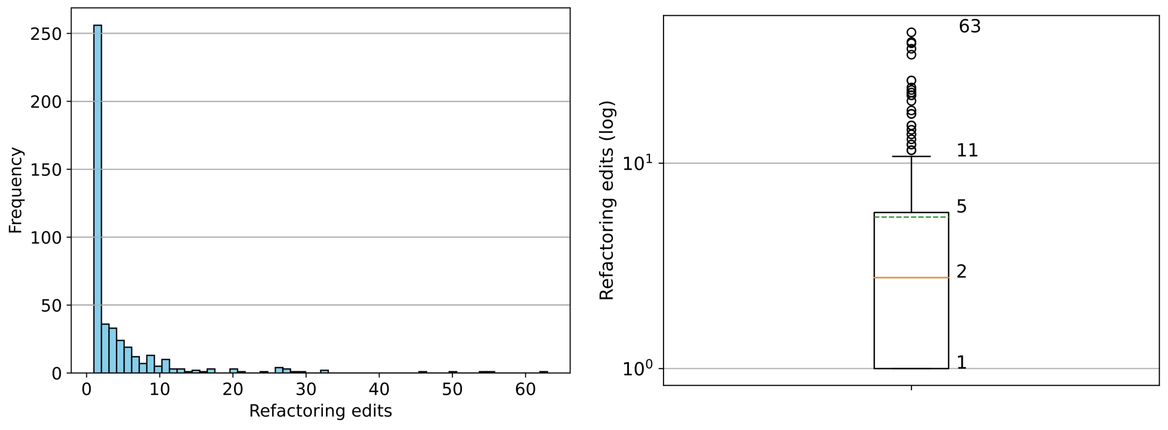
[4]Apache Samza PR #1051, available at `https://git.io/J3z9H`.

[5]Apache Fluo PR #1032, available at `https://git.io/J3mxZ`.

4.2 Results and Discussion

4.2.1 How Common are Refactoring-Inducing PRs?

We found 449/1,639 (27.4%) refactoring-inducing PRs, and 2,104 detected refactoring edits. As shown in Figure 4.2 (a), the histogram of refactoring edits is positively skewed, presenting outliers (denoting refactoring-inducing PRs comprising 11 to 63 refactoring edits detected in the subsequent commits). Thus, a low number of refactoring edits is quite frequent. The number of refactorings by PR is 4.7 on average (SD = 7.4) and 2 as median (IQR = 4), according to Figure 4.2 (b).



(a) Histogram (b) Boxplot

Figure 4.2: Distribution of refactoring edits in the refactoring-inducing PRs

By using bootstrapping resampling and a 95% confidence level, we obtained a confidence interval ranging from 25.3% to 29.6% for the proportion of refactoring-inducing PRs in Apache's merged PRs. This is a motivating result that expresses refactoring as a relevant concern at the PR level, while the presence of outliers can indicate scenarios scientifically relevant for further exploration.

Finding 1: We found 27.4% of refactoring-inducing PRs, which the percentage (proportion) in Apache's merged PRs is in [25.3%, 29.6%], for a 95% confidence level.

This finding corroborates with previous work on refactorings in practice. Murphy-Hill et al. identified refactorings in 10% of commits of the Eclipse project [100], and Brito et al. found that most of the refactorings occur in two or three commits when analyzing ten popular OSS projects in GitHub [47]. These findings together may explain the proportion of refactoring-inducing PRs in our sample (27.4%) since they have 3.5 on average (SD = 3.1) and three on median (IQR = 2) subsequent commits. Moreover, a refactoring-inducing PR presents at least one refactoring edit in one of its subsequent commit(s). It is noteworthy that those works did not investigate commits at the PR level nor distinguished PRs in light of our refactoring-inducing PR definition. Therefore, this study may advance the practical understanding of refactoring and code review at the PR level.

4.2.2 How Do Refactoring-Inducing PRs Compare to non-Refactoring-Inducing Ones?

From ARL, we analyzed eight pairwise association rules, four from refactoring-inducing PRs (AR_1–AR_4) and four from non-refactoring-inducing PRs (AR_5–AR_8), all catalogued in Table 4.3 in decreasing order of conviction.

Table 4.3: Association rules selected by manual inspection (AR_1–AR_4 for refactoring-inducing PRs, AR_5–AR_8 for non-refactoring-inducing PRs)

Id	***Association rule***	***Supp***	***Conf***	***Lift***	***Conv***
AR_1	{Very high no. of refactorings, Very high no. of added lines, Very high no. of subsequent commits} → {Very high no. of file changes}	0.11	0.96	3.42	17.98
AR_2	{Very high no. of refactorings, Very high no. of deleted lines, Very high no. of subsequent commits} → {Very high no. of file changes}	0.10	0.94	3.34	11.75
AR_3	{Very high length of discussion, Very high no. of reviewers} → {Very high no. of review comments}	0.13	0.82	3.11	4.18

Continued on next page

Table 4.3 – continued from previous page

Id	***Association rule***	***Supp***	***Conf***	***Lift***	***Conv***
AR_4	{Medium time to merge} → {High no. of reviewers}	0.15	0.51	1.08	1.08
AR_5	{Medium no. of file changes, Low no. of added lines} → {Low no. of subsequent commits}	0.17	1.0	1.81	∞
AR_6	{Medium no. of file changes, Low no. of deleted lines} → {Medium no. of subsequent commits}	0.17	1.0	1.81	∞
AR_7	{High no. of review comments, High length of discussion} → {Very high no. of reviewers}	0.10	0.92	1.21	3.11
AR_8	{High no. of review comments, Medium time to merge} → {Very high no. of reviewers}	0.11	0.89	1.16	2.11

Afterwards, we formulated eight hypotheses on the similarities/dissimilarities between refactoring-inducing and non-refactoring-inducing PRs, discussed as follows. As a complement, Table 4.4 shows a few descriptive statistics of the examined attributes from refactoring-inducing and non-refactoring-inducing PRs. We present the details regarding the checking for parametric tests assumptions and statistical testing of hypotheses in Appendix C.

Table 4.4: Descriptive statistics of PR attributes

Attribute	***Average***	***SD***	***Median***	***IQR***
refactoring-inducing PRs				
No. of added lines	316	2,633.7	50	114
No. of deleted lines	100.2	323.5	30	71
No of file changes	14.7	89.3	5	7
No. of subsequent commits	3.5	3.1	3	2
No. of review comments	9.5	106	6	9
Length of discussion	14.8	12.9	11	13
No. of reviewers	2.3	0.9	2	1
Time to merge (in days)	12.5	47.1	4	10
refactoring-inducing PRs				

Continued on next page

Table 4.4 – continued from previous page

Attribute	***Average***	***SD***	***Median***	***IQR***
No. of added lines	53.7	528	8	26
No. of deleted lines	35.2	258	6	15.7
No of file changes	5	49.1	2	3
No. of subsequent commits	2	1.9	1	1
No. of review comments	5.3	8	3	4
Length of discussion	9.6	10.4	7	8
No. of reviewers	2	0.8	2	0
Time to merge (in days)	8.5	32.3	2	7

H_1 Refactoring-inducing PRs are more likely to have more added lines than non-refactoring-inducing PRs

Refers to association rules: (AR_1,AR_5).

H_2 Refactoring-inducing PRs are more likely to have more deleted lines than non-refactoring-inducing PRs

Refers to association rules: (AR_2,AR_6).

Rationale: Together, AR_1 and AR_2 in comparison to AR_5 and AR_6 suggest that refactoring-inducing PRs comprise more code churn (number of added lines + number of deleted lines) than non-refactoring-inducing PRs.

Finding 2: Refactoring-inducing PRs comprise significantly more code churn than non-refactoring-inducing PRs, since refactoring-inducing PRs are significantly more likely to present a higher number of added lines ($U = 0.42 \times e^{+06}$, $p < .05$), CLES = 78.9% and deleted lines ($U = 0.42 \times e^{+06}$, $p < .05$), CLES = 77.9% than non-refactoring-inducing PRs.

This is an expected result in light of the findings from Hegedüs et al., since refactored code has significantly higher size-related measurements [71]. Thus, we speculate that reviewing larger code churn may potentially promote refactorings, supported by Rigby et al., who observed that the magnitude of code churn influences code reviewing [121; 122], and Beller et al. who discovered that the larger the churn, the more changes could follow [40].

H_3 Refactoring-inducing PRs are more likely to have more file changes than non-refactoring-inducing PRs

Refers to association rules: (AR_1,AR_5), (AR_2,AR_6).

Rationale: Together, AR_1 and AR_2 in comparison to AR_5 and AR_6 denote that refactoring-inducing PRs consist of more file changes than non-refactoring-inducing PRs.

Finding 3: Refactoring-inducing PRs encompass significantly more file changes than non-refactoring-inducing PRs (U = $0.41 \times e^{+06}$, $p < .05$), CLES = 76.1%.

We conjecture that reviewing code across files may motivate refactorings, an argument supported by Beller et al. regarding more file changes comprise more changes during code review [40]. By observing change-related aspects (churn and file changes), our findings confirm previous conclusions on the influence of the amount and magnitude of changes on code review [38; 80; 121; 122]. When analyzing the changes and refactorings, our findings reinforce prior conclusions on refactored code significantly present higher size-related measurements (e.g., number of code lines and file changes) [71], and larger changes promote refactorings [107].

We can not claim that a higher code churn and the number of file changes result from refactoring edits performed over the subsequent commits in refactoring-inducing PRs (Findings 2–3). We believe that a qualitative study could provide more precise arguments regarding the causes (perhaps, floss refactoring).

H_4 Refactoring-inducing PRs are more likely to have more subsequent commits than non-refactoring-inducing PRs

Refers to association rules: (AR_1,AR_5), (AR_2,AR_6).

Rationale: Together, AR_1 and AR_2 in comparison to AR_5 and AR_6 indicate that refactoring-inducing PRs tend to include a higher number of subsequent commits than non-refactoring-inducing PRs.

Finding 4: Refactoring-inducing PRs comprise significantly more subsequent commits than non-refactoring-inducing PRs (U = $0.37 \times e^{+06}$, $p < .05$), CLES = 69.1%.

Based on our previous findings on the magnitude of code churn and file changes, 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 [40]. Accordingly, we speculate that reviewing refactoring-inducing PRs might require more subsequent changes, in turn, denoted by more subsequent commits in comparison with non-refactoring-inducing PRs.

H_5 Refactoring-inducing PRs are more likely to have more review comments than non-refactoring-inducing PRs

Refers to association rules: (AR_3,AR_7/AR_8).

Rationale: When considering the same number of reviewers in AR_3 contrasting to AR_7 and AR_8, those association rules propose that code review in refactoring-inducing PRs tend to encompass a higher number of review comments than in non-refactoring-inducing PRs.

Finding 5: Refactoring-inducing PRs embrace significantly more review comments than non-refactoring-inducing PRs (U = $0.35 \times e^{+06}$, $p < .05$), CLES = 65.6%.

Beller et al. found that review comments drive the most changes during code review [40], and Pantiuchina et al. discovered that discussions among developers motivate almost 35% of refactoring edits, in OSS projects at GitHub [112]. Thus, we conjecture that, besides change-related aspects, GitHub's PR model can constitute a peculiar structure for code review, in which review comments influence the occurrence of refactorings, therefore explaining our result. This argument originates from the fact that a pull-based collaboration workflow provides reviewing resources [7] (e.g., a proper code reviewing GUI) for developers to improve/fix the code while having access to the history of commits and discussion. Our finding also provides insight for examination of review comments to get an in-depth understanding of refactoring-inducement.

H_6 Refactoring-inducing PRs are more likely to present a lengthier discussion than non-refactoring-inducing PRs

Refers to association rules: (AR_3,AR_7/AR_8).

Rationale: Regarding the same number of reviewers in AR_3 contrasting to AR_7 and AR_8, those association rules suggest that code review in refactoring-inducing PRs tends to contain a lengthier discussion than in non-refactoring-inducing PRs.

Finding 6: Refactoring-inducing PRs enclose significantly more discussion than non-refactoring-inducing PRs (U = $0.35 \times e^{+06}$, $p < .05$), CLES = 65.3%.

A more in-depth analysis could tell how profound these lengthier discussions are, although a higher number of comments might represent developers concerned with the code, willing then to extend their collaboration to the suggestion of refactorings. Previous findings may support those claims; Lee and Cole, when studying the Linux kernel development, acknowledged that the amount of discussion is a quality indicator [86]. Also, empirical evidence reports on the impact of the number of comments on changes [40; 112].

H_7 No significant distinction regarding the number of reviewers

Refers to association rules: (AR_3,AR_7).

Rationale: When contrasting AR_3 to AR_7, those association rules suggest no difference between refactoring-inducing and non-refactoring-inducing PRs as for the number of reviewers.

Finding 7: We found no statistical evidence that the number of reviewers is related to refactoring-inducement (U = $0.30 \times e^{+06}$, $p < .05$), CLES = 56.7%.

Refactoring-inducing and non-refactoring-inducing PRs present two reviewers as median – the same result found by Rigby et al. [119] in the OSS scenario. There are outliers that, in turn, could be justified by other technical factors, such as complexity of changes, as argued in [120]. However, our study does not address that scope.

H_8 Refactoring-inducing PRs are more likely to take a longer time to merge than non-refactoring-inducing PRs

Refers to association rules: (AR_4,AR_8).

Rationale: Regarding a gradual increase in the number of reviewers in AR_4 and contrasting it to AR_8 (Finding 7), those association rules propose that refactoring-inducing PRs take more time to merge than non-refactoring-inducing PRs.

Finding 8: Refactoring-inducing PRs take significantly more time to merge than non-refactoring-inducing PRs ($U = 0.31 \times e^{+06}$, $p < .05$), CLES = 57.4%.

We realize the influence of refactorings on time to merge, concluding that time for reviewing and performing refactoring edits both impact the time to merge. In special, this conclusion is aligned to Szoke et al., who observed a correlation between implementing refactorings and time [139], and from Gousios et al., who found that review comments and discussion affect time to merge a PR [68]. As argued by Kononenko et al., size-related factors influence the reviewing time [80; 82]. Accordingly, we also consider the impact of greater code churn and changed files, as it occurs in refactoring-inducing PRs, on time to merge.

4.2.3 Is Refactoring Induced by Code Reviews?

To answer RQ_3, we sampled 228 refactoring-inducing PRs[6], 57 ones from each of the *Low*, *Medium*, *High*, and *Very High* categories encompassing one, two, three to five, and six to 63 refactoring edits, respectively. As a result of the manual validation of refactoring edits, RefactoringMiner 1.0 obtained a precision of 98.2% and a recall of 99.7%, as detailed in Table 4.5.

[6]Available at `https://git.io/JMYo8`.

Table 4.5: Results of the manual validation of refactoring edits mined by RefactoringMiner 1.0

TP	***FP***	***FN***	***Precision (%)***	***Recall (%)***
1,886	35	5	98.2	99.7

By examining 2,096 review comments and 1,207 discussion comments in the sampled PRs, we found 133/228 (58.3%) in which at least one refactoring edit was induced by review comments. Such PRs comprise 815 subsequent commits, and 1,891 detected refactorings, 545 of which were induced by review comments. As shown in Table 4.6, most of the refactoring edits are low-level (95.9%), regardless of the category of refactoring-inducing PRs. We speculate that such a high proportion is due to how RefactoringMiner defines refactoring mechanisms, as explained in Chapter 2, Subsection 2.1.3. We found 223/545 (40.9%) *Rename* edits (being *readability* a common motivation cited by reviewers) and 160/545 (29.3%) *Change Type* edits as the most induced by review in our stratified sample.

Table 4.6: Refactoring-inducing PRs, in which refactoring edits were induced by code review, by level of refactoring.

Category	***Low-level refactorings***	***High-level refactorings***	***Proportion***
Low	34/35 (97.1%)	1/35 (2.9%)	35/57 (61.4%)
Medium	61/61 (100.0%)	none (0.0%)	61/127 (48.0%)
High	134/136 (98.5%)	2/136 (1.5%)	136/285 (47.7%)
Very high	294/313 (93.9%)	19/313 (6.1%)	313/1,422 (22.0%)
All categories	*523/545 (95.9%)*	*22/545 (4.1%)*	*545/1,891 (28.8%)*

Finding 9: In a stratified sample of 228 refactoring-inducing PRs, 133 ones (58.3%) present at least one refactoring edit induced by code review.

Our finding indicates the influence that code review has at the PR level, thus corroborating with Paixão et al., who found that refactorings' motivations may emerge from code review [109], and Pantiuchina et al., who analyzed discussion in merged PRs (containing at least one refactoring in one of their commits) and found refactorings also triggered from discussion [112]. However, our study differs from those previous ones because we distinguished refactoring-inducing from non-refactoring-inducing PRs by exploring reviewing-related aspects and refactoring-inducement (that is, considering PR subsequent commits).

4.3 Implications

By distinguishing refactoring-inducing from non-refactoring-inducing PRs, we can potentially advance the understanding of code reviewing at the PR level and assist researchers, practitioners, and tool builders in this context.

Researchers: No prior MCR studies made a distinction between refactoring-inducing and non-refactoring-inducing PRs when analyzing their research questions, which might have affected their findings or discussions. For instance, by also regarding refactoring-inducing PRs, Gousios et al. [68] and Kononenko et al. [82] could have found different factors influencing the time to merge a PR; Pascarella et al. [113] could have identified further information to perform a proper code review in presence of refactorings; whereas, Pantiuchina et al. [112] could have different conclusions on the motivations for refactorings in PRs, as they analyzed PRs with refactorings detected even in the initial commit (i.e., these refactorings were not induced from review comments). Our findings, except for Findings 1, 7 and 9, indicate that refactoring-inducing and non-refactoring-inducing PRs have different characteristics. Therefore, we recommend that future experiment designs on MCR with PRs to make a distinction between refactoring-inducing and non-refactoring-inducing PRs, or consider their different characteristics when sampling PRs. As the motivating findings from Aniche et al.[30], when exploring supervised *Machine Learning* (ML) to recognize relevant refactoring opportunities in OSS projects, we speculate that researchers can use supervised ML in order to identify refactoring-inducing and non-refactoring-inducing PRs based on our findings. Researchers can also use our mined data, developed tools, and research methods, publicly available [22], to investigate code reviewing in pull-based development.

Practitioners: Our findings indicate that there is no statistical difference in the number of reviewers between refactoring-inducing and non-refactoring-inducing PRs (Finding 7). But, all other findings show that refactoring-inducing PRs are associated with more code churn (Finding 2), more file changes (Finding 3), more subsequent commits (Finding 4), more review comments (Finding 5), lengthier discussions (Finding 6), and more time to merge (Finding 8) than non-refactoring-inducing PRs. Thus, we suggest to project managers to invite more reviewers when a PR becomes refactoring-inducing, to share the expected increase in review workload, and, perhaps more importantly, to share the knowledge of design changes caused by subsequent refactoring edits to more team members.

Tool builders: In connection to our implication for practitioners, tool builders can develop bots [136; 85] that recommend reviewers based on some criteria [97] when a PR becomes refactoring-inducing, to assist the project managers in inviting additional reviewers. Our findings indicate that refactoring-inducing PRs have higher complexity in code churn (Finding 2) and file changes (Finding 3). Therefore, it is necessary to help developers distinguish refactoring edits from non-refactoring edits directly in the review board of pull-based development platforms, such as GitHub and Gerrit, where the reviews are actually taking place. In the past, researchers implemented refactoring-awareness in the code diff mechanism of IDEs [64; 28; 65]. Even though not directly related to our results, we believe that adding refactoring-awareness directly in the GitHub or Gerrit review board – such as the refactoring-aware commit review Chrome browser extension [90] – would allow reviewers to trace the refactorings performed throughout the commits of a PR, provide prompt feedback, and concentrate efforts on other aspects of the changes, such as collateral effects of refactorings and proposing specific tests. This recommendation is in agreement with Gousios et al. [69], who emphasized the need for untangling code changes and supporting change impact analysis directly in the PR interface.

4.4 Limitations

Table 4.7 lists the procedures proposed to deal with the issues of **validity and reliability** regarding data interpretation. It is worth emphasizing that, since we consider data obtained in line with our data mining design, this study relies on its countermeasures and threats to validity and reliability (Chapter 3, Section 3.2). Nevertheless, we propose a few actions intending to increase the validity of the study, methods triangulation [125]. In order to mitigate researcher bias, the manual examination of review comments was conducted by three developers, after an initial calibration. Despite our efforts to perform an initial calibration, there may be limitations concerning conclusions, since we carried out apart analyses. We persist in making an effort to systematically explain the performed procedures, taken decisions, and the obtained results in each design step.

Table 4.7: Validity and reliability countermeasures for comparing refactoring-inducing and non-refactoring-inducing PRs

Type	***Description***
Internal validity	A design proposal to guide the study
Construct validity	Regular revision of the mining design by supervisors
	Establishment of a chain of evidence based on the methods triangulation (quantitative and qualitative data analysis)
	Randomness and a confidence level of 95% and a margin of error of 4.5% for sampling a representative number of refactoring-inducing PRs for the qualitative analysis
Reliability	Effort towards clarifying the data processing and data analysis procedures to enable replications

Firstly, we propose methods triangulation (statistical testing of hypotheses and a manual analysis of a stratified sample of refactoring-inducing PRs) intending to reduce deficiencies from any single method and enhance our study's validity [124].

It is noteworthy that we defined the study design after a pre-running of the clustering and ARL processes. However, for clustering, we obtained a noise of 76.3%. We carried out a rigorous selection of algorithms and input parameters for ARL and we carefully defined workflows for our research design procedures aiming to explain the decisions taken. Even so, there might be other risks of threats to internal validity due to any non-previously identified deficiencies in our research design.

Although we established a chain of evidence for the data interpretation and description of the study design, we did not validate the detected refactorings before data analysis, so expressing a potential threat to construct validity (RQ_1 and RQ_2), since this study relies on our data mining design (Chapter 3, Section 3.2). When addressing RQ_3, we validated all detected refactorings in our stratified sample, so identifying 1,886 true positives, 35 false positives, and five false negatives.

Furthermore, as already admitted in the refactoring-inducing PR definition, we cannot claim that all refactoring edits were caused by reviewing. To deal with such limitation, we carried out a qualitative analysis of review comments from 228 randomly selected refactoring-inducing PRs, considering a sample size meeting a confidence level of 99.9% and a margin of error of 4.5%. Thus, this empirical study provides a particular motivation for a further qualitative investigation of review comments to acquire in-depth knowledge on code reviewing practice in refactoring-inducing PRs; in this direction, we perform a characterization study (Chapter 6).

As explained in Chapter 3, it is not suitable to generalize the conclusions, except when considering other OSS projects that follow a geographically distributed development [152] and are aligned to "the Apache way" principles [17]. Thus, our findings are exclusively extended to cases that have common characteristics with Apache's projects.

Also, we systematically structured all procedures to deal with reliability issues, thus providing a reproduction kit to enable replications, publicly available at [22].

4.5 Concluding Remarks

We investigated technical aspects characterizing refactoring-inducing PRs, through an ARL process on features of a dataset containing 1,639 merged PRs, followed by a methods triangulation for analysis. Concerning the preliminary version of this study [53], our previous findings and implications remain. Our results reveal significant differences between refactoring-inducing and non-refactoring-inducing PRs, and a substantial number of refactoring edits induced by code reviewing.

Our findings are motivating, so directing efforts towards a further investigation intending to achieve an in-depth understanding of refactoring-inducing PRs. In this sense, we describe a characterization study regarding code review in refactoring-inducing PRs in the next Chapter.

Chapter 5

Characterizing Code Review in Refactoring-Inducing Pull Requests

In this chapter, we describe a qualitative characterization study on code reviewing-related aspects in refactoring-inducing PRs, based on the sample obtained from mining Apache's Java repositories in GitHub (Chapter 3).

Following, we describe the research design (Section 5.1); then, we present the results and discuss them (Section 5.2). Next, we argue a few implications and guidelines (Section 5.3), and limitations (Section 5.4).

5.1 Research Design

From our previous study (Chapter 4), Finding 6 indicates a statistical difference in the number of review comments between refactoring-inducing and non-refactoring-inducing PRs. This finding motivated us to conjecture that refactoring-inducing and non-refactoring-inducing PRs have distinct characteristics regarding review comments. Given that, this qualitative study investigates code reviewing-related aspects (review comments and discussion) intending to characterize code review in refactoring-inducing PRs. Note that *discussion*, as we deemed, denotes either a dialog between a PR author and reviewer(s) or an author's response in opposition to a specific review comment – we give examples forward. Thus, we designed these **research questions**:

- RQ$_1$: How are review comments characterized in refactoring-inducing and non-refactoring-inducing PRs? We first identified the characteristics of review comments in refactoring-inducing and non-refactoring-inducing PRs.
- RQ$_2$: What are the differences between refactoring-inducing and non-refactoring-inducing PRs, in terms of review comments? Then, we investigated the existence of similarities/dissimilarities regarding review comments in refactoring-inducing non-refactoring-inducing PRs by contrasting their corresponding characteristics.
- RQ$_3$: How do reviewers suggest refactorings in refactoring-inducing PRs? We scrutinized the review comments in refactoring-inducing PRs in order to identify patterns for suggesting refactorings.
- RQ$_4$: Do suggestions of refactoring justify the reasons? While scrutinizing the suggestions of refactoring, by manually examining review comments, we explored whether/how reviewers provide rationales for their proposals.
- RQ$_5$: What is the relationship between suggestions and actual refactorings in refactoring-inducing PRs? Likewise, while scrutinizing suggestions of refactoring in review comments, we also investigated patterns that would denote any relationship between suggestions and actual refactorings.

Accordingly, we designed a qualitative study that comprises four rounds of analysis (performed from June 21 to October 08, 2021), where each one follows the procedures depicted in Figure 5.1, as described in the next subsections.

5.1.1 Selection of a Purposive Sample

At each round, we investigated review comments from a *purposive sample of PRs* while cross-referencing their detected refactoring edits. We adopted such a non-probability sampling to select refactoring-inducing and non-refactoring-inducing PRs from the *code review dataset* (Chapter 3, Subsection 3.1.3). It is worth clarifying that we followed that sampling strategy until we reached *data saturation* (when no new information emerges) [96] – it was achieved after four rounds of analysis.

At each round, we examined a purposive sample (Step 1) fitting a valuable scenario to the current purposes of the analysis. We chose purposive sampling because it

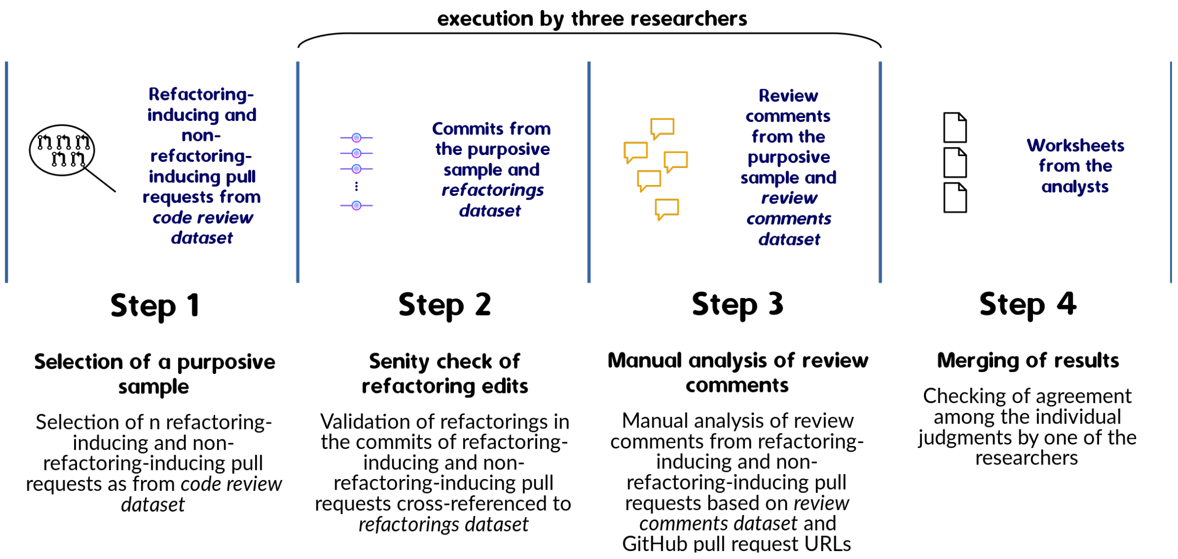

execution by three researchers
Refactoring-inducing and non-refactoring-inducing pull requests from code review dataset
Commits from the purposive sample and refactorings dataset
Review comments from the purposive sample and review comments dataset
Worksheets from the analysts
Step 1
Selection of a purposive sample
Selection of n refactoring-inducing and non-refactoring-inducing pull requests as from code review dataset
Step 2
Senity check of refactoring edits
Validation of refactorings in the commits of refactoring-inducing and non-refactoring-inducing pull requests cross-referenced to refactorings dataset
Step 3
Manual analysis of review comments
Manual analysis of review comments from refactoring-inducing and non-refactoring-inducing pull requests based on review comments dataset and GitHub pull request URLs
Step 4
Merging of results
Checking of agreement among the individual judgments by one of the researchers

Figure 5.1: An overview of round i of our characterization study of code review in refactoring-inducing PRs

provides an in-depth understanding of the data by exploring scenarios suitable at each round, in line with emergent patterns or ideas [92; 114]. Note that, in all rounds, we selected more representative samples, in line with emergent patterns, intending to obtain more accurate results than those achieved from selecting other probability sampling strategies. For instance, in the second round, we explored refactoring-inducing PRs containing only one low-level refactoring edit. Then, in the third round, we examined refactoring-inducing PRs comprising high-level refactoring edits in order to investigate whether the emerged patterns from Round 2 remain in Round 3. Table 5.1 outlines the main objective of each round of analysis. The emerged patterns from Round 1 continue until Round 4, in which we reach the saturation point.

Table 5.1: A summary of the main objective of each round of qualitative analysis

Round	Main objective
Round 1	Investigating an initial random sample of PRs
Round 2	Checking if emerged patterns remain in refactoring-inducing pull requests consisting of only one refactoring edit

Continued on next page

Table 5.1 – continued from previous page

Round	Main objective
Round 3	Exploring if emerged patterns persist in refactoring-inducing pull requests consisting of high-level refactorings
Round 4	Inspecting if emerged patterns continue regardless of the sequence of refactoring edits

Based on guidelines [55], we empirically considered 20 as the minimum size for the purposive samples. In specific, Creswell has recommended 15-20 interviewers during a grounded theory study, which comes closest to our characterization study (when investigating comments from reviewers). Thus, as a PR has at least one reviewer, we took into account 20: ten refactoring-inducing PRs and ten non-refactoring-inducing PRs. Then, for the first round of analysis (sample 1), looking for a fair comparison between groups when addressing RQ_1 and RQ_2, we randomly selected ten refactoring-inducing and ten non-refactoring-inducing PRs that contain five review comments and two reviewers – intermediate values regarding the average and median of number of review comments and reviewers in both refactoring-inducing and non-refactoring-inducing PRs (Chapter 4, Table 4.4).

We also considered 20 as the size for the second purposive sample (sample 2, Round 2): ten refactoring-inducing and ten non-refactoring-inducing PRs that contain only one subsequent commit. At that point, we investigated whether the patterns that emerged from distinct compositions of refactoring edits (Round 1, in which the number of refactorings is 5.7 on average and 2.5 on the median) even remain in refactoring-inducing PRs consisting of only one refactoring edit – the most simple setting of refactoring in a PR. We addressed PRs comprising a single subsequent commit in order to compare refactoring-inducing and non-refactoring-inducing PRs (RQ_1 and RQ_2).

In the third purposive sample (sample 3, Round 3), to address RQ_3 to RQ_5, we considered 13 refactoring-inducing PRs from 36 that present high-level refactorings in order to explore whether the emergent patterns from the previous rounds (which comprise less complex PRs) persist. Those 13 refactoring-inducing PRs embrace a diversified setting of refactoring edits of distinct types (found in those 36), including only one type of refactoring (e.g., Dubbo #3654) and a mix of types of refactoring (e.g., Incubator-Iceberg #183). We decided to explore refactoring-inducing PRs consisting of high-level refactorings because only 2/24 (8.3%) refactoring-inducing PRs contain high-level refactoring edits in the previous samples (Rounds 1 and 2). To deal with RQ_1 and RQ_2, we examined a randomly selected sample of 13 non-refactoring-inducing PRs that present ten review comments – median of the number of review comments in those 13 refactoring-inducing PRs. To be concise, we use the format <Repository #number> to represent a PR of an Apache repository.

In the fourth purposive sample (sample 4, Round 4), to address RQ_3 to RQ_5, we studied the 26 refactoring-inducing PRs that present distinct sequences of refactoring edits (of different types) in PR commits history (e.g., instances of *Rename Variable* and *Extract Variable* in a single commit against ones in two separated commits). In particular, we explored whether the emergent patterns from the previous rounds persist regardless of the sequence of refactoring edits applied to the commits history. To address RQ_1 and RQ_2, we investigated a randomly selected sample of 26 non-refactoring-inducing PRs that present seven review comments – the median value of the number of review comments in those 26 refactoring-inducing PRs.

Table 5.2 summarizes the number of refactoring-inducing and non-refactoring-inducing PRs, review comments, subsequent commits, and refactoring edits considered in each round of analysis.[1] The different number of refactoring-inducing and non-refactoring-inducing PRs in rounds 1, 2, and 4 are due to the manual validation of refactoring edits mined by RefactoringMiner, described in Subsection 5.1.2. We provide more details regarding the characteristics of our samples and results in Appendix D.

The magnitude of both refactoring-inducing and non-refactoring-inducing PRs increases in the following order: *sample 2* $<$ *sample 1* $<$ *sample 4* $<$ *sample 3*, when considering size-related aspects (number of subsequent commits, number of file changes, number of added lines, and number of deleted lines), as displayed in Appendix D (Tables D.1 and D.2).

[1]More details on each purposive sample are available at `https://git.io/JMbnV`.

Table 5.2: Summary of the purposive samples

Round	***No. of refactoring-inducing PRs***	***No. of non-refactoring-inducing PRs***	***No. of review comments***	***No. of subsq. commits***	***No. of refactoring edits***
Round 1	13	7	100	40	68
Round 2	11	9	87	20	11
Round 3	13	13	327	126	209
Round 4	28	24	409	160	78
All rounds	*65*	*53*	*923*	*346*	*366*

5.1.2 Sanity Check of Refactoring Edits and Manual Analysis of Review Comments

As shown in Figure 5.1, at each round, three researchers checked all commits searching for false positives and false negatives in refactoring edits (Step 2), supported by the *refactorings dataset* (Chapter 3, Subsection 3.1.2). Thus, the researchers individually verified the type and description of the refactoring data generated from RefactoringMiner.

RefactoringMiner mistakenly detected an edit as a *Rename Method* (Kafka #6565, sample 2) and did not identify edits of *Extract Attribute* (Accumulo-Examples #19, sample 4), *Extract Variable* (Commons-Text #39, sample 1 and Tinkerpop #893, sample 4), *Extract Method* (Hadoop #942, sample 1), *Inline Variable* (Dubbo #3185, sample 2), *Move Attribute* (Beam #6261, sample 3), and *Rename Method* (Kafka #7132, sample 2). Therefore, as shown in Table 5.3, RefactoringMiner achieved a precision of 99.7% and recall of 98.1% for refactoring detection in 118 PRs.

Table 5.3: Results of the manual validation of refactoring edits mined by RefactoringMiner 1.0

Sample	***TP***	***FP***	***FN***	***Precision (%)***	***Recall (%)***
Sample 1	66	0	2	100	97.1

Continued on next page

Table 5.3 – continued from previous page

Sample	***TP***	***FP***	***FN***	***Precision (%)***	***Recall (%)***
Sample 2	9	1	2	90	81.8
Sample 3	208	0	1	100	99.5
Sample 4	76	0	2	100	97.4
All samples	*359*	*1*	*7*	*99.7*	*98.1*

Next, each researcher apart examined the review comments from all PRs of the sample (Step 3), assisted by the *refactorings dataset* and *code review dataset.* To clarify, besides seeking patterns in review comments, the researchers cross-referenced the type of refactorings performed (if so) and the review comments left in all commits of a PR. In such a procedure, it was crucial to directly examine PRs at GitHub because we could access commits and review comments in chronological order.

5.1.3 Merging of Results

At each round, one researcher checked all individual judgments in order to achieve a concluding judgment concerning the sample (Step 4). As decision criteria, we considered the agreement of responses by at least two researchers. To support such a procedure, we used worksheets[2] structured with the following fields:

- *repo*: repository name,
- *pr_number*: PR number,
- *category*: refactoring-inducing PR or non-refactoring-inducing PR,
- *pr_url*: PR URL,
- *commit*: commit SHA,
- *initial_flag*: whether a commit is initial or subsequent commit,
- *refactoring_type*: type of a refactoring edit,
- *refactoring_detail*: description of a refactoring edit, as RefactoringMiner output,
- *confirmed_refactoring_flag*: if a refactoring is a true positive,

[2] Available at `https://git.io/JMbnV`.

- *covered_refactoring_flag*: if a refactoring edit was induced by code review,
- *floss_refactoring_flag*: if there is the presence of floss refactoring in a commit,
- *direct_review_comment_flag*: if a review comment directly suggest a refactoring edit,
- *discussion_flag*: if there was discussion related to a review comment in a commit,
- *rationale_flag*: if a review comment presents a rationale to suggest a refactoring edit, and
- *notes*: specific comments of a researcher.

It is noteworthy that, in such subjective analysis, the researchers considered the refactoring-inducement in settings where review comments either explicitly suggested refactoring edits or left any actionable recommendation that induced refactoring. For example, "*... the name is really misleading ...*" induced a *Rename Method* (Avro #525), whereas "*Won't you need to use a single instance for both arguments?*" inspired an *Extract Attribute* (Beam #4407).

Furthermore, a researcher explored the individual answers for all research questions intending to identify emerged patterns. Therefore, the concluding analysis of each step denotes an incremental knowledge regarding refactoring-inducing PRs in light of our research questions. Those intermediate judgments and emerged patterns are available in our reproduction kit [22].

5.2 Results and Discussion

It is noteworthy that this qualitative study aims to advance the knowledge regarding code review in refactoring-inducing PRs. Therefore, the following findings are supplementary to those from our previous study (Chapter 4). For instance, note that we designed distinct sampling strategies in the two studies – a stratified sampling to explore refactorings induced by code review (Chapter 4, Subsection 4.2.3) and purposive sampling in this study (Subsection 5.1.1).

Before answering the research questions, we describe a few preliminary results from the analysis of the general characteristics of refactoring-inducing and non-refactoring-inducing PRs, intending to support the following results and discussion (Subsections 5.2.2–5.2.6).

5.2.1 Preliminary Results

During the analysis of review comments, a few peculiarities emerged.

Refactoring-inducement rates

First, as we can see in Table 5.4, 49/65 (75.4%) of refactoring-inducing PRs are due to code review, being the refactoring edits induced by code review and led by the PR authors in 14/49 (28.6%) ones. To clarify, a refactoring-inducing PR in which a refactoring was led by its author expresses that there is no mention (direct or indirect) of the refactoring in the review comments. We identified refactoring edits led only by authors in 16/65 (24.6%) of refactoring-inducing PRs, comprising 3/13 (23.1%) in sample 1 (Beam #4460, Flink #7971, Samza #1030), 2/11 (18.2%) in sample 2 (Incubator-Pinot #479, Kafka #5423), 4/13 (30.8%) in sample 3 (Beam #6261, Dubbo #3654, Kafka #6657, Usergrid #102), and 7/28 (25%) in sample 4 (Accumulo #151, Dubbo #2445, Dubbo #4099, Logging-log4j #213, Kafka #4574, Tinkerpop #893, Tomee #89). We identified 35/65 (53.8%) refactoring-inducing PRs that exclusively consist of refactorings induced by code review – an almost similar result to our Finding 9 (Chapter 5), in which we indicate 58.3% of such setting.

Table 5.4: Inducement by code review in refactoring-inducing PRs

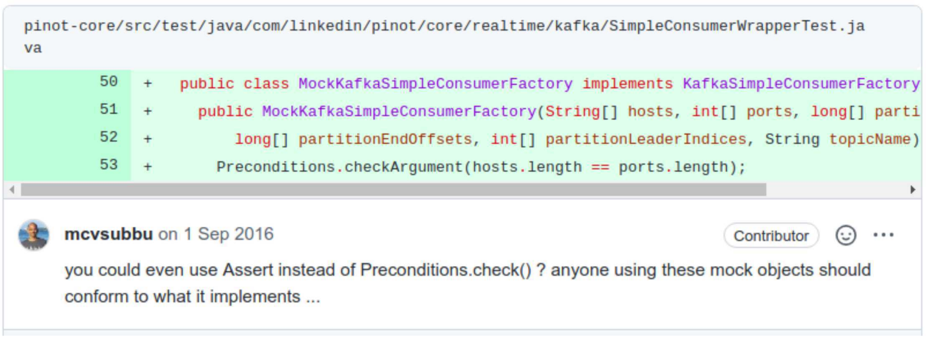
Sample	Judgement	PRs
Induced by code review		
Sample 1	10/13 (76.9%)	Dubbo #3299, Commons-text #39, Flink #9143, Fluo #837, Hadoop # 942, Incubator-iceberg #254, Kafka #5194

Continued on next page

Table 5.4 – continued from previous page

Sample	**Judgement**	**PRs**
Sample 2	9/11 (81.8%)	Beam #4407, Beam #4458, Brooklyn-Server #1049, Dubbo #3185, Kafka #5784, Kafka #7132, Samza #1051, Servicecomb-Java-Chassis #346, Tomee #275
Sample 3	9/13 (69.2%)	Cloudstack #2071, Cloudstack #3454, Kafka #5590
Sample 4	21/28 (75%)	Accumulo-Examples #19, Brooklyn-Server #964, Cloudstack #2833, Dubbo #3174, Dubbo #3257, Kafka #4796, Kafka #6853, Knox #69, Knox #74, Sling-Org-Apache-Sling-Feature-Analyser #16, Struts #43, Tika #234, Tinkerpop #1110, Tomee #407
All samples	*49/65 (75.4%)*	
(also) Led by the author		
Sample 1	3/10 (30%)	Dubbo #2279, Flink #7970, Flink #7945
Sample 2	none	
Sample 3	6/9 (66.7%)	Flink #8222, Incubator-Iceberg #119, Incubator-Iceberg #183, Kafka #4735, Kafka #4757, Servicecomb-Java-Chassis #678
Sample 4	7/21 (33.3%)	Avro #525, Flink #7165, Flink #8620, Kafka #5501, Kafka #5946, Kafka #6848, Rocketmq-Externals #45
All samples	*14/49(28.6%)*	

Note that, in sample 2, 9/11 refactoring-inducing PRs present refactoring edits exclusively induced by code review (2/11 have refactorings solely led by the authors). Thus, in 9/9 refactoring-inducing PRs, code review induced the refactorings. In sample 3, code review induced refactorings in 9/13 refactoring-inducing PRs, being exclusively induced by code review in 3/9 of them. In 6/9 of refactoring-inducing PRs, the refactorings were induced by code review and also led by the authors. Figures 5.2 and 5.3 depict refactoring-inducing PRs in which authors conducted refactorings and code review induced refactorings, respectively. Figure 5.2 displays a single review comment, suggesting the use of Assertion in Incubator-Pinot #479, which resulted in no effect in terms of refactoring. In this case, the author led the refactoring edit, an Extract Variable. Figure 5.3 shows a review comment indicating an outdated protocol in Cloudstack #2071, which influenced a *Rename Package* edit.

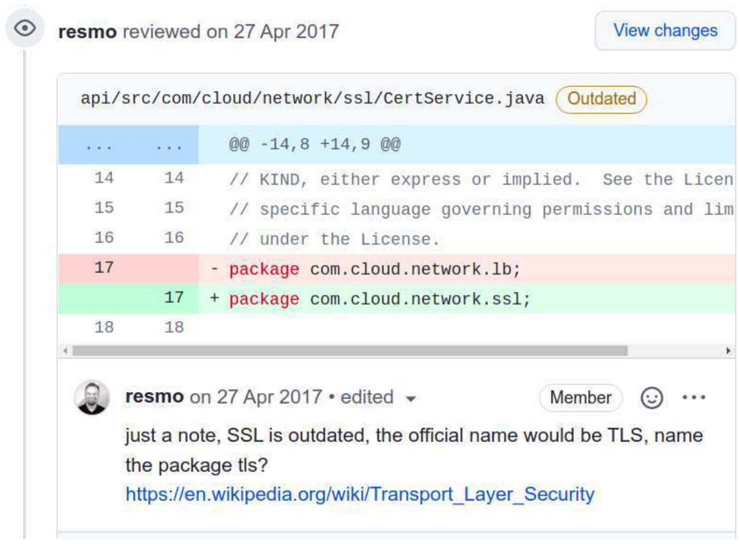


```
pinot-core/src/test/java/com/linkedin/pinot/core/realtime/kafka/SimpleConsumerWrapperTest.ja
va
+  public class MockKafkaSimpleConsumerFactory implements KafkaSimpleConsumerFactory
+    public MockKafkaSimpleConsumerFactory(String[] hosts, int[] ports, long[] parti
+        long[] partitionEndOffsets, int[] partitionLeaderIndices, String topicName)
+      Preconditions.checkArgument(hosts.length == ports.length);
```

mcvsubbu on 1 Sep 2016 Contributor

you could even use Assert instead of Preconditions.check() ? anyone using these mock objects should conform to what it implements ...

Figure 5.2: A single review comment, from Apache Incubator-Pinot PR #479, which induced no refactoring edit



resmo reviewed on 27 Apr 2017 View changes

api/src/com/cloud/network/ssl/CertService.java Outdated

```
@@ -14,8 +14,9 @@
 // KIND, either express or implied.  See the Licen
 // specific language governing permissions and lim
 // under the License.
- package com.cloud.network.lb;
+ package com.cloud.network.ssl;
```

resmo on 27 Apr 2017 • edited Member

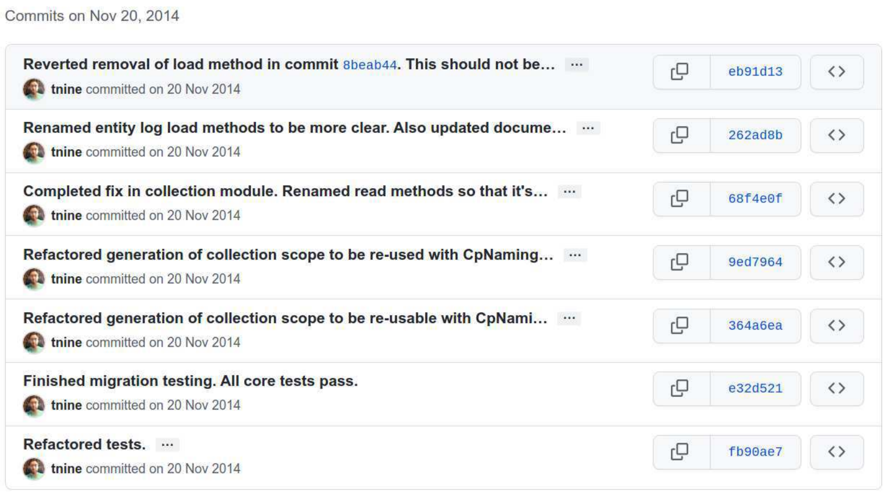
just a note, SSL is outdated, the official name would be TLS, name the package tls?
https://en.wikipedia.org/wiki/Transport_Layer_Security

Figure 5.3: A review comment, from PR Apache Cloudstack #2071, which induced a *Rename Package* edit

Almost half of refactoring-inducing PRs in which code review induced the refactorings, figuring 168/359 (46.8%) edits. For comparison purposes, Pantiuchina et al. found that reviewing discussion triggered about 35% of refactoring edits when analyzing 551 PRs from 150 distinct projects in GitHub [112]. Nonetheless, they considered refactoring edits occurring in any PR commits (including the initial ones), regardless of what led to the refactorings. Therefore, although considering a distinct research design, our result corroborates with Pantiuchina et al. because it emphasizes refactoring as a relevant aiding from code reviewing in the pull-based development model, already associated with a large number of contributions [153].

How authors document refactoring edits

Second, we observed the presence of *self-affirmed refactorings* or *self-admitted refactorings* (when a refactoring is cited explicitly in a commit message by using keywords like "*Refactor...*", "*Mov...*", "*Renam...*"), as illustrated in Figure 5.4, in subsequent commits of 9/65 (13.8%) refactoring-inducing PRs (Table 5.5). In five of those PRs (Beam #6261, Dubbo #4099, Flink #7971, Tomee #407, and Usergrid #102), all refactorings edits were led by the author. In the other ones (Avro #525, Struts #43, Tinkerpop #1110, and Tomee #407), authors submitted self-affirmed refactorings to meet code review suggestions.


Commits on Nov 20, 2014

Reverted removal of load method in commit 8beab44. This should not be... ···
tnine committed on 20 Nov 2014 — eb91d13

Renamed entity log load methods to be more clear. Also updated docume... ···
tnine committed on 20 Nov 2014 — 262ad8b

Completed fix in collection module. Renamed read methods so that it's... ···
tnine committed on 20 Nov 2014 — 68f4e0f

Refactored generation of collection scope to be re-used with CpNaming... ···
tnine committed on 20 Nov 2014 — 9ed7964

Refactored generation of collection scope to be re-usable with CpNami... ···
tnine committed on 20 Nov 2014 — 364a6ea

Finished migration testing. All core tests pass.
tnine committed on 20 Nov 2014 — e32d521

Refactored tests. ···
tnine committed on 20 Nov 2014 — fb90ae7

Figure 5.4: Example of self-affirmed refactoring in five subsequent commits, from Apache Usergrid PR #102

Table 5.5: Refactoring-inducing PRs containing self-affirmed refactorings in their subsequent commits

Sample	***No. of PRs***	***PRs***
Sample 1	1/13 (7.7%)	Flink #7971
Sample 2	0/11 (0%)	
Sample 3	2/13 (15.4%)	Beam #6261, Usergrid #102
Sample 4	6/28 (21.4%)	Avro #525, Dubbo #4099, Struts #43, Tinkerpop #1110, Tomee #89, Tomee #407
All samples	*10/65 (15.4%)*	

In an effort of comparing, this result slightly differs from the findings of AlOmar et al., who manually analyzed how developers document refactoring edits during software evolution exploring PR commits (including initial and subsequent ones) on GitHub [27]. They identified that authors use a variety of expressions to target the commits, as well as we observed. However, while they detected explicit specification of improvements (e.g., quality attributes and code smell) in the commit messages as well we noticed, we also found direct mentions such as "*Refactor*" (Avro #525).

Types of change

Third, refactoring-inducing and non-refactoring-inducing PRs comprise the three *primary types of change* (adaptive, corrective, and perfective), as indicated in Table 5.6. We list the refactoring-inducing PRs by type of change in Appendix D (Table D.3). To clarify, the researchers manually explored the objective of each sample of PRs (thus, considering initial and subsequent commits, PR descriptions, and commit messages) by searching for keywords that could denote type of changes. Thus, we classified the type of changes according to maintenance activities, as defined in previous guidelines [98; 137]. In specific, adaptive, corrective, and perfective changes comprise adding new features (e.g., "add", "new", "update"), fixing faults (e.g., "fix", "correct"), and restructurings to accommodate future changes (e.g., "simplify", "optimize"), respectively. We emphasize that such a judgment is subjective and endorsed by the researchers. In a few refactoring-inducing PRs, they found both adaptive and corrective changes (Accumulo #151 and Dubbo #2279) and corrective and perfective changes (Dubbo #3654 and Tomee #275). Only (1/38) 2.6% of non-refactoring-inducing PRs comprise perfective changes related to enhancements of code documentation.

Table 5.6: Type of changes by category of PRs

Sample	**Type of changes**		
	Adaptive	**Corrective**	**Perfective**
Refactoring-inducing PRs			
Sample 1	3/13 (23.1%)	5/13 (38.5%)	4/13 (30.8%)
Sample 2	4/11 (36.4%)	4/11 (36.4%)	2/11 (18.2%)
Sample 3	7/13 (53.8%)	3/13 (23.1%)	2/13 (15.4%)
Sample 4	11/28 (39.2%)	8/28 (28.6%)	8/28 (28.6%)
All samples	*25/65 (38.5%)*	*20/65 (30.8%)*	*16/65 (24.6%)*
non-Refactoring-inducing PRs			
Sample 1	none	4/4 (100.0%)	none
Sample 2	none	6/6 (100.0%)	none
Sample 3	6/11 (54.5%)	4/11 (36.4%)	1/11 (9.1%)
Sample 4	9/17 (52.9%)	8/17 (47.1%)	none
All samples	*15/38 (39.5%)*	*22/38 (57.9%)*	*1/38 (2.6%)*

We clarify that, even in Java repositories, PRs can present non-Java files (e.g., Scala and Python code), sometimes resulting in no review comment about Java files. We found such a characteristic in 15/53 (28.3%) of the non-refactoring-inducing PRs examined (Beam #4261, Beam #4419, Beam #5772, Beam #5785, Beam #7696, Beam #8140, Cloudstack #2706, Flink #4055, Flink #9451, Incubator-Iotdb #342, Kafka #6298, Kafka #5368, Kafka #6758, Parquet-Format #98, Tinkerpop #690). Accordingly, we considered 38/53 non-refactoring-inducing PRs when computing the number of PRs by type of change.

We carried out such a classification aiming to explore potential emergent patterns from distinct types of changes towards a more comprehensive characterization of refactoring-inducing PRs. Our results corroborate with Palomba et al., who found evidence of refactoring edits in the presence of those different types of changes when exploring 63 releases of three distinct projects [110]. Accordingly, we can also consider adaptive, corrective, and perfective changes as opportunities for refactoring edits at the PR level.

Self-affirmed minor PRs and review comments

We define a self-affirmed minor PR as one in which the title or description self-declares as a minor PR, whereas a self-affirmed minor review comment is one in which a reviewer declares it as minor. For that, we searched for the keywords "*minor*" and "*nit*" (e.g., "*LGTM, just a minor comment. Should 32 be chunkSize?*" in Kafka #4574 and "*nit: Null value is encoded... –> A null value is encoded...*" in Kafka #4735).[3] We performed such an exploration in order to investigate potential patterns that could emerge from that self-affirmations; for instance, would they be present only in non-refactoring-inducing PRs?

We found self-affirmed minor PRs and self-affirmed minor review comments in both refactoring-inducing and non-refactoring-inducing PRs (Appendix D, Tables D.4 and D.5). We observed self-affirmed review comments that induced edits of *Rename* (Brooklyn-Server #964, Flink #7945, Flink #8620, Kafka #5784, Kafka #6848), *Split* (Brooklyn-Server #1049), *Inline* (Dubbo #3185), and *Extract* (Flink #8620, Kafka #4735). Therefore, we distinguished no association between self-affirmed minor PRs/review comments and refactoring-inducement because they occur in both refactoring-inducing and non-refactoring-inducing PRs.

Code review bots

Fourth, the researchers identified that 8/65 (12.3%) of refactoring-inducing PRs and 1/53 (1.9%) of non-refactoring-inducing PRs ran a *code review bot* (Appendix D, Table D.6). A repository's code review bot is easily detectable since it leaves comments in a PR, including the bot commands. For instance, the Apache flinkbot[4] checks the PR

[3]Reviewers may prefix review comments with "*nit:*", meaning that to fix a point is not mandatory but welcome.

[4]`https://github.com/flinkbot`

description, whether a PR needs attention from a specific reviewer, the architecture, and the overall code quality. Thus, we observed no association between running a code review bot and refactoring-inducement.

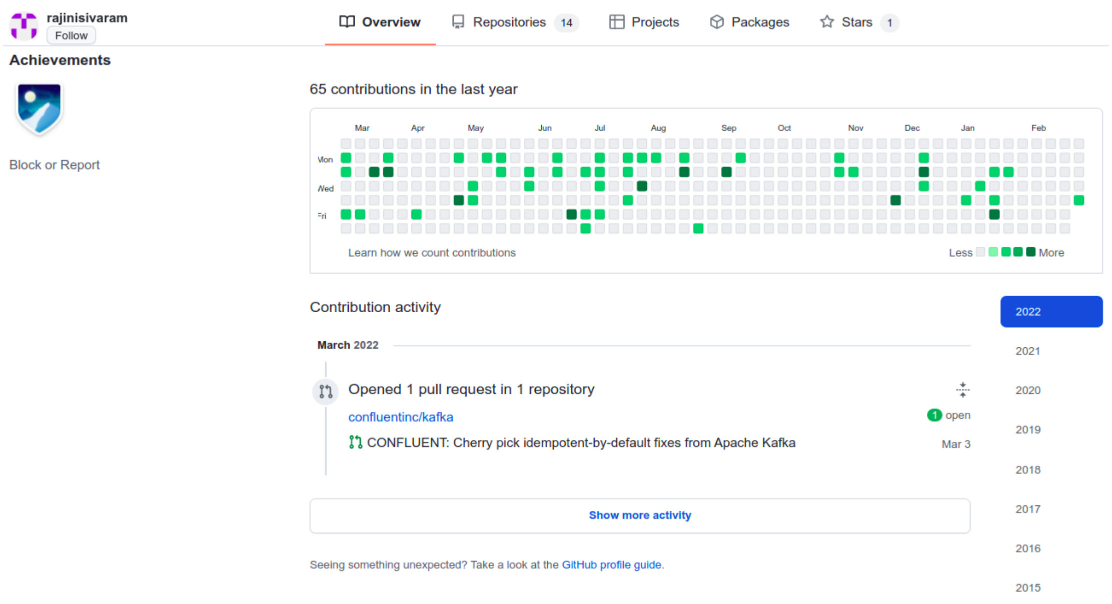
Finding 1: We found almost 50% of refactoring-inducing PRs, in which code review induced refactoring edits. The refactorings happen in PRs consisting of distinct types of change. Self-affirmed refactorings and code review bots are not frequent.

Reviewers' experience

Fifth, we investigated the experience of PR authors and reviewers, by manually examining their contributions in GitHub profiles. For counting the number of contributions of a PR author/reviewer, we computed the number of contributions considering the joining date of their profiles in GitHub until the PR creation date. To illustrate our counting strategy, consider Kafka #5784, a PR created in October 2018. Each one of the contributors has a profile from which we can access the number and the description of their contributions by year, as exemplified in Figure 5.5.[5] Specifically, the author of such a PR joined GitHub in 2015; thus, we manually counted the number of his contributions from 2015 until September 2018 (the previous month to the PR date creation) – 1,123 contributions. Accordingly, we computed the number of contributions for all authors and reviewers in our sample's PRs.

We explored such a subject because there is empirical evidence that the reviewer experience is the main factor influencing code review quality as perceived by developers [80]. We considered the number of contributions instead of contributions due to a high number of them in our samples (Appendix D, Table D.7), so expressing a time-intensive task. This setting differs from Rigby et al. that computed experience in terms of the length of time a developer has been with a project [121]. They found that reviewers typically have more experience than authors when investigating Apache HTTP Server. Thus, we extend the knowledge on the experience of Apache authors and reviewers by studying their experience in refactoring-inducing and non-refactoring-inducing PRs.

[5]Available at `https://github.com/rajinisivaram`



65 contributions in the last year

Contribution activity

March 2022

Opened 1 pull request in 1 repository

confluentinc/kafka 1 open

CONFLUENT: Cherry pick idempotent-by-default fixes from Apache Kafka Mar 3

Seeing something unexpected? Take a look at the GitHub profile guide.

Figure 5.5: Example of a GitHub profile (author of Apache Kafka PR #5784)

Furthermore, Apache designates roles for contributors [23]. A *contributor* is a developer who contributes to a project in the form of code or documentation; a *committer* is a contributor who has a write access to the code repository, and a *Project Management Committee* (PMC) *member* is a committer who has a write access to the code repository and can approve/disapprove the changes. Not all PR participants indicate their Apache's roles. Thus, we inferred such scenario from examining the profile of authors and reviewers (Appendix D, Table D.8). In addition to Apache contributors, we recognized that committers and PMC members also submit PRs to Apache repositories. It is noteworthy that we did not include the PRs with no review in Java code to compute the experience.

Given that, we could realize that authors of refactoring-inducing PRs are less experienced than authors of non-refactoring-inducing PRs, whereas reviewers of refactoring-inducing PRs are slightly more experienced than reviewers of non-refactoring-inducing PRs. For that, we analyzed the distribution of contributions of authors and reviewers in both groups of PRs (Appendix D, Table D.7). Since we found no statistical evidence that the number of reviewers is related to refactoring-inducement (Finding 7, Chapter 4), such scenario denotes a relevant motivating factor behind refactoring-inducing, by considering reviewing-related apart from review comments.

Finding 2: The experience of the PR author, inferred from the number of contributions, is a motivating factor behind refactoring-inducing PRs.

Based on this finding, we conjecture that less problem-prone code tends to give origin to non-refactoring-inducing PRs more often, which can be partially explained by the authors' experience. We contextualize the importance of reviewers to characterize refactoring-inducing PRs in Subsection 5.2.3. Even so, we claim that a further investigation on the content of contributions could provide a better understanding of the relationship between the experience of authors/reviewers and refactoring-inducing PRs.

Moreover, we identified a few particular scenarios when considering experience concerning the number of authors' and reviewers' contributions. First, code review induced refactoring edits in Fluo #837, although the number of the author's contributions (3,127) is greater than the number of contributions of its reviewers (3,040 and 2,192). The same occurs in Dubbo #3174. We conjecture that the aggregate of experiences might explain those cases. Code review also induced refactorings in PRs in which the authors have a higher number of contributions than their reviewers. In this setting, Beam #4458, Dubbo #3185, Samza #1051, Sling-Org-Apache-Sling-Feature-Analyser #16, Tika #234, Tomee #275, and Tomee #407 present only one reviewer while Brooklyn-Server #964 and Tinkerpop #1110, two reviewers. The authors of Brooklyn-Server #964 and Tinkerpop #1110 present a higher number of contributions (13,955 and 14,288) concerning the reviewers (1,660/2,933 and 24/556). Accordingly, this seems counter-intuitive in contrast to previous findings with two reviewers providing a more effective code review [119].

Second, we observed that a few PR authors led refactoring edits, even when their number of contributions was shorter in contrast to reviewers, as occurred in Accumulo #151, Beam #4460, Beam #6261, Dubbo #2445, Dubbo #3654, Dubbo #4099, Kafka #4574, Kafka #5423, Kafka #6657, Logging-Log4j #213, Samza #1030, and Tomee #89. We believe that an in-depth examination of code might help to explain that scenario.

5.2.2 How Are Review Comments Characterized in Refactoring-Inducing and non-Refactoring-Inducing PRs?

We structure this answer considering code review in non-refactoring-inducing PRs and in three dimensions of refactoring-inducing PRs: those with refactorings led by authors (16/65, 24.6%), those with refactorings induced by code review (35/65, 53.9%), and those with refactorings both led by the authors and induced by code review (14/65, 21.5%). In refactoring-inducing PRs Beam #6261, Dubbo #3654, Flink #8620, and Kafka #4757, we found the presence of floss refactoring to accommodate new tests in the three first ones and a new feature in a class in the last one – all suggested by reviewers. Thereby, we judged the associated refactoring edits as led by the PR authors.

Review comments in refactoring-inducing PRs with refactorings led by PR authors

Table 5.7 summarizes the emerged characteristics of code reviews in refactoring-inducing PRs, with refactoring edits submitted entirely by the authors. Particularly, the authors properly provide clarifications to reviewers' questions not related to potential refactorings. A typical review comment in this case:

- address *code aesthetics* (e.g., "*whitespace between braces and other symbol.*" in Flink #7971),
- present questions on *simple issues regarding code logic* in contrast to those in PRs in with refactorings induced by code review (e.g., "*Should 32 be chunkSize?*" in Kafka #4574), and
- sometimes, give *reasons* and *suggestions on code logic*, using expressions such as "*could we use...?*", "*use...*", "*can be replaced with...*" (e.g., "*token.sum() can be replaced with getToken()*" in Dubbo #3654).

Table 5.7: Characteristics of review comments in refactoring-inducing PRs, with refactorings led by the authors

Characteristic	Examples (PRs)
Addressing code aesthetics	Code format (Flink #7971, Dubbo #2445), e.g., "*format your code, pls.*"
Questioning simple issues on code logic	Access by name or index (Beam #4460), e.g., "*Don't we still use this for Dataflow in the NonFnApi mode (which passes connections by index)?*"
	Error in conditional statement (Kafka #6657, Tomee #89), e.g. "*if the endTime is less than 0 wouldn't we want to throw an exception?*"
	Treatment of specific values (Samza #1030, Kafka #4574), e.g. "*Does it make sense to fail if the partition is empty?*"
	Dealing with potential failure (a version in Accumulo #151, a generic class in Dubbo #4099, a license in Logging-Log4j #213, a file system in Tinkerpop #893), e.g., "*Logback is EPL/ LGPLv2.1. Not sure if the licensing is compatible here.*"
	Method calls (questioning the effect of a call in Beam #6261), "e.g., *Is the mockstatic call why we're adding powermock here?*"
Suggesting improvements to the code	Adding case tests (Kafka #5423, Incubator-Pinot #479, Dubbo #2445), e.g., "*Add a unit test for this case?*"
	Method calls (proposing a method replacement in Dubbo #3654, Kafka #6657), e.g., "*nit: use Objects.requireNonNull with the same message here and below*"
	Adding code documentation (Beam #6261, Logging-Log4j #213), e.g., "*Could you add detail on the documentation? perhaps some small examples, etc?*"
	Change the content of test files (Usergrid #102, Tomee #89), e.g., "*Could we put this into a resource file instead of having it hang around in the test?*"
	Use of assertion (Incubator-Pinot #479), e.g., "*you could even use Assert instead of Preconditions.check()?*"

Review comments in refactoring-inducing PRs with refactorings induced by code review

Table 5.8 indicates the emerged characteristics of reviews in refactoring-inducing PRs with refactoring edits induced only by code review. We observed that review comments:

- ask questions about *code logic* (e.g., "*should we simply keep Timeout instance (instead of keeping it in a map)?*" in Dubbo #3299),
- suggest *improvements* to the code (e.g., "*Maybe put the table name in a variable, so the string parameter is more obviously the table name in this API call, rather than something else.*" in Accumulo-Examples #19),
- and provide warnings on *good development practices* (e.g., "*Constants, private static final, are usually all caps: MAX_RETRY_COUNT. Within the context of a retryAnalyzer you could get away with count and MAX. They are both private.*" in Brooklyn-Server #1049).

Also, *uncertain review comments* (like "*I'm not sure ...*", "*wondering if...*", "*just thinking loud here ...*", and "*As far as I remember ...*") are not usually welcome; for instance, a reviewer wondering whether another strategy is appropriate to deal with code logic led to discussion but no effect (in Flink #9143 and Kafka #7132), as illustrated in Figure 5.6. Also, using *embedded code* in review comments is only appreciated when it is a suggestion instead of an imposition. For instance, Dubbo #3299's author ignored a review comment containing an embedded code (for treating a variable status); however, Dubbo #3185's author embraced a review comment proposing an embedded code ("*what do you say ...?*") as an alternative for simplifying a code. We speculate that such scenarios may affect code ownership, so explaining the behavior of PR authors. We provide more examples in Subsection 5.2.6.

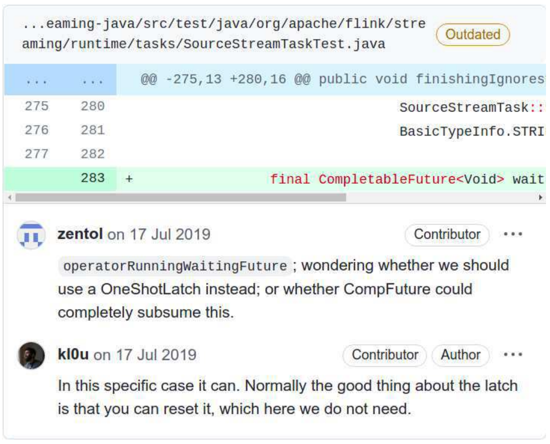
Table 5.8: Characteristics of review comments in refactoring-inducing PRs, with refactorings induced by code review

Characteristic	**Examples (PRs)**
Questioning issues on code logic	Use of specific types (Dubbo #3174 and Dubbo #3299), e.g., "*Be careful that this attachment TIMEOUT_FILTER_START_TIME will be passed throughout the RPC chain because of the drawback of RpcContext.*"
	Method calls (a question on a method signature at Commons-Text #39, a doubt in Fluo #837, a question on using a single instance to distinct arguments in Beam #4407, a question on a method call as a substitute for other ones in Kafka #5590, and Struts #43, a question on a parameter value in Accumulo-Examples #19 and Cloudstack #2833, a question on the return value in Kafka #4796), e.g., "*Is it safe to call toString here on arbitrary bytes?*"

Continued on next page

Table 5.8 – continued from previous page

Characteristic	**Examples (PRs)**
Suggesting improvements to the code	Refactoring (Accumulo-Examples #19, Beam #4458, Brooklyn-Server #964, Cloudstack #2071, Cloudstack #3454, Dubbo #3257, Flink #9143, Hadoop #942, Incubator-Iceberg #254, Kafka #4796, Kafka #5194, Kafka #5590, Kafka #5784, Kafka #6853, Kafka #7132, Knox #69, Knox #74, Samza #1051, Servicecomb-Java-Chassis #346, Sling-Org-Apache-Sling-Feature-Analyser #16, Struts #43, Tomee #275, Tomee #407, Tika #234, Tinkerpop #1110), e.g., "*Can we pass in a Path instead of a String for keystorePath?*"
	Adding code documentation (due to obfuscated name in Fluo #837, to deal with a move refactoring in Kafka #5194, due to a code update in Beam #4407, asking for comments in English in Servicecomb-Java-Chassis #346, to add a header in Tomee #407), e.g., "*it's better to write new comments in english.*"
	Adding case tests (Beam #4458, Flink #9143, Kafka #5194, Tinkerpop #1110), e.g., "*nit: add also another case for something not ending in ConfigProvider?*"
	Adding an exception handling (Kafka #5590), e.g., "*Same as with deserialize. Should we consider throwing an exception?*"
	Discard a method, by providing explanations about code design (Incubator-Iceberg #254), e.g., "*I don't think there is a need for this to be left. The truncate length should always be included when getting metrics.*"
	Issues on GUI layout (Cloudstack #3454), e.g., "*it's better now, but can you make them centered left, they seem to be bottom left now.*"
Warnings on good development practices	Code conventions (Brooklyn-server #1049), e.g., "*Constants, private static final, are usually all caps*"
	Switch-case against multiple if-else (Cloudstack #2833), e.g., "*it would be better to use a switch case instead of multiple if else, especially for comparing multiple enums*"



```
...eaming-java/src/test/java/org/apache/flink/stre
aming/runtime/tasks/SourceStreamTaskTest.java          Outdated

...  ...  @@ -275,13 +280,16 @@ public void finishingIgnores
275  280                                   SourceStreamTask::
276  281                                   BasicTypeInfo.STRI
277  282
     283  +                    final CompletableFuture<Void> wait
```

zentol on 17 Jul 2019 Contributor

`operatorRunningWaitingFuture` ; wondering whether we should use a OneShotLatch instead; or whether CompFuture could completely subsume this.

kl0u on 17 Jul 2019 Contributor Author

In this specific case it can. Normally the good thing about the latch is that you can reset it, which here we do not need.

Figure 5.6: An uncertain review comment Apache Flink PR #9143

Review comments in refactoring-inducing PRs with refactorings both led by PR authors and induced by code review

In refactoring-inducing PRs, comprising refactoring edits both led by the authors and induced by code review, the review comments present characteristics that already emerged from exploring the refactoring-inducement in PRs (Table 5.9). Uncertain review comments remain in such a scenario, where the sentences include terms such as "*Looks like...*" and "*not sure if...*" in Incubator-Iceberg #183.

Table 5.9: Characteristics of review comments in refactoring-inducing PRs, with refactorings both led by the authors and induced by code review

Characteristic	Examples (PRs)
Addressing code aesthetics	Indentation and code format (Incubator-Iceberg #119, Incubator-Iceberg #183, Kafka #6848), e.g., "*Nit: extra blank line.*"

Continued on next page

Table 5.9 – continued from previous page

Characteristic	**Examples (PRs)**
Questioning issues on code logic	Use of specific types (Dubbo #2279, Kafka #4735), e.g., "*Collection is useless.*"
	Error in a conditional statement (Kafka #5501), e.g., "*nit: do we want to consider setting producer to null here as well if eosEnabled?*"
	Treatment of specific values (Rocketmq-Externals #45), e.g., "*Do we need use the next begin offset in other PullStatus?*"
	Method calls (missing a method implementation in Dubbo #2279, doubt on the return type of a method in Flink #8222, generalization of methods in Incubator-Iceberg #183, the content of a method output in Kafka #4735, optimization in a method call in Flink #8620, Kafka #4757, Servicecomb-Java-Chassis #678), e.g., "*Can you not use .withClientSslSupport(). withClientSaslSupport()?*"
Suggesting improvements to the code	Refactoring (Avro #525, Flink #7165, Flink #7945, Flink #8222, Flink #8620, Incubator-Iceberg #119, Incubator-Iceberg #183, Kafka #4735, Kafka #5501, Kafka #5946, Kafka#6848, Rocketmq-Externals #45), e.g., "*After adding the time conversions to this method the name (or the behavior) is really misleading.*"
	Adding an exception handling (Dubbo #2279, Flink #7970), e.g., "*I think we should not easily catch on Throwable for simplicity but instead it is clear here that we should only expect IOException or ClassCastException.*"
	Adding code documentation (to make it consistent with other classes in Flink #8222, to update a method description after a change in Incubator-Iceberg #119), e.g., "*Gets -> Get, let's to be consistent with javadoc of other methods.*"
	Adding case tests (Flink #8222, Flink #8620, Incubator-Iceberg #119, Kafka #5946), e.g., "*renameTable too? maybe add test?*"

Continued on next page

Table 5.9 – continued from previous page

Characteristic	Examples (PRs)
Warnings on good development practices	Use of methods in tests (Incubator-Iceberg #183), e.g., "*Tests shouldn't use methods in other tests because they are hard to keep track of.*"
	Use of global instances (Servicecomb-Java-Chassis #678), e.g., "*global instance. it's better do not use this directly. you can save a reference in filter when UT create a new instance for it.*"
	Instructions to create integration tests (Kafka #5946), e.g., "*Adding the exception is fine, but you can just throw it directly: throw new OffsetOutOfRangeException(...) Not need to assign it to variable first :)*"

Based on Tables 5.7–5.9, when code review induces refactoring edits, we realize that review comments deal with more complex issues regarding code logic and concerns on good development practices.

Finding 3: In refactoring-inducing PRs in which code review induces refactoring edits, review comments address major issues on code logic, major improvements to the code (including refactorings directly), and warnings on good development practices.

Finding 4: In refactoring-inducing PRs in which the authors lead refactoring edits, the review comments address code aesthetics, minor issues on code logic, and minor improvements to the code.

Review comments in non-refactoring-inducing PRs

Review comments in non-refactoring-inducing PRs directly address minor issues on code logic through questions and suggestions (Table 5.10). We also found review comments including embedded code as an imposition for substituting the code ("*should we change...*"), with no effect (Dubbo #4870 and Kafka #5111). However, such a structure of review comment is welcomed when the author asks for support from the reviewer, as it occurs in Fluo #929. Therefore, using embedded code is appreciated in the form of a suggestion, as we previously argued.

Table 5.10: Characteristics of review comments in non-refactoring-inducing PRs

Characteristic	**Examples (PRs)**
Examining code aesthetics	Extra blank lines, code format (Cloudstack #3430, Dubbo #4208, Fluo #929, Kafka #4430), e.g., "*Code formatting for this section of code is different (spaces used)*"
Addressing issues on code logic	Error in conditional statements (Flink #2096, Fluo #929, Kafka #6818,Servicecomb-Java-Chassis #691, Servicecomb-Java-Chassis #698), e.g., "*I think b==0?true:false can be replaced with b==0.*"
	Exception handling (Brooklyn-Server #411, Cloudstack #2553, Dubbo #3748, Servicecomb-Java-Chassis #691), e.g., "*IllegalStateException is better?*"
	Method calls (using another version of an overloaded method in Beam #6050, doubt on a specific method call in Dubbo #3184, Dubbo #3447, Servicecomb-Java-Chassis #691, Servicecomb-Java-Chassis #744, suggesting a method overriding in Cloudstack #2714, doubt on visibility modifiers in Struts #191), e.g., "*What about overriding the method getCause to return this value here?*"
	Reverting of changes (Cloudstack #2553, Cloudstack #3276, Dubbo #3447, Incubator-Iotdb #67, Kafka #5111), e.g., "*L116 and 118 also have changes that should be reverted.*"
	Removing of code fragments (Dubbo #3184, Dubbo #3317, Dubbo #4870, Kafka #5219, Kafka #6818), e.g., "*why those code are removed? are we still need -Djava.net.preferIPv4Stack= true after this change are made?*"
	Requiring additional states for sessions (Kafka #6427), e.g., "*I wonder if we could include any additional state from the session itself to get closer to the root of the problem.*"

Continued on next page

Table 5.10 – continued from previous page

Characteristic	Examples (PRs)
Suggesting improvements to the code	Adding assertion (Beam #6317, Dubbo #3447), e.g., "*nit: Add assertion for a part of the exception message?*"
	Adding case tests (Cloudstack #2714, Cloudstack #3276, Fluo #929, Kafka #6818), e.g., "*can you implement a marvin test for the test-case.*"
	Adding code documentation (Accumulo-Testing #21, Brooklyn-server #411, Cloudstack #3430, Incubator-Pinot #880, Kafka #4430, Servicecomb-Java-Chassis #691, Tomee #283), e.g., "*It might nice to add some comments describing the difference between informational and comparison.*"
	Fixing code documentation (Accumulo-Examples #50, Beam #6050, Dubbo #4208, Plc4x #9), e.g., "*This comment could be improved to explain why the client needs to be left open here. I don't think it's merely a matter of 'simplicity'.*"
	Alternatives to code fragments (Beam #6050, Cloudstack #3276, Incubator-Pinot #880, Kafka #4430, Kafka #5111, Servicecomb-Java-Chassis #691, Servicecomb-Java-Chassis #969, Tinkerpop #282, Tinkerpop #524), e.g., "*we should not copy the code from the old addGlobalStore() but rather call the old addGlobalStore() passing the generated names.*"
	Alternatives to error/output messages (Dubbo #3317, Fluo #929, Incubator-Iotdb #67, Servicecomb-Java-Chassis #691), e.g., "*Suggest using 'LOGGER.error("Cannot get PID of IoTDB process because ", e);*"'
	Fixing a typo (Flink #91), e.g., "*typo: terminated*"
	Use of better argument values (Cloudstack #3333, Cloudstack #3430, Dubbo #3331, Kafka #6438, Kafka #6565), e.g., "*I think this may cause an issue, because the vlan.getVlanTag() could be something like a range 100-200, so if vlanId is 101 the check should be done in the range.*"

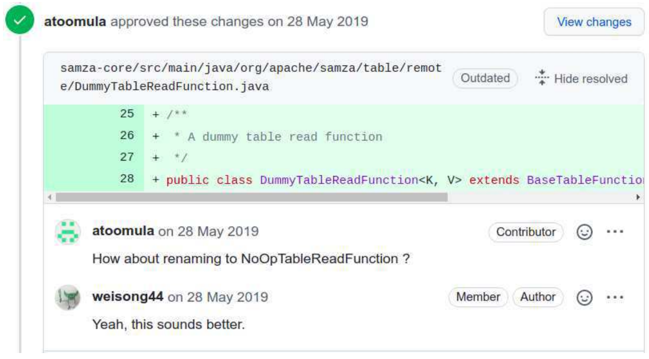
Those minor issues denote subjects having no relation to refactoring, such as fixing an error in conditional statements and using the correct range of variables. Usually, the authors provide answers to the reviewers, even in the presence of discussions, resulting in no effect (Beam #7696, Brooklyn-server #411, Cloudstack #2553, Cloudstack #2714, Cloudstack #3276, Dubbo #3184, Dubbo #3317, Dubbo #3748, Dubbo #4870, Incubator-Iotdb #67, Kafka #5111, Kafka #6818, Servicecomb-Java-Chassis #691, Servicecomb-Java-Chassis #744, Tinkerpop #282, Tinkerpop #524).

Finding 5: In non-refactoring-inducing PRs, review comments address code aesthetics, marginal issues on code logic, and marginal improvements to the code.

5.2.3 What Are the Differences Between Refactoring-Inducing and non-Refactoring-Inducing PRs, in Terms of Review Comments?

We realize that review comments in refactoring-inducing and non-refactoring-inducing PRs are different concerning the following criteria:

- **Addressed issues**. Based on Tables 5.7–5.10, code reviewing addresses *code aesthetics*, *code logic*, and *improvements* in both refactoring-inducing and non-refactoring-inducing PRs. From Findings 3–5, we recognize that review comments concern trivial issues on code logic and suggestions of minor improvements to the code occurred more in non-refactoring-inducing PRs than in refactoring-inducing PRs. In a typical PR with refactorings induced by code review, usually, review comments point out a structural problem, as in Samza #1051 (Figure 5.7), in which the author applied a *Rename Class* after a review comment. Reviewers ask questions regarding issues of minor scope in non-refactoring-inducing PRs (in this case, a typical review comment asks on alternatives already implemented by the author, e.g., a question on an overloaded method in Beam #6050, Figure 5.8), whereas they deal with major issues that can induce refactoring in refactoring-inducing PRs (in this case, a typical review comment addresses more complex structural issues, as in Tinkerpop #1110 when concerning a potential *Extract Method*, Figure 5.9).


atoomula approved these changes on 28 May 2019 View changes

samza-core/src/main/java/org/apache/samza/table/remote/DummyTableReadFunction.java Outdated Hide resolved

```
+ /**
+  * A dummy table read function
+  */
+ public class DummyTableReadFunction<K, V> extends BaseTableFunctio
```

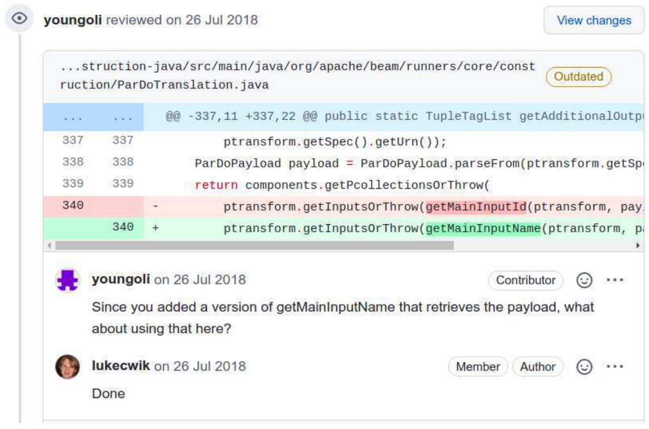
atoomula on 28 May 2019 Contributor

How about renaming to NoOpTableReadFunction ?

weisong44 on 28 May 2019 Member Author

Yeah, this sounds better.

Figure 5.7: A typical suggestion of *Rename* edit from Apache Samza PR #1051


youngoli reviewed on 26 Jul 2018 View changes

...struction-java/src/main/java/org/apache/beam/runners/core/construction/ParDoTranslation.java Outdated

```
@@ -337,11 +337,22 @@ public static TupleTagList getAdditionalOutp
        ptransform.getSpec().getUrn());
    ParDoPayload payload = ParDoPayload.parseFrom(ptransform.getSp
    return components.getPcollectionsOrThrow(
-       ptransform.getInputsOrThrow(getMainInputId(ptransform, pay
+       ptransform.getInputsOrThrow(getMainInputName(ptransform, p
```

youngoli on 26 Jul 2018 Contributor

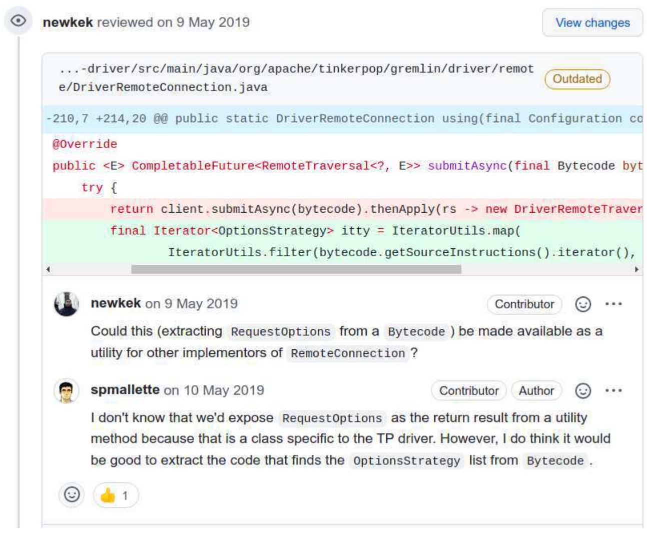
Since you added a version of getMainInputName that retrieves the payload, what about using that here?

lukecwik on 26 Jul 2018 Member Author

Done

Figure 5.8: A review comment on a minor scope issue from Apache Beam PR #6050

Also, reviewers provide suggestions of minor improvements to the code (e.g., alternatives to output messages) in non-refactoring-inducing PRs, whereas directly suggest refactoring (e.g., alternative to a method signature) in refactoring-inducing PRs. The same patterns emerged when comparing review comments in refactoring-inducing PRs, where authors led the refactoring edits (Table 5.7), and non-refactoring-inducing PRs (Table 5.10). In such a context, we found more questions on minor issues on code logic in non-refactoring-inducing PRs (e.g., a doubt concerning a method call) than in refactoring-inducing PRs (e.g., a question on the effect of a method call); and suggestions of minor improvements in non-refactoring-inducing PRs (e.g., proposing the use of "*A*", "*B*", ... instead of "*X*", "*Y*", ... as method arguments) than in refactoring-inducing PRs (e.g., suggesting a method replacement). We found warnings on good practices of de-


newkek reviewed on 9 May 2019

View changes

```
...-driver/src/main/java/org/apache/tinkerpop/gremlin/driver/remot
e/DriverRemoteConnection.java
```

Outdated

```
-210,7 +214,20 @@ public static DriverRemoteConnection using(final Configuration co
@Override
public <E> CompletableFuture<RemoteTraversal<?, E>> submitAsync(final Bytecode byt
    try {
        return client.submitAsync(bytecode).thenApply(rs -> new DriverRemoteTraver
        final Iterator<OptionsStrategy> itty = IteratorUtils.map(
                IteratorUtils.filter(bytecode.getSourceInstructions().iterator(),
```

newkek on 9 May 2019 Contributor

Could this (extracting `RequestOptions` from a `Bytecode`) be made available as a utility for other implementors of `RemoteConnection` ?

spmallette on 10 May 2019 Contributor Author

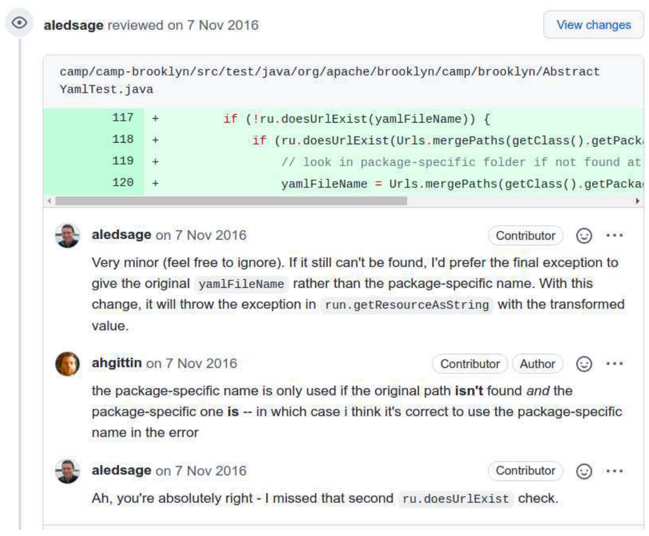
I don't know that we'd expose `RequestOptions` as the return result from a utility method because that is a class specific to the TP driver. However, I do think it would be good to extract the code that finds the `OptionsStrategy` list from `Bytecode` .

👍 1

Figure 5.9: A review comment on a major scope issue from Apache Tinkerpop PR #1110

velopment only in refactoring-inducing PRs that, in turn, may induce refactorings (Tables 5.8 and 5.9).

- **Discussion**. We found reviewing discussion in 19/65 (29.2%) refactoring-inducing PRs and 15/38 (39.5%) non-refactoring-inducing PRs, where 38 indicates the number of non-refactoring-inducing PRs that contain reviewed Java files. We also detected reviewing discussion in two non-refactoring-inducing PRs (Dubbo #3447 and Dubbo #3184), in which reviewers agree with each other when addressing code issues (those issues concern no structural changes, so no inducing refactorings). In refactoring-inducing PRs, reviewing discussion tends to arise as from either warning on good practices of development (authors may require explanations due to no knowledge of such practices – Kafka #4757) or questions/opinions concerning code logic, such as addressing potential failures and alternatives to a specific decision making (e.g., change a package name to meet project standards may break external plugins to use of a service – Servicecomb-Java-Chassis #678). We realize that authors properly provide clarifications for questions/opinions of reviewers. The reviewing discussion in non-refactoring-inducing PRs usually fits a pattern: everything begins with a review comment suggesting some change to the code, followed by direct arguments from the author to refute such a suggestion. As a result, the author's view prevailed (Figure 5.10).



aledsage reviewed on 7 Nov 2016

camp/camp-brooklyn/src/test/java/org/apache/brooklyn/camp/brooklyn/AbstractYamlTest.java

```
+        if (!ru.doesUrlExist(yamlFileName)) {
+            if (ru.doesUrlExist(Urls.mergePaths(getClass().getPack
+                // look in package-specific folder if not found at
+                yamlFileName = Urls.mergePaths(getClass().getPacka
```

aledsage on 7 Nov 2016

Very minor (feel free to ignore). If it still can't be found, I'd prefer the final exception to give the original `yamlFileName` rather than the package-specific name. With this change, it will throw the exception in `run.getResourceAsString` with the transformed value.

ahgittin on 7 Nov 2016

the package-specific name is only used if the original path **isn't** found *and* the package-specific one **is** -- in which case i think it's correct to use the package-specific name in the error

aledsage on 7 Nov 2016

Ah, you're absolutely right - I missed that second `ru.doesUrlExist` check.

Figure 5.10: A typical discussion in non-refactoring-inducing PRs, from Apache Brooklyn-Server PR #411

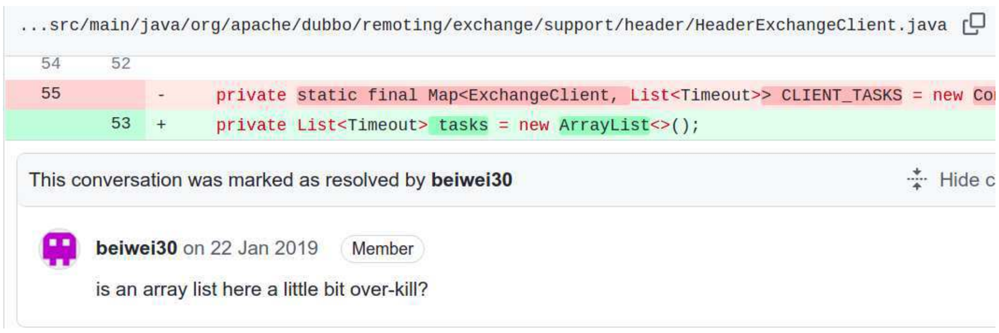
- **Structure of the content**. We observed that, usually, review comments are more polite in refactoring-inducing PRs than in non-refactoring-inducing PRs. They consist of sentences that incorporate expressions like "*Maybe we should ...*" and "*How about ...?*", so triggering the authors to think better concerning the code reviewed, which tends to result in refactoring. Furthermore, we identify a lot of review comments that impose a change in non-refactoring-inducing PRs by using expressions such as "*... should be ...*", which tends to have no effect.
- **Experience of reviewers**. By counting the number of contributions of authors and reviewers (Appendix D, Table D.7), we realized that review comments are submitted by reviewers more experienced than authors in refactoring-inducing PRs, while the opposite happens in non-refactoring-inducing PRs. Accordingly, the experience of authors of non-refactoring-inducing PRs reflects in both characteristics of their code and their ability to provide clarifications and win discussions; whereas the experience of reviewers in refactoring-inducing PRs reflects in precise review comments that may induce or inspire authors to refactor the code. Moreover, we analyzed the experience of authors and reviewers (number of con-

tributions) in the three subgroups of refactoring-inducing PRs (with refactoring led by the author, with refactorings induced by code review, and with refactorings both led by the author and induced by code review). We noticed that, in all subgroups, the reviewers are more experienced (in the number of contributions) than the authors (Appendix D, Table D.9). Thus, based on Finding 2, we conjecture that less problem-prone code seems to give origin to non-refactoring-inducing PRs more often, which can be partially explained by the experience of their authors.

Finding 6: The addressed issues and content structure of review comments (polite and precise), usually leveraged due to a higher experience (inferred from the number of contributions) of a reviewer concerning an author, are motivating factors behind refactoring-inducing PRs.

In practice, precise and polite review comments tend to induce refactoring edits because they shed light on code issues of significant scope, therefore causing authors to embrace refactoring suggestions or get inspired by these suggestions. In this context, polite denotes review comments that make suggestions rather than impositions, while precise expresses opposition to uncertain review comments. Thus, the benefits of refactoring the code become apparent already in code review time, either through a direct suggestion of refactoring (Figure 5.11), a rationale (Figure 5.12), or a warning on good development practices (Figure 5.13). In those examples, the review comments trigger a new authors' perception, causing their agreement with the suggestions. Thus, in such a scenario, the experience of the PR reviewer(s) became crucial.

Therefore, our analysis suggests there is a strong relation between code review and refactoring edits because polite and precise review comments trigger refactorings at the PR level, thus corroborating with previous works regarding the effectiveness of code review on refactoring the code [112; 109].



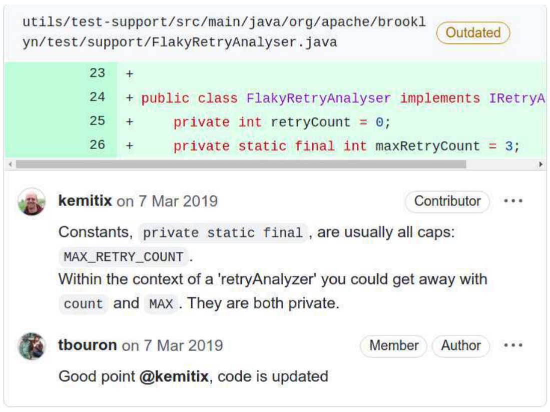
```
...src/main/java/org/apache/dubbo/remoting/exchange/support/header/HeaderExchangeClient.java
54  52
55      -     private static final Map<ExchangeClient, List<Timeout>> CLIENT_TASKS = new Co
    53  +     private List<Timeout> tasks = new ArrayList<>();
```

This conversation was marked as resolved by **beiwei30**

beiwei30 on 22 Jan 2019 Member

is an array list here a little bit over-kill?

Figure 5.11: A direct review comment, from Apache Dubbo PR #3299, which induced a *Change Variable Type* edit



```
utils/test-support/src/main/java/org/apache/brookl
yn/test/support/FlakyRetryAnalyser.java
23  +
24  + public class FlakyRetryAnalyser implements IRetryA
25  +     private int retryCount = 0;
26  +     private static final int maxRetryCount = 3;
```

kemitix on 7 Mar 2019 Contributor

Constants, `private static final`, are usually all caps: `MAX_RETRY_COUNT`.
Within the context of a 'retryAnalyzer' you could get away with `count` and `MAX`. They are both private.

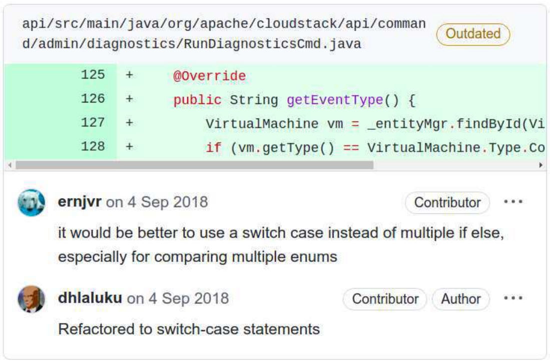
tbouron on 7 Mar 2019 Member Author

Good point **@kemitix**, code is updated

Figure 5.12: A review comment providing a reason (code conventions), from Apache Brooklyn-Server PR #1049, which induced a *Split Attribute* edit

5.2.4 How Do Reviewers Suggest Refactorings in Refactoring-Inducing PRs?

Suggestions of refactoring are usually polite and precise, being well-understood by the PR authors in refactoring-inducing PRs. Those suggestions use expressions such as "***Maybe*** *introduce variable long expiredCheckpoint = 1L; and pass into StateRestorer to make this clearer?*" in Kafka #5946 and "*So to be more "future proof", **can we** send in the filterConfig and move the boolean into the createSSLContext? This way we can use other filterconfig items later if needed to configure the SSLContext*" in Knox #74. We conjecture that the experience of reviewers against authors in refactoring-inducing PRs might explain such a pattern.



api/src/main/java/org/apache/cloudstack/api/command/admin/diagnostics/RunDiagnosticsCmd.java Outdated

```
+    @Override
+    public String getEventType() {
+        VirtualMachine vm = _entityMgr.findById(Vi
+        if (vm.getType() == VirtualMachine.Type.Co
```

ernjvr on 4 Sep 2018 Contributor

it would be better to use a switch case instead of multiple if else, especially for comparing multiple enums

dhlaluku on 4 Sep 2018 Contributor Author

Refactored to switch-case statements

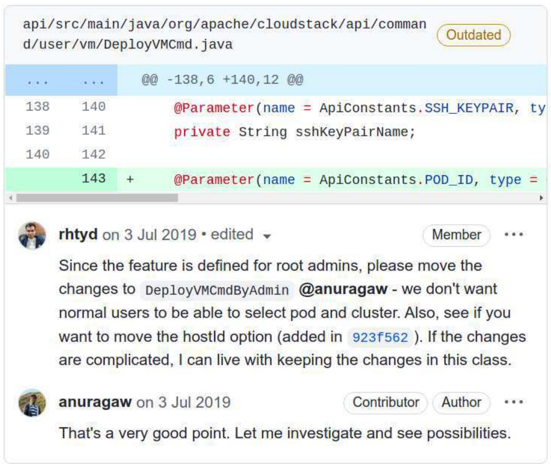
Figure 5.13: A warning on a conditional to deal with enumerations, from Apache Cloudstack PR #2833, which induced a *Change Variable Type* and a *Rename Variable* edits

Also, sometimes, the reviewers provide a rationale for their suggestions, such as security-related issues in Cloudstack #3454 (Figure 5.14). In brief, the purpose of a rationale is likely to let the author know that an adjustment is not necessary for the code operation but to bring more benefits. Review comments, in the form of warning on good practices of development, may induce refactoring and provide explanations and examples (using structures of the code under analysis, e.g., a class, a method) of how to deal with problematic points (e.g., single responsibility of methods in Incubator-Iceberg #119, Figure 5.15).

Finding 7: Reviewers tend to make polite and precise suggestions of refactoring.

5.2.5 Do Suggestions of Refactoring Justify the Reasons?

Reviewers provide reasons that induced refactoring edits in 26/65 (40.0%) of the refactoring-inducing PRs. Specifically, we identified reasons in 12/26 (46.1%), 2/26 (7.8%), and 12/26 (46.1%) of the refactoring-inducing PRs with refactorings led by the authors, with refactorings induced by code review, and with refactorings both led by author and induced by code review, respectively. Nevertheless, the type of refactoring is not explicitly pointed, except for a few *Rename*, *Move*, and *Extract* instances, as it occurs in Cloudstack #2071 (Figure 5.3). In addition, even suggestions of low-level refactorings may provide reasons, as in Kafka #5946 (Figure 5.16). Indeed, only review


api/src/main/java/org/apache/cloudstack/api/command/user/vm/DeployVMCmd.java

```
@@ -138,6 +140,12 @@
      @Parameter(name = ApiConstants.SSH_KEYPAIR, ty
      private String sshKeyPairName;

+     @Parameter(name = ApiConstants.POD_ID, type =
```

rhtyd on 3 Jul 2019 • edited

Since the feature is defined for root admins, please move the changes to `DeployVMCmdByAdmin` **@anuragaw** - we don't want normal users to be able to select pod and cluster. Also, see if you want to move the hostId option (added in `923f562`). If the changes are complicated, I can live with keeping the changes in this class.

anuragaw on 3 Jul 2019

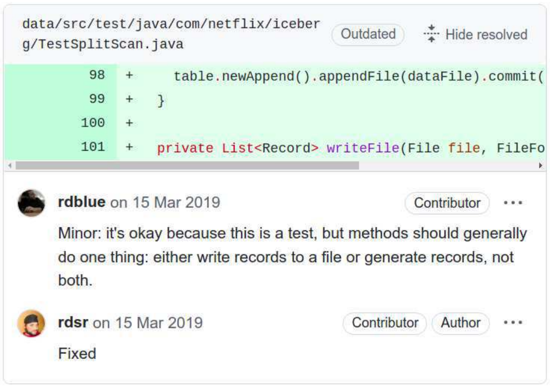
That's a very good point. Let me investigate and see possibilities.

Figure 5.14: A review comment providing a reason (secure-related issues), from Apache Cloudstack PR #3454, which induced two *Push Down Attribute* and two *Push Down Method* edits

comments in Cloudstack #2071, Flink #7945, Incubator-Iceberg #119, Incubator-Iceberg #183 submit reasons for high-level refactoring instances (Figure 5.17).

Reasons are contextualized sentences regarding problematic situations, structured like alerts (e.g., constants format in Brooklyn-Server #1049, Figure 5.12), sometimes accompanied by examples (e.g., code's structure under analysis in Kafka #5194). In other cases, reasons begin with a question on code logic ("*Is this type over-kill?*", "*Can we ... ?*") assisted by examples (e.g., questioning a class functionality in Flink #7945, Figure 5.17). In other cases, reviewers provide a direct explanation (e.g., informing about a class useless in Dubbo #2279, Figure 5.18). This result emphasizes code review as a traditional practice, conducted by expert reviewers, in Apache projects [122]. Furthermore, we identified no suggestion of refactoring edits providing a reason for non-refactoring-inducing PRs.

Finding 8: Reviewers tend to provide reasons, occasionally supported by examples, to elucidate the benefits of refactoring.


data/src/test/java/com/netflix/iceberg/TestSplitScan.java

```
+    table.newAppend().appendFile(dataFile).commit(
+  }
+
+  private List<Record> writeFile(File file, FileFo
```

rdblue on 15 Mar 2019

Minor: it's okay because this is a test, but methods should generally do one thing: either write records to a file or generate records, not both.

rdsr on 15 Mar 2019

Fixed

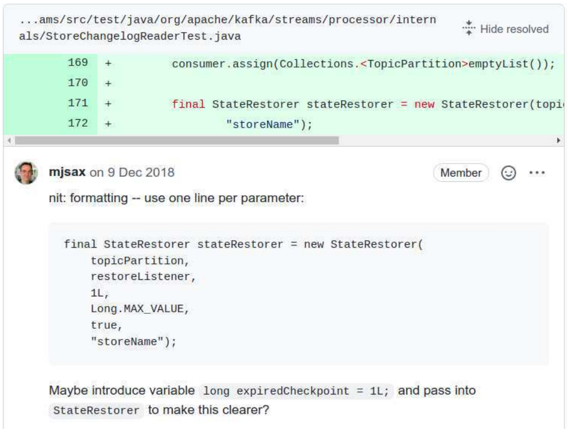
Figure 5.15: A warning on responsibility of methods, from Apache Incubator-Iceberg PR #119, which induced a *Change Return Type*, a *Rename Method*, a *Change Parameter Type*, and a *Rename Parameter* edits

5.2.6 What Is the Relationship Between Suggestions and Actual Refactorings in Refactoring-Inducing PRs?

When the reviewers provide direct suggestions of refactoring, refactoring edits are performed. We identify such a pattern in 43/43 (100%) of refactoring-inducing PRs, in which code review induced refactoring edits. Those direct suggestions may include explanations using examples in the code under analysis (e.g., suggesting a rename in Flink #8620, Figure 5.19) and reasons (e.g., a large class body in Flink #7945, Figure 5.17).

Nevertheless, when there is space for discussion, refactorings may not be done. To reinforce such emerged pattern, as we previously argued, the proportion of reviewing discussion is higher in non-refactoring-inducing PRs than in refactoring-inducing PRs. In particular, we identified a few discussions due to uncertain review comments (e.g., in Flink #9143, Figure 5.6), and to impositions of change consisting of embedded code (e.g., a reviewer suggests a method change, using "*should we ...*", by indicating an embedded code in Dubbo #4870, Figure 5.20). In both cases, the authors presented arguments leading to no subsequent change.

Finding 9: Direct suggestions of refactoring tend to be embraced by the PR authors.


```
...ams/src/test/java/org/apache/kafka/streams/processor/intern
als/StoreChangelogReaderTest.java
```

```
+        consumer.assign(Collections.<TopicPartition>emptyList());
+
+        final StateRestorer stateRestorer = new StateRestorer(topi
+                "storeName");
```

mjsax on 9 Dec 2018

nit: formatting -- use one line per parameter:

```
final StateRestorer stateRestorer = new StateRestorer(
    topicPartition,
    restoreListener,
    1L,
    Long.MAX_VALUE,
    true,
    "storeName");
```

Maybe introduce variable `long expiredCheckpoint = 1L;` and pass into `StateRestorer` to make this clearer?

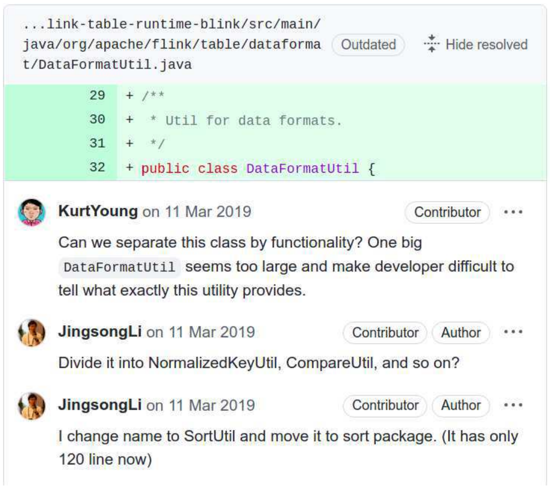
Figure 5.16: A suggestion of a refactoring, from Apache Kafka PR #5946, which induced an *Extract Variable* edit

5.3 Implications and Guidelines

By characterizing code review in refactoring-inducing PRs, we can potentially advance the understanding of code reviewing at the PR level while assisting researchers, practitioners, and tool builders in such a scenario.

Researchers: To the best of our knowledge, no prior MCR studies investigated PRs in light of our refactoring-inducing definition, thus exploring the refactorings performed specifically during the code review time (subsequent commits). In this context, Finding 1 sheds light on a novel view for works concerning pull-based development. Based on our findings, we provide directions for researchers towards three dimensions.

First, our study points out a few motivating factors behind refactoring-inducing PRs (Findings 2 and 6). Finding 2 is an expected result since we suppose that more experienced developers implement less problem-prone code. Nevertheless, Finding 6 indicates distinct characteristics of code reviewing in refactoring-inducing and non-refactoring-inducing PRs. Thus, reviewers of refactoring-inducing PRs are usually more experienced than their authors then such a pattern assists the knowledge transfer and learning of a less experienced author. This result corroborates Bacchelli and Bird that, when defining MCR, emphasize knowledge transfer as its relevant characteristic [33]. However, the authors of non-refactoring-inducing PRs may miss opportunities of improving their code quality due to reviewer limitations (e.g., less experienced),


...link-table-runtime-blink/src/main/java/org/apache/flink/table/dataformat/DataFormatUtil.java

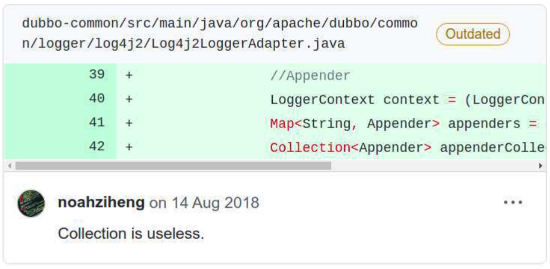
```
+ /**
+  * Util for data formats.
+  */
+ public class DataFormatUtil {
```

KurtYoung on 11 Mar 2019
Can we separate this class by functionality? One big `DataFormatUtil` seems too large and make developer difficult to tell what exactly this utility provides.

JingsongLi on 11 Mar 2019
Divide it into NormalizedKeyUtil, CompareUtil, and so on?

JingsongLi on 11 Mar 2019
I change name to SortUtil and move it to sort package. (It has only 120 line now)

Figure 5.17: A review comment providing a reason (long class body), from Apache Flink PR #7945, which induced two *Move and Rename Class* edits


dubbo-common/src/main/java/org/apache/dubbo/common/logger/log4j2/Log4j2LoggerAdapter.java

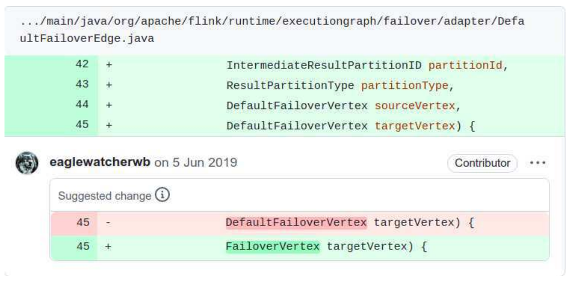
```
+                    //Appender
+                    LoggerContext context = (LoggerCon
+                    Map<String, Appender> appenders =
+                    Collection<Appender> appenderColle
```

noahziheng on 14 Aug 2018
Collection is useless.

Figure 5.18: A warning on a class useless, from Apache Dubbo PR #2279, which induced a *Change Variable Type* and a *Rename Variable* edits

although such a scenario provides a learning opportunity to a less experienced reviewer. This result reinforces other works on code review practice, such as Sadowski et al. that discovered education aspects as one of the main motivations behind code review at Google [126]. Therefore, we recommend future research to explore strategies to support an effective code review participation of authors and reviewers, knowledge transfer, and awareness of the changes among team members at the PR level; for instance, by specifying requirements for reviewer recommendation in line with our findings.

Second, Findings 3–5 reinforce the need for future (qualitative) research on MCR with PRs, distinguishing refactoring-inducing and non-refactoring-inducing PRs or considering their different characteristics when sampling PRs, as we claimed in Chapter 4,



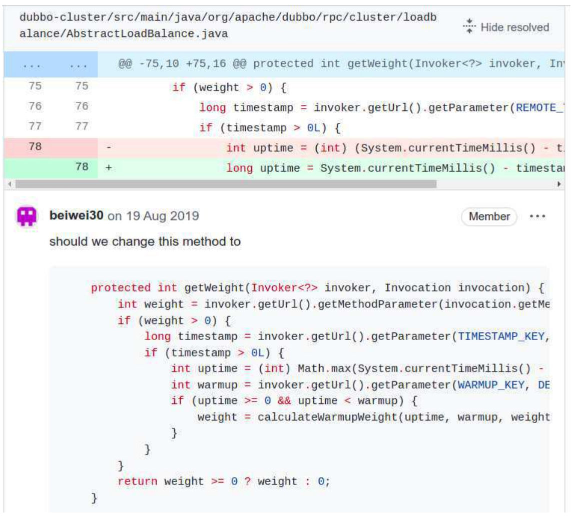
```
.../main/java/org/apache/flink/runtime/executiongraph/failover/adapter/Defa
ultFailoverEdge.java
+                IntermediateResultPartitionID partitionId,
+                ResultPartitionType partitionType,
+                DefaultFailoverVertex sourceVertex,
+                DefaultFailoverVertex targetVertex) {
```

eaglewatcherwb on 5 Jun 2019 Contributor

Suggested change

```
-                DefaultFailoverVertex targetVertex) {
+                FailoverVertex targetVertex) {
```

Figure 5.19: A review comment, from Apache Flink PR #8620, which induced a *Rename Method* edit



```
dubbo-cluster/src/main/java/org/apache/dubbo/rpc/cluster/loadb
alance/AbstractLoadBalance.java
@@ -75,10 +75,16 @@ protected int getWeight(Invoker<?> invoker, In
        if (weight > 0) {
            long timestamp = invoker.getUrl().getParameter(REMOTE_
            if (timestamp > 0L) {
-                int uptime = (int) (System.currentTimeMillis() - t
+                long uptime = System.currentTimeMillis() - timesta
```

beiwei30 on 19 Aug 2019 Member

should we change this method to

```
protected int getWeight(Invoker<?> invoker, Invocation invocation) {
    int weight = invoker.getUrl().getMethodParameter(invocation.getMe
    if (weight > 0) {
        long timestamp = invoker.getUrl().getParameter(TIMESTAMP_KEY,
        if (timestamp > 0L) {
            int uptime = (int) Math.max(System.currentTimeMillis() -
            int warmup = invoker.getUrl().getParameter(WARMUP_KEY, DE
            if (uptime >= 0 && uptime < warmup) {
                weight = calculateWarmupWeight(uptime, warmup, weight
            }
        }
    }
    return weight >= 0 ? weight : 0;
}
```

Figure 5.20: A review comment including an embedded code from Apache Dubbo PR #4870

Section 4.3. Since we found patterns in the context of review comments, future studies may leverage supervised machine learning techniques to advance in understanding code review practice. We also conjecture that using embedded code in review comments may be understood, by PR authors, as a ready-made solution. Such a structure is not welcome by the authors; maybe, this could affect code authorship (in the developer's perception). For further investigation, the design of future studies may consider surveys with PRs authors.

Third, we strongly recommend reproduction studies intending to confirm or refute our findings by considering other GitHub projects and pull-based platforms. We provide a reproduction kit as support in this direction [22].

Practitioners: Based on our Findings 6–8, we provide a few guidelines for reviewers composing valuable review comments towards code refactoring:

- Be polite when suggesting refactorings, using expressions such as "*can we ...?*", "*maybe ...*";
- Be precise in both suggestions of refactoring and reasons[6];
- Use the structure of the author's code (e.g., classes, methods, etc.) when providing explanations. This usually leads to thought by the code author(s)[7];
- Be direct when questioning the code logic[8], employing expressions that use the first person plural such as "*should we ...?*";
- Avoid imposing any change, mainly by providing embedded code.[9]

Those guidelines may improve reviewers' productivity when recommending patterns to compose more effective review comments. Thus, reviewing discussions may be reduced. As a consequence, the time to merge PRs may also decrease, as argued by Gousios et al. when investigating factors influencing the pull-based development effectiveness [68] – they found empirical evidence that code review affects time to merge a PR. In addition, those more effective review comments represent a rich learning opportunity for less experienced authors.

Tool builders: Supported by empirical evidence that integrating static analysis tools can improve the quality of code review [34], we suggest the implementation of checkers of review comments as a feature available at code review boards. To emphasize the feasibility of this suggestion, Rahman et al. developed a machine learning prediction model for assisting developers when formulating code review comments, based on the text and reviewer experience. Thus, tool builders may leverage our guidelines in the sense of assisting more effective code review at the PR level since PRs authors may miss opportunities to improve code due to not well-structured review comments. Although

[6]e.g., `https://git.io/JMyJL`.

[7]e.g., `https://git.io/JMyLu`.

[8]e.g., `https://git.io/JMyUL`.

[9]Examples available at `https://git.io/JMyTE` and `https://git.io/JMyt6`.

such a checker deals with natural language, we already identified a few terms and expressions to be avoided (e.g., "not sure if ...") and others that proved to be well accepted (e.g., "How about ...?") in code reviews, which can be used in preliminary tools.

5.4 Limitations

To deal with the issues of **validity and reliability** of our study, we propose the countermeasures introduced in Table 5.11. Because we consider data obtained in line with our data mining design, this study relies on its countermeasures and threats to validity and reliability (Chapter 3, Section 3.2).

Table 5.11: Validity and reliability countermeasures for characterizing code review in refactoring-inducing PRs

Type	***Description***
Internal validity	A design proposal to guide the study
Construct validity	Regular revision of the study design by supervisors
	Establishment of a chain of evidence based on data analyzed by three developers
	Manual validation of refactoring edits in refactoring-inducing and non-refactoring-inducing PRs
	Establishment of a chain of evidence based on data analyzed by three developers
Reliability	Effort towards clarifying the data analysis procedures to enable replications

Note that we elaborated the research design supported by guidelines [55; 96], whereas we employed special attention for sampling representative PRs in each round of analysis. Accordingly, each purposive sample comprises one refactoring-inducing PR from the diversity of settings available at that moment. For instance, the 13 refactoring-inducing PRs of sample 3 (Round 3) represent a set of available compositions of high-level refactorings found in 36 refactoring-inducing PRs, having high-level refactorings, at the beginning of the round.

Moreover, for each round of analyses, we used randomness to select non-refactoring-inducing PRs, having an equal median of the number of review comments in the set of refactoring-inducing PRs, intending a fair comparison for answering RQ_1 and RQ_2. We considered the median value because review comments present a non-normal distribution in our whole sample (Appendix B). Even so, there are risks of threats to internal validity due to any non-previously identified deficiencies in our research design.

We made an effort to establish a chain of evidence for the data interpretation and systematically explain the procedures, decisions, and obtained results in each design step. In addition, we propose a few actions intending to increase validity, including a sanity check of refactoring edits (checking false positives and false negatives in 118 PRs) and a data analysis performed by three researchers in order to reduce researcher bias.

The subjective nature of our qualitative study does not allow generalizations. However, we speculate that we can potentially extend our findings to cases that have common characteristics with Apache's projects, that is, other OSS projects that follow a geographically distributed development [152] and are aligned to "the Apache way" principles [17].

We systematically structured all procedures to deal with reliability issues, thus providing a reproduction kit to enable replications, publicly available at [22].

5.5 Concluding Remarks

In this qualitative study, we explored 118 PRs, the experience of their reviewers, 923 review comments, and 366 refactoring edits aiming at characterizing code review in refactoring-inducing PRs. Our results reveal motivating factors behind refactoring-inducing PRs, technical aspects of the structure of review comments, and guidelines for a more productive code reviewing.

By considering the samples of pull requests used in our previous studies (Chapters 4 and 5), we designed a mixed-methods study for characterizing refactoring edits in refactoring-inducing PRs, described in the next chapter.

Chapter 6

Characterizing Refactoring Edits in Refactoring-Inducing Pull Requests

In this chapter, we detail a characterization study of refactoring edits in refactoring-inducing PRs. For that, we investigated two samples: 449 refactoring-inducing PRs for quantitative analysis (*sample 1*, Chapter 3) and 65 ones for qualitative analysis (*four purposive samples*, Chapter 5). That 65 refactoring-inducing PRs are a subsample from the 449 ones. Note that our previous studies considered the refactoring-inducing definition in order to compare refactoring-inducing and non-refactoring-inducing PRs (Chapter 4) and characterize code review in refactoring-inducing PRs (Chapter 5).

Thus, we describe the research design (Section 6.1); then, we present the results and discuss them (Section 6.2). Next, we argue a few implications (Section 6.3) and limitations (Section 6.4).

6.1 Research Design

This study explores refactoring-related aspects at the PR level, aiming to characterize refactoring edits in Apache's refactoring-inducing PRs. In this sense, we formulated the following **research questions**:

- RQ_1: What types of refactoring edits often take place in PRs? First, we examined the types of refactoring edits, detected in refactoring-inducing PRs, intending to understand the practice of code refactoring at the PR level.

- RQ$_2$: How are the refactoring edits characterized? Then, we sought and analyzed technical aspects of refactoring edits in refactoring-inducing PRs, to compose a more comprehensive characterization of refactorings at the PR level. In this context, the level of refactorings (low-level or high-level) is one of the examined aspects.

As shown in Figure 6.1, our **study design** consists of a mixed-methods analysis. First, we analyze quantitative data from the *refactorings dataset*, mined from Apache's Java repositories in Github (*sample 1*, Chapter 3). Then, we analyze qualitative data using the *worksheets* derived from examining four purposive samples in order to characterize code review in refactoring-inducing PRs (*Merging of results*, Chapter 4).

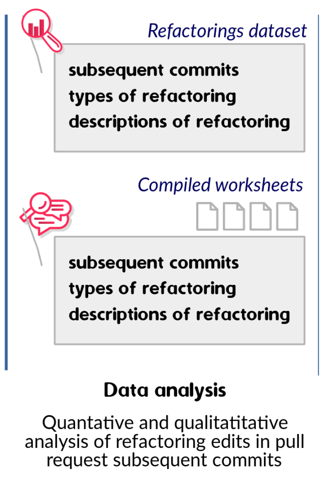

Refactorings dataset
subsequent commits
types of refactoring
descriptions of refactoring
Compiled worksheets
subsequent commits
types of refactoring
descriptions of refactoring
Data analysis
Quantative and qualitatitative analysis of refactoring edits in pull request subsequent commits

Figure 6.1: An overview of the characterization study of refactoring edits in refactoring-inducing PRs

Our quantitative analysis provides additional results and discussion about refactoring edits performed in light of refactoring-inducement, discussing subjects such as types of refactorings induced by code review or led by the PR authors, for instance.

Given that, we performed both data analyses to answer each research question. We ran the quantitative data analysis on the *refactorings dataset* (sample summarized in Table 6.1) to explore the number of refactoring edits, types of refactorings, and frequency of low-level and high-level refactorings by PR. We developed a Python script[1]

[1]Available at `https://git.io/JDY9A`.

to perform such an analysis and made it available in our reproduction kit [22].

Table 6.1: Sample of refactoring-inducing PRs for the quantitative analysis

Number of repositories	***Number of pull requests***	***Number of subsequent commits***	***Number of detected refactorings***
50	449	1,563	2,104

As we argued in Chapter 3, we did not validate the refactoring edits in the *refactoring dataset*. Then, we carried out a qualitative analysis based on a validated sample of refactoring edits in 65 refactoring-inducing PRs (summarized in Table 6.2), thus providing further investigation of the answers achieved by quantitative data analysis for our research questions.

Table 6.2: Sample of refactoring-inducing PRs for the qualitative analysis

Number of repositories	***Number of pull requests***	***Number of subsequent commits***	***Number of detected refactorings***
29	65	346	366

The examined types of refactoring edits consider the catalog of 40 types detectable by RefactoringMiner 1.0 in September 2019 (Table 3.5, Chapter 3) – a version closest to version 2.0 [145]. It is worth remembering that we classified those types of refactoring in line with technical descriptions by Fowler [60] in low-level and high-level refactorings. We considered such levels intending to investigate the practice of refactoring that configure more complex changes (or that changes the code design) in relation to less complex ones at the PR level. Also, since our quantitative data analysis relies on RefactoringMiner detection, we answer the research questions in light of the mechanisms specified by RefactoringMiner's developers for each type of refactoring. Thus, we recognize a potential inflated number and types of refactoring edits, as argued in Chapter 3, Section 3.2.

6.2 Results and Discussion

To address each research question, we present the results from the quantitative analysis accompanied by further qualitative examinations, then we discuss them.

6.2.1 What Types of Refactoring Edits often Take Place in PRs?

In our sample of 449 refactoring-inducing PRs, RefactoringMiner detected 34 distinct types of refactorings (Table 6.3). As we can see, *Rename Method* edits are the most frequent, also found by Vassalo et al. when investigating refactoring practice in Apache projects [147]. Given this diversity of types of refactoring edits, we understand that distinct subjects are addressed by refactorings at the PR level.

Table 6.3: Types of refactoring edits detected by RefactoringMiner 1.0 in the 449 refactoring-inducing PRs

Type of refactoring	***Number of detections***	***Proportion (%)***
Rename Method	355	16.87
Change Variable Type	209	9.93
Rename Variable	199	9.46
Extract Method	172	8.17
Change Parameter Type	156	7.41
Rename Parameter	155	7.37
Change Attribute Type	118	5.61
Change Return Type	114	5.42
Rename Attribute	105	4.99
Extract Variable	95	4.51
Rename Class	64	3.04
Move Attribute	44	2.09
Move Class	41	1.95
Extract And Move Method	34	1.61
Inline Variable	34	1.61

Continued on next page

Table 6.3 – continued from previous page

Type of refactoring	***Number of detections***	***Proportion (%)***
Inline Method	30	1.42
Move Method	26	1.23
Replace Variable With Attribute	26	1.23
Pull Up Method	23	1.09
Extract Attribute	18	0.85
Push Down Method	16	0.76
Move And Rename Class	12	0.57
Parameterize Variable	10	0.47
Extract Interface	10	0.47
Push Down Attribute	10	0.47
Pull Up Attribute	8	0.40
Extract Class	7	0.35
Extract Superclass	4	0.20
Change Package (Rename)	3	0.15
Merge Parameter	2	0.10
Split Parameter	1	0.05
Move Source Folder	1	0.05
Merge Variable	1	0.05
Split Attribute	1	0.05
Total	*2,104*	*100.0*

Kinds of refactoring edits

When grouping the types of refactorings by kind (Table 6.4), *Rename* edits are the most common. As a whole, we realized that *Rename*, *Change Type*, *Extract*, and *Move* edits represent a significant number of refactorings (almost 90%) in our sample, what suggests development efforts concerning code readability [110], understandability [147], maintainability [142], and cohesion and coupling [106]. Moreover, we recognize that high-level refactorings are defined in terms of low-level ones [60].

Table 6.4: Kinds of refactoring edits detected by RefactoringMiner 1.0 in the 449 refactoring-inducing PRs

Kind of refactoring	***Number of detections***	***Proportion (%)***
Rename	878	41.73
Change Type	597	28.37
Extract	306	14.54
Move	112	5.32
Inline	64	3.04
Extract and Move	34	1.62
Pull Up	31	1.47
Push Down	26	1.24
Replace	26	1.24
Move and Rename	12	0.57
Parameterize	10	0.47
Change Package	3	0.15
Merge	3	0.15
Split	2	0.09
Total	*2,104*	*100.0*

Change Type edits are not in the Fowler's catalog [61; 61], but they are detectable by RefactoringMiner [145] and available at IntelliJ IDEA, as an automatic refactoring coined as *Type Migration* [24]. *Change Type* edits consist of changing a class's member type. For instance, suppose that a developer implemented the Java source code 6.1 and wants to change the type of the variable *age* to better accommodate input values. Then, the developer can refactor the code applying a *Change Variable Type* (int $\rightarrow$ byte), as demonstrated in source code 6.2. *Change Type* refactorings are essential for class/library migration because it contributes to code maintainability [142].

```java
public void simpleMethod() {
    int age;
  //do something
}
```

Listing 6.1: A simple Java method

```java
public void simpleMethod() {
    byte age;
  //do something
}
```

Listing 6.2: A simple Java refactored method

Refactoring-inducing PRs consisting of a single type of refactoring

RefactoringMiner detected a single type of refactoring in 136/449 (30.3%) of refactoring-inducing PRs that, in turn, comprise 204/2,104 (9.7%) of refactoring edits in our sample. When examining the most frequent types of refactoring in those 136 PRs, we identified 22 distinct types of refactoring, among which *Rename Method*, *Change Variable Type*, *Rename Variable*, and *Extract Method* remain in the top 5 (Table 6.5).

Table 6.5: Types of refactoring edits detected by RefactoringMiner 1.0 in 136 refactoring-inducing PRs consisting of a single type of refactoring

Type of refactoring	***Number of detections***	***Proportion (%)***
Rename Method	77	37.74
Extract Method	24	11.76
Extract Variable	18	8.82
Rename Variable	15	7.35
Change Variable Type	13	6.37
Rename Attribute	8	3.93
Change Return Type	7	3.44
Rename Class	7	3.44
Move Class	6	2.94
Rename Parameter	5	2.45
Change Parameter Type	4	1.96
Extract Attribute	4	1.96
Move Attribute	3	1.47
Change Attribute Type	2	0.98

Continued on next page

Table 6.5 – continued from previous page

Type of refactoring	***Number of detections***	***Proportion (%)***
Move Method	2	0.98
Move And Rename Class	2	0.98
Inline Variable	2	0.98
Replace Variable With Attribute	1	0.49
Split Attribute	1	0.49
Push Down Method	1	0.49
Extract Interface	1	0.49
Pull Up Attribute	1	0.49
Total	*204*	*100.0*

Regarding kinds of refactoring, we realize that the top 5 (*Rename*, *Change Type*, *Extract*, *Move*, and *Inline*) remain in the 136 refactoring-inducing PRs (Table 6.6). Therefore, we conclude that even when a PR embraces a single type of refactoring, those edits often reflect concerns on code readability (e.g., a *Rename Class* in Samza #1051), maintainability (e.g., a *Change Variable Type* in Tomee #275), and cohesion (e.g., a *Extract Method* in Servicecomb-Java-Chassis #346).

Table 6.6: Kinds of refactoring edits detected by RefactoringMiner 1.0 in 136 refactoring-inducing PRs consisting of a single type of refactoring

Kind of refactoring	***Number of detections***	***Proportion (%)***
Rename	112	54.91
Extract	47	23.04
Change Type	26	12.74
Move	11	5.39
Inline	2	0.98
Move and Rename	2	0.98
Replace	1	0.49
Split	1	0.49

Continued on next page

Table 6.6 – continued from previous page

Kind of refactoring	***Number of detections***	***Proportion (%)***
Pull Up	1	0.49
Push Down	1	0.49
Total	*204*	*100.0*

Finding 1: The most common refactorings are *Rename*, *Change Type*, *Extract* and *Move* edits. We believe that code improvements concerning readability, maintainability, cohesion, and coupling are frequent at the PR level. This is valid even when a PR addresses a single type of refactoring.

In terms of most typical refactorings, our result corroborates the findings of Murphy-Hill and colleagues, which found *Rename* refactorings as the most frequent when studying the refactoring practices in two other OSS projects [100]; however, it differs from recent studies. Pantiuchina and colleagues observed *Extract* refactorings as the most common when exploring motivations behind refactoring edits at PR level in 150 projects in GitHub [112]. Whereas, Paixão and colleagues identified *Extract Method* and *Move Method* as the most typical types when investigating refactorings over reviews on Gerrit [109]. It is worth mentioning that the catalog of refactorings, considered by those studies, differs from the ones detectable by RefactoringMiner [145]; mainly concerning *Change Type* refactorings.

Additionally, as shown in Figure 6.2, the refactoring-inducing PRs present 3.5 on average (SD = 3.3) and 2 on median (IQR = 3) distinct types of refactoring edits. Thus, most of them address refactoring edits of different types. These results are aligned with findings from Brito and colleagues, who identified that most refactorings are composed of more than one type of refactoring over commits [47].

Validated refactoring edits in refactoring-inducing PRs

When qualitatively studying our sample of 65 refactoring-inducing PRs, firstly, we observed that a single review comment may induce multiple refactorings. The number of refactoring edits in a PR may depend on the context (e.g., number of file

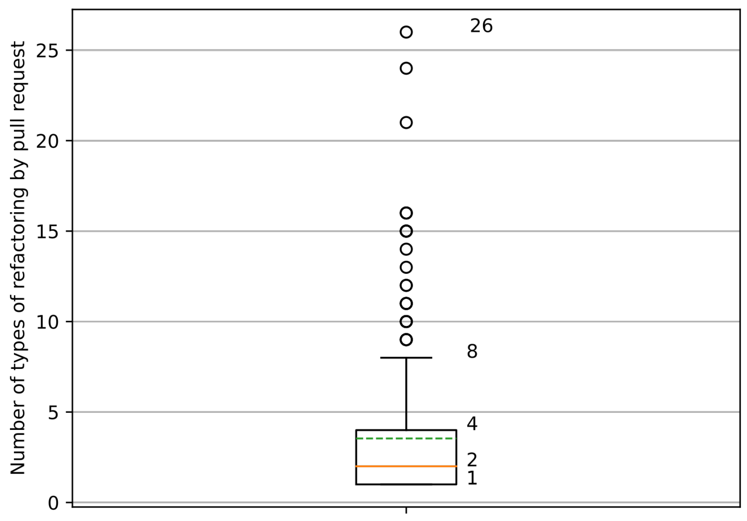
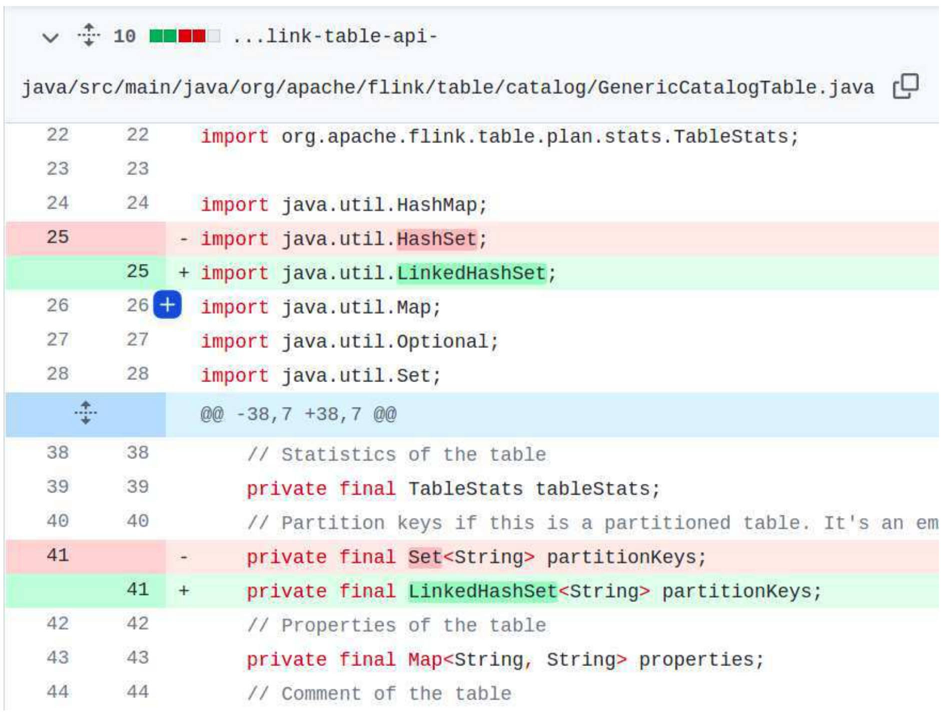


Figure 6.2: Number of types of refactoring detected by refactoring-inducing PR

changes and code churn) and how RefactoringMiner defines its refactoring mechanisms. For instance, the review comment "*New private method with obfuscated name should have some comment describing what it's supposed to do.*" induced seven *Rename Method* edits in Fluo #837, so renaming seven distinct methods. The review comment "*just a note, SSL is outdated, the official name would be TLS, name the package tls?*" induced a *Change Package* (*Rename*) in Cloudstack #2071 – RefactoringMiner also detected two *Move Class* edits because such a package contained the classes. Moreover, we realized that *Change Type* edits may be due to other refactorings. To illustrate, consider a *Change Attribute Type* edit in Flink #8222 (Figure 6.3). Such refactoring is accompanied by a *Change Return Type* in createPartitionKeys() in GenericInMemoryCatalogTest, a *Change Parameter Type* in GenericCatalogTable (TableSchema, TableStats, LinkedHashSet<String>, Map<String,String>) in GenericCatalogTable, and a *Change Parameter Type* in createPartitionedTable(TableSchema, LinkedHashSet<String>, Map<String, String>, String) in CatalogTestUtil.

We also identified refactoring-inducing PRs that configure a counter-intuitive scenario: when a refactoring edit produces a reviewer's comment. For instance, in Kafka #5194 (Figure 6.4), a reviewer suggested adding a test case after a *Move Class* of ConfigProvider in the first subsequent commit (induced by code review).



```
10 ...link-table-api-
java/src/main/java/org/apache/flink/table/catalog/GenericCatalogTable.java
  import org.apache.flink.table.plan.stats.TableStats;

  import java.util.HashMap;
- import java.util.HashSet;
+ import java.util.LinkedHashSet;
  import java.util.Map;
  import java.util.Optional;
  import java.util.Set;
@@ -38,7 +38,7 @@
      // Statistics of the table
      private final TableStats tableStats;
      // Partition keys if this is a partitioned table. It's an em
-     private final Set<String> partitionKeys;
+     private final LinkedHashSet<String> partitionKeys;
      // Properties of the table
      private final Map<String, String> properties;
      // Comment of the table
```

Figure 6.3: A *Change Attribute Type* edit in Apache Flink PR #8222

In addition, we also explored the kinds of refactoring edits, so realizing that the top 4 remain in refactoring-inducing PRs (Table 6.7). We observed an almost similar result regardless of how refactoring edits were triggered (led by the authors or induced by code review). Specifically, our purposive sample has 13/65 (20%) of refactoring-inducing PRs comprising high-level refactorings, which explain the number of *Pull Up* and *Move and Rename* edits. In Beam #6261, Dubbo #3654, Flink #8620, and Kafka #4757, we found the presence of floss refactoring to accommodate new tests in the three first ones and a new feature in a class in the last one – all suggested by reviewers. Thereby, we judged the associated refactoring edits as led by the PR authors. Given that, our qualitative analysis reinforces Finding 1 and its discussion.

Table 6.7: Kinds of refactorings in 65 refactoring-inducing PRs (validated refactorings)

Kind of refactorings	*Refactoring-inducement*		Total
	Led by authors	Induced by code review	
Change Type	50	65	115/366 (31.4%)

Continued on next page

Table 6.7 – continued from previous page

Kind of refactoring	*Refactoring-inducement*		Total
	Led by authors	Induced by code review	
Rename	56	45	101/366 (27.6%)
Extract	19	19	38/366 (10.4%)
Move	27	5	32/366 (8.7%)
Extract and Move	18	3	21/366 (5.8%)
Pull Up	6	13	19/366 (5.2%)
Push Down	15	4	19/366 (5.2%)
Replace	2	5	7/366 (1.8%)
Move and Rename	none	7	7/366 (1.8%)
Inline	1	1	2/366 (0.6%)
Split	1	1	2/366 (0.6%)
Parameterize	2	none	2/366 (0.6%)
Merge	1	none	1/366 (0.3%)
Total	*198/366 (54.1%)*	*168/366 (45.6%)*	*366/366 (100%)*

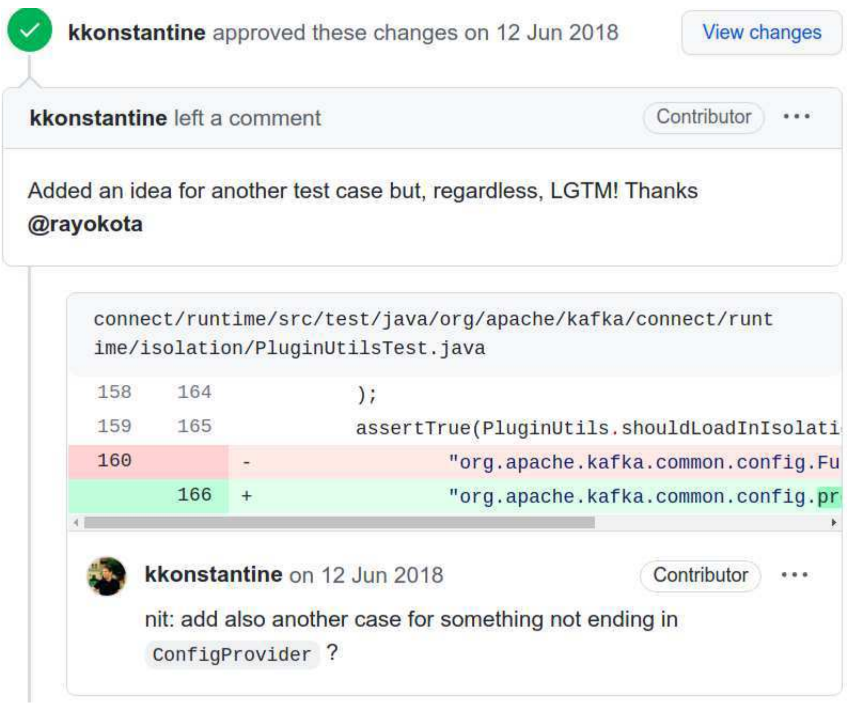
Note that floss refactoring can occur in PR initial commits – what is out of the refactoring-inducing PR definition. For instance, in Flink #7945 that submits adaptative changes (a new feature), RefactoringMiner detected an *Extract Superclass* and *Change Type* edits in the initial commits. Our data analysis did not include them because we concentrated effort on investigating changes over PR subsequent commits.

6.2.2 How are the Refactoring Edits Characterized?

We explored the number of detected refactorings by RefactoringMiner over the PR subsequent commits and the levels of refactoring in order to answer this question.

Refactoring edits over PR subsequent commits

As depicted in Figure 6.5 (a), the histogram of number of subsequent commits is positively skewed, thus a low number of them by PR is quite frequent. Refactoring-inducing PRs have 3.5 on average (SD = 3.1) and 3 on median (IQR = 2) subsequent commits, as shown in Figure 6.5 (b).



kkonstantine approved these changes on 12 Jun 2018

kkonstantine left a comment

Added an idea for another test case but, regardless, LGTM! Thanks @rayokota

```
connect/runtime/src/test/java/org/apache/kafka/connect/runt
ime/isolation/PluginUtilsTest.java
158   164            );
159   165            assertTrue(PluginUtils.shouldLoadInIsolati
160        -                "org.apache.kafka.common.config.Fu
      166  +                "org.apache.kafka.common.config.pr
```

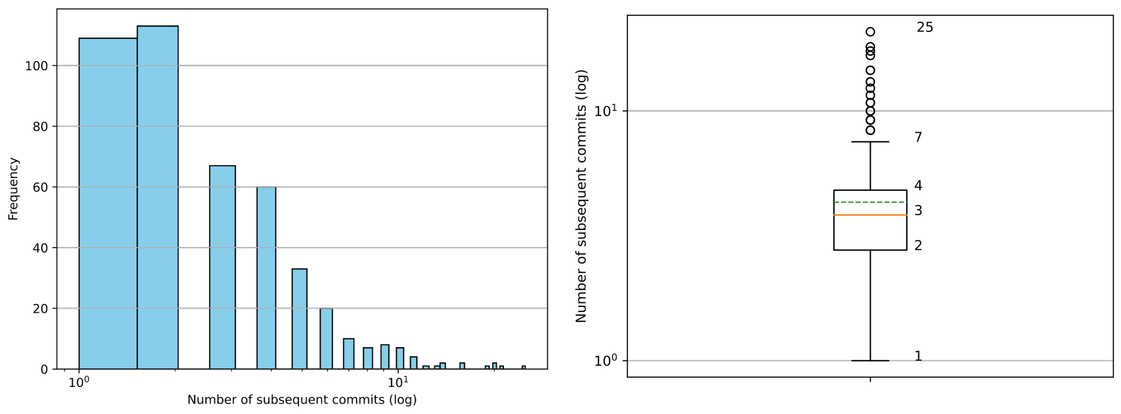
kkonstantine on 12 Jun 2018

nit: add also another case for something not ending in `ConfigProvider` ?

Figure 6.4: A *Move Class* edit that induced a review comment in Apache Kafka PR #5194

We visually explored the relationship between number of subsequent commits and number of refactorings (Figure 6.6) and got no clear general trend. Then, we ran Spearman's correlation test, considering 3-quantiles binning of data, and found a weak and positive monotonic correlation between the two variables (r_s = 0.24, n = 449, p < .05). Further, by examining the number of refactorings detected from the first until the third subsequent commit (Table 6.8), we realized that almost 78% of refactorings occur in a decreasing number of edits over the three first subsequent commits.

Given that, we conjecture that most refactoring edits occur in the initial PR subsequent commits and their number gradually decreases over them. When examining 65 refactoring-inducing PRs (validated refactorings), we identified six refactoring-inducing PRs that do not follow the pattern of decreasing the number of refactorings over subsequent commits: Dubbo #4099, Flink #7945, Flink #8222, Kafka #4735, Servicecomb-Java-Chassis #678, and Usergrid #102. In particular, they include refactoring edits led by the author. However, most refactoring-inducing PRs comprising only refactorings induced by code review (33/35) follow the pattern (Dubbo #3174



(a) Histogram

(b) Boxplot

Figure 6.5: Number of subsequent commits in refactoring-inducing PRs

Table 6.8: Number of detected refactorings from the first until the third subsequent commit in refactoring-inducing PRs

Subsequent commit	***Number of detected refactorings***	***Proportion***
1st	855	855/2,104 (40.6%)
2nd	428	428/2,104 (20.3%)
3rd	346	346/2,104 (16.5%)
Total	*1,629*	*1,629/2,104 (77.3%)*

and Sling-Org-Apache-Sling-Feature-Analyser #16 are exceptions) – 35/65 (53.8%) refactoring-inducing PRs consist of edits induced only by code review.

> *Finding 2.* PR authors tend to perform refactoring edits in the initial subsequent commits for addressing the suggestions from code review.

Previous research found that changes are driven by review comments [40], and refactorings are also triggered by discussion [112]. Finding 2 reinforces the capacity of code reviewing on refactoring edits because the subsequent commits can express the implemented changes in response to reviewers' comments. We clarify that we extended the knowledge on the practice of refactoring at the PR level because we

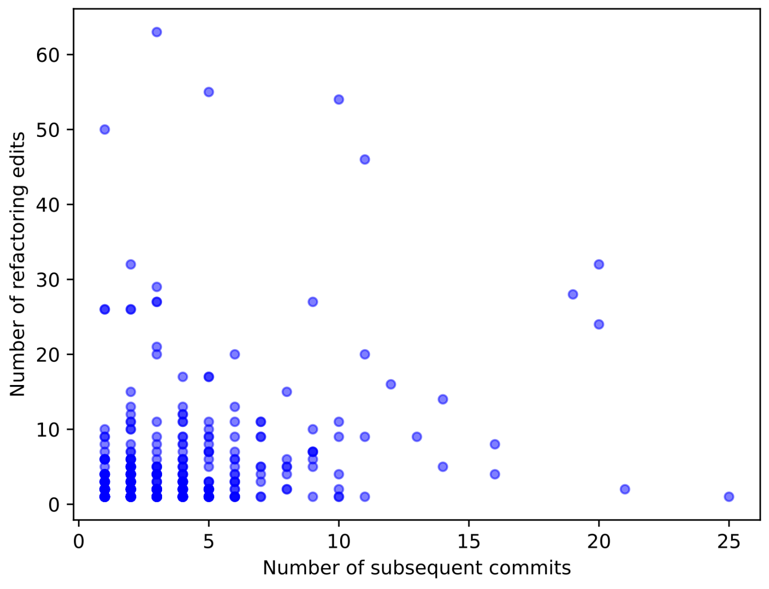


Figure 6.6: Scatterplot of number of refactorings in relation to number of subsequent commits in 449 refactoring-inducing PRs

explored refactorings as one of the changes occurring in code under review (Beller and colleagues do not deal with such a context) and we examined PRs in light of our refactoring-inducing PR definition (Pantiuchina and colleagues included initial PR commits in their study [112]).

Furthermore, we observed that the relevance of code review reflects on the time to merge a PR. In this context, refactoring-inducing PRs, in which code review induced refactorings, take 8.8 on average (SD = 13.8) and three on median (IQR = 10) days to merge; whereas, the other refactoring-inducing PRs take 24.8 on average (SD = 44.9) and 5.5 on median (IQR = 20.5) days to merge.

Low-level and high-level refactorings

We categorize 2,104 refactoring edits detected by RefactoringMiner in low-level and high-level refactorings, as our classification introduced in Chapter 3, Table 3.5. In the sample of 449 refactoring-inducing PRs, the most of refactorings are low-level edits (about 95%), as shown in Table 6.9. Moreover, Table 6.10 presents the types of high-level refactorings detected in our sample.

Table 6.9: Low-level and high-level refactorings detected by RefactoringMiner in 449 refactoring-inducing PRs

Level	***Number of detected refactorings***	***Proportion (%)***
High-level	119	5.7
Low-level	1,985	94.3
Total	*2,104*	*100.0*

Table 6.10: Types of high-level refactorings detected by RefactoringMiner 1.0 in 449 refactoring-inducing PRs

Type of refactoring	***Number of detections***	***Proportion (%)***
Move Class	41	34.5
Pull Up Method	23	19.3
Push Down Method	16	13.4
Push Down Attribute	10	8.4
Extract Interface	10	8.4
Pull Up Attribute	8	6.7
Extract Class	7	5.9
Extract Superclass	4	3.4
Total	*119*	*100.0*

Finding 3. Most of the refactorings detected in Apache's refactoring-inducing PRs consist of low-level edits.

We categorized the types of refactoring into low-level and high-level edits in the light of their impact on code design, based on technical descriptions provided by Fowler to explain that high-level refactorings are structured in terms of low-level refactorings [60]. Hence, our classification scheme may differ from previous research at that point. Our result reinforces the conclusions of Murphy-Hill and colleagues concerning the significant number of refactorings performed in low level [100], since those edits do not alter the interface of programs, constituting reduced impacts of change in comparison to high-level refactorings. It is worth remembering that our strategy for counting low-level and high-level refactorings considers each refactoring edit individually; that is, we compute a high-level refactoring independently of the low-level refactorings that it may include. For instance, we count one *Extract Superclass*, which comprises one *Move Attribute* and one *Move Method* edits, as one high-level and two low-level refactorings (e.g., Incubator-Iceberg #183).

In our purposive sample of 65 refactoring-inducing PRs (validated refactorings), low-level refactorings also happen more often – 297/366 (81.1%) edits (Table 6.11), which also corroborates results from a stratified sample in our comparative study between refactoring-inducing and non-refactoring-inducing PRs (Chapter 4, Table 4.6). Of those 297 low-level refactoring edits, 238 (80.1%) are not part of high-level edits.

We explored the age of PRs by computing the difference between the creation date of repositories and the creation date of PRs (in years). We realize that newer refactoring-inducing PRs present a higher number of high-level refactoring edits (Appendix D, Table D.10). Accordingly, it seems that high-level refactorings are more likely to happen in newer refactoring-inducing PRs, maybe denoting their repositories' code evolution. In addition, we identified high-level refactorings in ten refactoring-inducing PRs in which code review induced edits (*Extract Interface* and *Extract Superclass* were suggested only in this category), and eight, in which authors led the refactorings. This result reinforces the occurrence of more low-level refactorings than high-level ones.

Table 6.11: Number of low-level and high-level refactorings in 65 refactoring-inducing PRs (validated refactorings)

Refactoring-inducement	***Low-level edits***	***High-level edits***
Code review	138	30
Author	159	39
Total	*297/366 (81.1%)*	*69/366 (18.9%)*

In addition, we leveraged the classification of 65 refactoring-inducing PRs into the primary types of change (adaptive, corrective, and perfective) from the previous study (Chapter 5), and we contrasted them to kinds of refactoring edits (Table 6.12).[2] Then,

[2](*) Including high-level edit(s).

we realize that the most typical kinds of refactoring (*Change Type*, *Rename*, *Extract* and *Move* edits) are performed in refactoring-inducing PRs regardless of refactoring-inducement and type of change. The least common refactorings (*Merge*, *Parameterize*, *Replace*, and *Split*) are more frequent in refactoring-inducing PRs in which code review induced edits, regardless of the type of change.

Table 6.12: Kinds of refactorings by type of change in 65 refactoring-inducing PRs (validated refactorings)

Refactoring-inducement	**Type of change**		
	Adaptive	***Corrective***	***Perfective***
Author	Change Package Change Type Extract* Extract and Move Move* Rename	Change Type Extract* Extract and Move Inline Move* Pull Up* Push Down* Rename	Change Type Move* Rename
Code review	Change Package Change Type Extract* Extract and Move Merge Move* Move and Rename* Parameterize Pull Up* Push Down* Rename Replace	Change Type Extract* Pull Up* Rename Split	Change Package Change Type Extract Inline Move* Parameterize Push Down* Replace Rename Split

Although considering a distinct catalog of types, refactoring detection tool, and level under investigation (commit versus PR), this result embraces previous findings from Palomba et al. that investigated the relationship between types of refactoring and the type of change [110]. After all, we identified refactorings dealing with principles of object-oriented programming (*Extract*, *Move*, and inheritance-related refactorings) in adaptative changes, comprehensibility (*Rename*) and maintainability (*Change Type*) in corrective changes, and comprehensibility (*Rename*) in perfective changes.

In addition, we observed that code review induced more high-level refactorings (*Move and Rename*, *Pull Up*, and *Push Down* edits) in refactoring-inducing PRs that consist of adaptative and perfective changes. Whereas, authors addressed high-level refactorings (*Extract*, *Move*, *Pull Up*, and *Push Down*) in refactoring-inducing PRs that focus on corrective changes. Thus, we speculate that authors tend to conduct more complex refactoring edits to deal with corrective changes.

Although we do not investigate the intentions behind refactorings, our result corroborates findings from Paixão and colleagues on developers commonly employing refactoring to address adaptative and perfective changes [109].

6.3 Implications

As follows, we provide a few directions for researchers, practitioners, and tool builders.
Researchers: Findings 1 and 3 suggest that most refactoring-inducing PRs concentrate effort in low-level refactorings (including, low-level refactorings can be part of high-level refactorings [60]). Hence, perhaps a PR can be understood as an isolated unit at code review time, which makes more complex changes (e.g., refactor the code design) a challenge. We propose that future research regarding the practice of refactoring at the PR level to investigate code repositories to identify patterns related to missing opportunities of implementing more complex refactorings at code review time. Empirical evidence on a lack of discussing architectural changes during code review supports our claim [108]. For that, researchers may start exploring repositories over their initial phases (i.e., over the first months/years), which could facilitate some visualization of missing opportunities of code improvement. As a result, researchers can potentially achieve guidelines for practitioners and tool builders towards more effective code review practice. We strongly recommend replication studies in order to get advanced knowledge on the practice of code refactoring at the PR level.

Practitioners: Finding 2 indicates that when code review induces refactoring, the authors apply them in the initial subsequent commits, which likely reduces the time to merge a PR – maybe, this indicates that restructuring code is the priority of reviewers. Therefore, we claim that practitioners should pay more attention to resources for assisting the code review process, such as the guidelines proposed in Chapter 5, Section 5.3.

Tool builders: As a whole, our findings highlight the relevance of code review as a trigger for code refactoring, including a reduction in time to merge. Therefore, we suggest the development of tools for recommending reviewers based on the experience of them against authors, aiming to provide opportunities for more complex code improvements such as code design changes at reviewing time. Previous research found empirical evidence that the reviewer experience is the main factor influencing code review quality [80], but we consider that such a subject deserves more attention. Specifically, when dealing with cases in which, for instance, a PR author is already an experienced contributor (author/reviewer). Our claim is supported by empirical evidence on the need for user-centric approaches to design reviewer recommendation solutions [83]. *Review Bot*, for instance, provides a reviewer recommendation based on the change history of code lines [34], but we suggest another route: the contributions of reviewers against authors (accessible from GitHub profiles) may be considered, as we argued in Chapter 5, Section 5.3.

6.4 Limitations

This characterization study presents a few limitations, despite the efforts in the purpose of countermeasures to deal with **validity and reliability** issues (Table 6.13).

First, we made an effort to design the analysis procedures and systematically explain them in order to establish a chain of evidence when describing the sequence of results and conclusions. For that, we considered both quantitative and qualitative refactoring-related data. Nevertheless, 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.

Table 6.13: Validity and reliability countermeasures for the characterization study of refactoring edits in refactoring-inducing PRs

Type	***Description***
Internal validity	Proposal for a research design to guide the data analysis procedures
Construct validity	Establishment of a chain of evidence based on a mixed-methods data analysis (quantitative and qualitative)
	Quantitative study based on refactoring-inducing pull requests found in a representative sample of Apache's merged PRs
	Qualitative study based on a manually validated sample of refactoring-inducing PRs
Reliability	Effort towards clarifying the data analysis procedures in order to enable replications

The quantitative analysis investigates a representative sample of Apache's merged PRs, as explained in Chapter 3, Section 3.2. Three researchers validated the refactorings edits of a sample of refactoring-inducing PRs for the qualitative analysis.

We reinforce that our findings, in potential, are exclusively extended to cases that have common characteristics with Apache's projects. To enable replications, we provide a few instructions in our reproduction kit publicly available [22].

6.5 Concluding Remarks

In this chapter, we describe a characterization study about refactoring-inducing PRs, focusing on refactoring edits. In particular, this study complements our two previous studies towards a more comprehensive understanding of the code review process at the PR level.

Next, we discuss our findings and contributions in relation to previous research.

Chapter 7

Related Work

This thesis aims to characterize refactoring-inducing PRs. For that, we mined merged PRs from GitHub (Chapter 3) and carried out three inter-dependent characterization studies (Chapters 4–6). Accordingly, the related works enclose contributions from two fields: characterization of code review and characterization of refactorings throughout code evolution, described in the following sections.

7.1 Characterization of Code Review

The code review practice has been the focus of characterization studies that explore both OSS and industrial scenarios. Hence, it is perceptible the enhancement of techniques, tools, and recommendations for supporting the code review over time as a result of research initiatives over time. Table 7.1 lists the projects and data analyzed by related works.

Table 7.1: A summary of characterization studies on code review

Related work	***Year***	***Type***	***Analyzed project(s)***	***Analyzed data***
[122]	2008	Case study	Apache HTTP Server	Archival of email discussion Version history data

Continued on next page

Table 7.1 – continued from previous page

Related work	***Year***	***Type***	***Analyzed project(s)***	***Analyzed data***
[123]	2011	Empirical study	Apache HTTP Server, FreeBSD, Linux Kernel KDE, Subversion	Archival of email discussion
[120]	2013	Empirical study	Google Android and Chromium OS Microsoft Bing, Office and MS SQL	Code review data from Gerrit and CodeFlow
[33]	2013	Empirical study	Microsoft projects	Survey and interviews data Code review data from CodeFlow
[121]	2014	Empirical study	Apache HTTP Server, FreeBSD, Linux, KDE Subversion	Interviews data Archival of email discussion Version history data
[94]	2014	Case study	Qt, VTK, and ITK	Code review data from Gerrit
[40]	2014	Empirical study	ConQAT and GROMACS	Code review data from Gerrit
[68]	2014	Empirical study	GitHub projects	PRs data
[45]	2015	Empirical study	Microsoft projects	Interviews data Code review data from CodeFlow
[81]	2015	Empirical study	Mozilla projects	Version history data Issues tracking data from Bugzilla
[73]	2016	Case study	Xen project	Archival of email discussions
[95]	2016	Empirical study	Qt, VTK, and ITK	Code review data from Gerrit

Continued on next page

Table 7.1 – continued from previous page

Related work	***Year***	***Type***	***Analyzed project(s)***	***Analyzed data***
[80]	2016	Empirical study	Mozilla projects	Survey data
[116]	2017	Empirical study	Proprietary projects	PRs data from GitHub
[87]	2017	Empirical study	Angular.js, ElasticSearch, and Rails	PRs data from GitHub
[44]	2017	Empirical study	OSS and Microsoft projects	Surveys data
[126]	2018	Case study	Google projects	Interviews and Survey data Code review data from CRITIQUE
[82]	2018	Empirical study	Active Merchant	Survey data PRs data from GitHub

Initially, the characterization studies of code review 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 [122]. They characterized Apache's peer reviews as early, frequent reviews of small, independent, complete contributions conducted asynchronously by a small number of expert reviewers. *These findings express code reviewing as a traditional practice in Apache projects, which inspired us to explore refactoring-inducing PRs in its repositories.*

Rigby and Storey studied reviewers interactivity in five OSS projects. They identied a few benefits and weaknesses of *broadcast peer review* – a modality in which hundreds of potential reviewers could examine a code. At that point, code review has expanded the reach of collaborative development [123]. The differential of OSS code review practices for code quality has become evident, as claimed by Rigby et al., when suggesting a few recommendations suitable to the industry as from the OSS code reviewing [119]. In this context, Rigby and Bird studied convergent code review practices in both industrial and OSS scenarios aiming at discovering efficient methods of review [120]. Also, Rigby et al. extended the knowledge on peer review practice, examining a lot of aspects, including the experience (i.e., length of time a developer has been with a project) of authors and reviewers [121]. They found the reviewers typically have more experience than authors. *We expand the knowledge about the experience of Apache authors and reviewers by providing arguments denoting a difference between the experience of authors and reviewers in refactoring-inducing and non-refactoring-inducing PRs* (Chapter 5, Section 5.2.1).

As a whole, the previous findings signal a review process that fits MCR – a lightweight review, focusing on solving problems instead of finding defects. Bacchelli and Bird characterized the motivations behind code review and the associated challenges for researchers and practitioners while defining the current MCR process – an informal, tool-based, and asynchronous code review [33]. Such a work sheds light on a code review addressing code improvements while favoring knowledge transfer. *This motivated us investigating code review as a process driven by initiatives towards code improvement. By considering refactoring as a potential code improvement (as empirical evidence of its benefits* [57; 71; 138]*) in code review time, we drew the foundations of this thesis.*

Later, we notice a series of research initiatives to better understand MCR. Sadowski et al. studied code review practice at Google, discovering educational aspects as one of the motivations behind such a process [126]. As well, *our findings* (Chapter 5, Section 5.3) *reinforce the intrinsic nature of code review in terms of learning opportunities.* McIntosh et al. explored the relationship between MCR and software quality, identifying that a low proportion of changes reviewed and involvement degree of reviewers generate additional post-release defects [94; 95]. Beller et al. investigated the benefits of MCR by examining the issues fixed at code review time in OSS projects. They found that most of the changes are due to review comments, concerning code improvement instead of fixing defects [40]. *As a whole, those findings reinforce code improvement as a relevant objective of MCR, thus motivating us to advance knowledge on code reviewing-related aspects such as review comments.*

Also, studies explored code review quality. In this context, Kononenko et al found empirical evidence that aspects such as reviewer experience (i.e., the overall number of completed reviews) and the thoroughness of feedback are associated with the code review quality [81; 80]. Bosu et al. identified factors influencing the review comments usefulness [45] – for instance, review comments asking questions to understand code were considered non-useful. But, such an aspect may induce refactoring edits at the PR level in line with our findings. *In this context, the guidelines for structuring review comments denote a differential of our studies* (Chapter 5, Section 5.3).

The analysis of technical aspects of code reviewing has been the focus of several empirical studies, *thus constituting an enriched set of technical aspects from which we selected the ones under investigation in our studies.* In this perspective, we can consider:

- code churn [68; 80; 95],
- number of file changes [68; 80; 82],
- number of commits [68; 82],
- number of review comments [68; 80],
- length of discussion [80; 82; 95],
- number of reviewers [75; 95],
- time to merge [73; 68],
- reviewer experience [82; 116], and
- review comments [40; 120].

Pull-based development is the focus of a few empirical studies. In this context, Gousios et al. investigated GitHub PRs data for understanding the factors influencing their effectiveness (e.g., code churn, number of commits, number of file changes, and number of review comments) [68]. In particular, besides providing us with a few features for exploring in our studies, this work found empirical evidence that code review affects the time to merge a PR. *We advanced the knowledge regarding pull-based development when we found that refactoring-inducing PRs with refactorings induced by* *code review take less time to merge than refactoring-inducing PRs with refactorings led by PR authors* (Chapter 6, Subsection 6.2.2).

Review comments in pull-based development has been examined in a few studies. Rahman et al. developed a machine learning prediction model for assisting developers when formulating code review comments after comparing useful and non-useful review comments based on the content of review comments and reviewer experience [116]. This work emphasizes the reviewer experience as a relevant factor influencing the useful review comments, whereas *our studies shed light on considering the experience of reviewers against authors when dealing with refactorings at the PR level.* Li et al. found 15 typical code review patterns in GitHub PRs such as reviewing for code correctness (e.g., indicating blank lines) [87]. *We go further in the sense of encompassing code review properties while investigating the refactoring-inducing and non-refactoring-inducing PRs.* Bosu et al. revisited the OSS and industrial scenario to investigate code review practice, including interactivity aspects and human effort required in review-related tasks [44]. They identified a few subjects for further research, such as assisting reviewers in articulating review comments. In particular, *our guidelines are an initiative to meet such a claim.*

7.2 Characterization of Refactoring Edits throughout Code Evolution

Table 7.2 summarizes projects and data analyzed in characterization studies on refactorings over code evolution.

Table 7.2: A summary of characterization studies on refactorings over code evolution

Related work	***Year***	***Type***	***Analyzed project(s)***	***Analyzed data***
[100]	2012	Empirical study	Eclipse and Mylyn	Usage data from developers Version history data

Continued on next page

Table 7.2 – continued from previous page

Related work	***Year***	***Type***	***Analyzed project(s)***	***Analyzed data***
[78]	2012	Empirical study	Microsoft Windows 7	Interviews Version history data
[132]	2013	Experiment	Apache Commons Collections and JHotDraw	Software versions
[79]	2014	Experiment	Microsoft Windows 7	Version history data
[129]	2016	Empirical study	GitHub projects	Version history data Survey
[110]	2016	Empirical study	Apache Ant and Xerces Argo UML	Version history data
[147]	2019	Empirical study	Android, Apache, and Eclipse	Version history data
[47]	2020	Empirical study	GitHub projects	Version history data
[112]	2020	Empirical study	GitHub projects	Version history data
[109]	2020	Empirical study	Couchbase and Eclipse	Code review data from Gerrit
[111]	2020	Empirical study	Eclipse projects	Survey data Code review data from Gerrit
[26]	2021	Case study	Xerox projects	Survey data Code review data from Xerox review framework

Empirical characterization studies of refactorings have explored code change histories for distinct purposes. Murphy-Hill et al., when exploring refactoring practices in the OSS scenario, found empirical evidence on frequent use of floss refactoring, rare self-affirmed refactorings in commits, and about 50% of high-level refactoring edits [100]. Kim et al. explored refactoring in the industrial setting, and they identified benefits (e.g., a reduction of post-release defects [78]) and challenges (e.g., measuring the impact of refactoring requires multi-dimensional assessment [79]). *As a whole, those findings shed light on the relevance of examining code repositories to better understand the refactoring practice.*

In another research line, Soares et al. compared three distinct approaches (manual analysis, commit message, and dynamic analysis) to check behavioral preservation, besides evaluating two techniques to identify refactoring edits between two software versions [132]. *Their findings identify the challenges related to assurance of behavior preservation and the limitations of automatic refactoring detection.* To deal with these constraints in our studies, we propose a few countermeasures (*Limitations*, Chapters 3–6).

Motivations behind refactoring edits have been empirically explored in both OSS and industrial settings. In this perspective, Kim et al. identified readability [79], Vassallo et al. uncovered understandability and maintainability [147], AlOmar et al. found readability and understandibility [26], whereas Silva et al. and Palomba et al. discovered changes in the requirements (e.g., bug fixes requests) [129; 110] as main reasons to refactor code. These studies are relevant because they drive future research and advance in developing support tools. In this sense, Silva et al. suggest that refactoring recommendation systems should refocus from code-smell-oriented to maintenance-task-oriented solutions [129]. Brito et al. mined refactorings is PR commits and figured that most of the edits are of different types and performed in up to three commits [47]. *These studies focus on the commit level and represent a reference for our studies regarding refactorings at the PR level.* Also, they reinforce the relevance of the analysis of changes applied to the code as a means to get details on the practices of refactoring.

Distinct from the previous works (that focus in the commit level), Pantiuchina et al. observed that code readability, change- and fault-proneness, and experience of developers are factors influencing refactorings, when exploring refactoring at the PR level [112]. It is worth clarifying that Pantiuchina et al. analyzed discussion and commits of merged PRs, containing at least one refactoring in any one of their commits, without pre-processing squashed and merged PRs (to the best of our knowledge). They found that most of the refactorings are triggered from either the original intents of PRs or discussion with other developers. Findings from Pantiuchina et al. are motivating, since they indicate the influence that code review has on refactoring edits at the PR level. Our studies differ from such previous work because we specifically provide a characterization of refactoring edits in refactoring-inducing PRs (Chapter 6). Particularly, *we propose an exploration of refactorings performed as part of changes at subsequent commits from merged PRs, as a complementary study to the investigation regarding code reviewing-related aspects and refactoring-inducement* (Chapters 4 and 5).

Recently, 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 [109]. Their findings suggest aspects of refactoring and code review to be explored at the GitHub PR level (e.g., typical types of refactoring edits). 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. In addition, Panichella and Zaugg proposed a taxonomy for types of review changes, including refactoring as a structural change [111]. *This thesis, aiming to characterize refactoring-inducing PRs, provides a study of refactorings-related aspects according to the refactoring-inducement context, independently of developers' intents.*

7.3 Concluding Remarks

Figure 7.1 shows an overview of related works closest to this thesis. In particular, we emphasize that our main objective is to fill up the knowledge gap on PRs aligned to our refactoring-inducing definition instead of investigating motivations behind refactoring edits in time of code review. For this purpose, we firstly investigated the differences between refactoring-inducing and non-refactoring-inducing PRs (Chapter 4). Therefore, we speculate that our findings complement the related studies in the sense of a better understanding of MCR practice concerning code evolution.

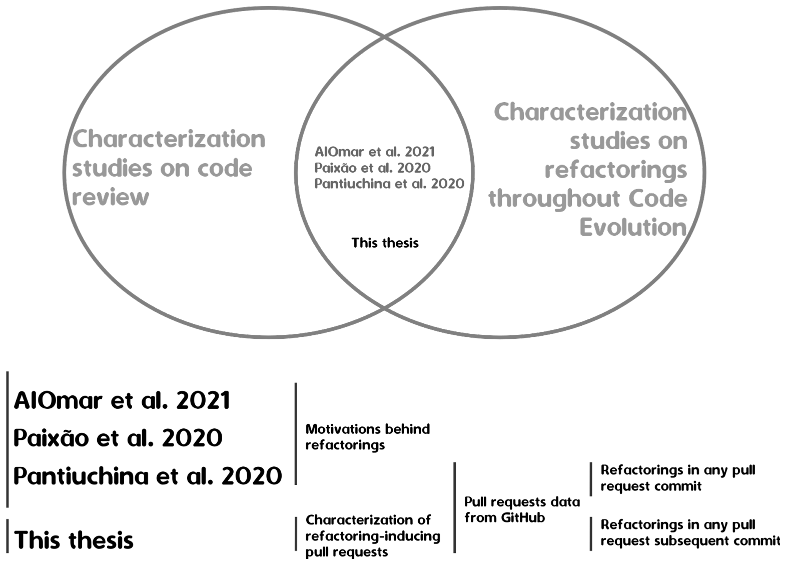

Characterization studies on code review
AlOmar et al. 2021
Paixão et al. 2020
Pantiuchina et al. 2020
This thesis
Characterization studies on refactorings throughout Code Evolution
AlOmar et al. 2021
Paixão et al. 2020
Pantiuchina et al. 2020
Motivations behind refactorings
This thesis
Characterization of refactoring-inducing pull requests
Pull requests data from GitHub
Refactorings in any pull request commit
Refactorings in any pull request subsequent commit

Figure 7.1: An overview of this thesis concerning other works

Our empirical studies consider 1,639 Apache's merged PRs, comprising 4,000 subsequent commits, 2,104 refactoring edits identified within 449 PRs, considering up to 40 distinct types of refactoring detectable by a state-of-the-art tool.

In the next chapter, we provide a few conclusions and actionable directions for future research.

Chapter 8

Conclusions

This thesis tries to cope with an existing knowledge gap in the characterization of refactoring-inducing PRs. For that, we mined refactorings and code review data from GitHub and designed three complementary empirical studies. First, we concentrated on discovering similarities/dissimilarities between refactoring-inducing and non-refactoring-inducing PRs in a mixed-methods study. We found significant differences concerning code churn, number of file changes, number of subsequent commits, number of review comments, length of discussion, and time to merge in the quantitative study supported by ARL. This unsupervised machine learning technique was crucial to formulate hypotheses on similarities/dissimilarities. By following such an approach based on similarities/dissimilarities, we could identify the initial group of change- and code reviewing-related properties towards a characterization of refactoring-inducing PRs. We identified a relevant number of refactoring-inducing PRs (133/228, 58.3%) in which code review induced one refactoring edit, at least, in the qualitative study.

Based on the found results, we conceived a study to qualitatively investigate review comments in both refactoring-inducing and non-refactoring-inducing PRs in order to understand how review comments might induce refactoring edits at code review. We observed differences between refactoring-inducing and non-refactoring-inducing PRs when analyzing review comments. In particular, we found that beyond the content and structure of review comments following specific patterns (including direct suggestions of refactorings), the experience of reviewers and authors is a factor motivating for refactoring-inducing PRs. Then, we carried out a mixed-methods analysis of refactoring edits in refactoring-inducing PRs to characterize refactoring practices. From that, we found typical code improvements addressed at the PR level, confirming findings from previous works, while we claim that practitioners might miss opportunities to promote more complex changes (e.g., high-level refactorings).

In contrast to non-refactoring-inducing PRs, refactoring-inducing PRs incorporate a higher workload in size (code churn, file changes, and subsequent commits) and code review (review comments). Their review comments address more complex issues, following a few patterns, which we compiled through guidelines for practitioners towards a more effective code review. The refactorings comprise mostly low-level edits, indicating that practitioners need support from researchers and tool builders in order to leverage opportunities to manage more complex edits (e.g., architectural changes) at the PR level.

It is worth emphasizing how the findings from our three empirical studies conceived a more comprehensive characterization of refactoring-inducing PRs, thus providing implications for researchers, practitioners, and tool builders. In our first study, we recommended that future experiment designs on MCR with PRs make a distinction between refactoring-inducing and non-refactoring PRs (Chapter 4, Findings 2-6 and 8). By exploring the review comments in our second study, we advanced the knowledge of the differences between refactoring-inducing and non-refactoring-inducing PRs and also among subgroups of refactoring-inducing PRs: with refactorings led by the authors, with refactorings induced by code review, and with refactorings both led by the author and induced by code review (Chapter 5, Findings 3–6). Those findings reinforce our initial recommendations for researchers because they emphasize differences among PRs that might affect answers to research questions.

In our first study, we suggested project managers to invite more reviewers when a PR becomes refactoring-inducing in order to share the expected workload in reviewing and the knowledge of changes caused by refactoring edits to more team members (Chapter 4, Finding 7). In our second study, we realized the importance of considering reviewers more experienced than authors to reduce the missing opportunities for code improvements due to reviewer limitations (Chapter 5, Findings 2 and 6). Moreover, findings from our third study reinforce such a claim because we identified a high proportion of low-level refactorings in contrast to high-level ones in refactoring-inducing PRs (Chapter 6, Findings 1 and 3). In light of such a perception, we propose that future research specifies requirements for reviewer recommendations intending a more effective code reviewing.

Based on findings from our second study (Chapter 5, Findings 6–8), we proposed a few guidelines for composing valuable review comments towards code refactoring. In our third study, one of our findings reinforces our initial recommendations since it emphasizes the importance of code review inducing refactoring edits to reduce time to merge a PR (Chapter 6, Finding 2).

In summary, to the best of our knowledge, this is the first research work exploring aspects concerning refactorings and code review in light of refactoring-inducing PRs. As we conjectured, there are differences between refactoring-inducing and non-refactoring-inducing PRs. Although we can not claim the completeness of our methodology to deal with such a subject, we believe that our findings indicate actionable implications for researchers, practitioners, and tool builders towards a better understanding of code review practice as well as the implementation of novel resources to assist pull-based development. Our main contributions are:

- We investigated PRs merged by *merge pull request* and *squash and merge* options. We avoid either PRs merged by *rebase and merge* or merged PRs that suffered rebasing, intending to reduce threats to validity. To deal with squashed commits, we implemented a script that recovers them (*git squash* converts all commits in a PR into a single commit).
- We performed a quantitative and qualitative analysis of both code review and refactorings aspects in three empirical studies that, together, provide the first results towards a theory on refactoring-inducing PRs.
- We provided guidelines for articulating more effective review comments intending code improvements by refactoring. In this context, we observed that refactoring-inducing PRs, in which code review induced refactorings, present more effective code reviews than those which authors led the edits in terms of time to merge.

- We made available a complete reproduction kit including the mined datasets, implemented scripts, and instructions to enable replications and future research [22].

As future research, we propose a few directions:

- Investigating code characteristics and types of refactoring in refactoring-inducing PRs in order to get an in-depth understanding of how refactorings directly affect code changes. For instance, checking if file changes are a consequence of refactoring edits and specific types of refactoring;
- Exploring the association rules from our comparative study between refactoring-inducing and non-refactoring-inducing PRs to determine meaningful cause/effect relationships among features, also examining the types of refactoring, thus enriching our conclusions.
- Designing surveys with developers to enhance our conclusions, in specific, to confirm/refute our guidelines for more effective review comments. Also, assessing the guidelines for valuable review comments towards code refactoring by running a controlled experiment in which we instruct a group of reviewers and observe both the quality of review comments and code changes during reviewing;
- Considering our research design to investigate projects implemented in other programming languages rather than Java, for instance, using RefDiff to mine refactorings [128];
- Confirming/refuting our findings, by replication, in other projects in both OSS and industrial scenarios, so providing foundations for a theory on refactoring-inducing PRs; and
- Since refactoring can introduce subtle bugs, investigating the bug-proneness in refactoring-inducing and non-refactoring-inducing PRs. In this context, the complexity of bugs concerning refactorings might be explored, aiming to advance the knowledge regarding code review practices in pull-based development.

References

[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] GitLab developer guide GraphQL API. https://docs.gitlab.com/ee/api/graphql/. Accessed on: November 2021.

[9] GitLab platform. https://gitlab.com. Accessed on: November 2021.

[10] Phabricator code review system. https://www.phacility.com/phabricator/. Accessed on: June 2020.

[11] RefactoringMiner – a refactoring detection tool. https://github.com/tsantalis/RefactoringMiner. Accessed on: September 2019.

[12] Review Board code review system. https://www.reviewboard.org/. Accessed on: June 2020.

[13] Synopsys repositories comparison – report. https://www.openhub.net/repositories/compare. Accessed on: June 2020.

[14] Manifesto for Agile software development. https://agilemanifesto.org/, February 2001. Accessed on: August 2020.

[15] An experimental investigation on the innate relationship between quality and refactoring. *Journal of Systems and Software*, 107(C):1–14, September 2015.

[16] 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.

[17] Briefing: The Apache way. https://www.apache.org/theapacheway/, 2019. Accessed on: June 2020.

[18] Preliminary investigations on refactorings and modern code review. https://git.io/JTLum, July 2019.

[19] The state of the Octoverse – GitHub 2019 report. https://octoverse.github.com/, September 2019. Accessed on: July 2020.

[20] The Apache® Software Foundation projects statistics. https://projects.apache.org/statistics.html, November 2020. Accessed on: December 2020.

[21] Stack Overflow annual developer survey. https://insights.stackoverflow.com/survey/2020#most-popular-technologies, 2020. Accessed on: December 2020.

[22] Characterizing refactoring-inducing pull requests – Reproduction kit. https://github.com/flaviacoelho/thesis-reproduction-kit, November 2021.

[23] Community-led development "The Apache Way". https://apache.org/foundation/how-it-works.html#roles, 2021. Accessed on: June 2021.

[24] Type migration. `https://www.jetbrains.com/help/idea/type-migration.html`, March 2021. Accessed on: August 2020.

[25] 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.

[26] Eman A. AlOmar, Hussein AlRubaye, Mohamed W. Mkaouer, Ali Ouni, and Marouane Kessentini. Refactoring practices in the context of modern code review: An industrial case study at Xerox. In *Proceedings of the IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice*, ICSE-SEIP '21, pages 348–357, Virtual, May 2021. ACM.

[27] 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 Computer Society Press.

[28] Everton L. G. Alves, Myoungkyu Song, and Miryung Kim. RefDistiller: A refactoring-aware code review tool for inspecting manual refactoring edits. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '14, pages 751–754, Hong Kong, China, November 2014. ACM.

[29] 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, April 2018.

[30] Mauricio Aniche, Erick Maziero, Rafael Durelli, and Vinicius Durelli. The effectiveness of supervised machine learning algorithms in predicting software refactoring. *IEEE Transactions on Software Engineering*, pages 1–1, September 2020. Early access article.

[31] Mihael Ankerst, Markus M. Breunig, Hans-Peter Kriegel, and Jörg Sander. OPTICS: 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, May 1999. ACM.

[32] Howard Anton and Chris Rorres. *Elementary Linear Algebra: Applications Version*. Wiley publishing, eleventh edition, 2014.

[33] Alberto Bacchelli and Christian Bird. Expectations, outcomes, and challenges of modern code review. In *Proceedings of the 35th International Conference on Software Engineering*, ICSE '13, pages 712–721, San Francisco, USA, May 2013. IEEE Computer Society Press.

[34] Vipin Balachandran. Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation. In *Proceedings of the 35th International Conference on Software Engineering*, ICSE '13, pages 931–940, San Francisco, USA, May 2013. IEEE Computer Society Press.

[35] Mike Barnett, Christian Bird, João Brunet, and Shuvendu K. Lahiri. Helping developers help themselves: Automatic decomposition of code review changesets. In *Proceedings of 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 134–144, Florence, Italy, May 2015. IEEE Computer Society Press.

[36] Gabriele Bavota, Bernardino De Carluccio, Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, and Orazio Strollo. When does a refactoring induce bugs? an empirical study. In *Proceedings of the 12th International Working Conference on Source Code Analysis and Manipulation*, SCAM '12, pages 104–113, Trento, Italy, September 2012. IEEE Computer Society Press.

[37] Gabriele Bavota and Barbara Russo. Four eyes are better than two: On the impact of code reviews on software quality. In *Proceedings of the 31st IEEE International Conference on Software Maintenance and Evolution*, ICSME '15, pages 81–90, Bremen, Germany, September 2015. IEEE Computer Society Press.

[38] 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.

[39] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Longman Publishing, USA, 1999.

[40] 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 202–211, Hyderabad, India, May 2014. ACM.

[41] 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.

[42] Ana Carla Bibiano, Eduardo Fernandes, Daniel Oliveira, Alessandro Garcia, Marcos Kalinowski, Baldoino Fonseca, Roberto Oliveira, Anderson Oliveira, and Diego Cedrim. A quantitative study on characteristics and effect of batch refactoring on code smells. In *Proceedings of the 13th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '19, pages 1–11, Porto de Galinhas, Brazil, September 2019. IEEE Computer Society Press.

[43] Giuseppe Bonaccorso. *Machine Learning Algorithms*. Packt Publishing, first edition, 2017.

[44] 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 Transaction on Software Engineering*, 43(1):56–75, January 2017.

[45] 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, May 2015. IEEE Computer Society Press.

[46] 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, Tucson, USA, May 1997. ACM.

[47] Aline Brito, Andre Hora, and Marco T. Valente. Refactoring graphs: Assessing refactoring over time. In *Proceedings of the 27th International Conference on Software Analysis, Evolution and Reengineering*, SANER '20, pages 367–377, Ontario, Canada, February 2020. IEEE Computer Society Press.

[48] Neil Burdess. *Starting statistics: a short, clear guide*. Sage Publications, 2010.

[49] M. Emre Celebi and Kemal Aydin. *Unsupervised Learning Algorithms*. Springer Publishing Company, first edition, 2016.

[50] Scott Chacon and Ben Straub. *Pro Git*. Apress, second edition, 2014.

[51] Oscar Chaparro, Gabriele Bavota, Andrian Marcus, and Massimiliano Di Penta. On the impact of refactoring operations on code quality metrics. In *Proceedings of the 30th IEEE International Conference on Software Maintenance and Evolution*, ICSME '14, pages 456–460, Victoria, Canada, October 2014. IEEE Computer Society Press.

[52] Flávia 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, May 2019. IEEE Computer Society Press.

[53] Flávia Coelho, Nikolaos Tsantalis, Tiago Massoni, and Everton L. G. Alves. An empirical study on refactoring-inducing pull requests. In *Proceedings of the 15th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '21, pages 1–12, Bari, Italy, October 2021. ACM.

[54] Frans Coenen, Graham Goulbourne, and Paul Leng. Tree structures for mining association rules. *Data Mining and Knowledge Discovery*, 8(1):25–51, January 2004.

[55] J. W. Creswell. *Qualitative Inquiry and Research Design: Choosing among Five Traditions*. Sage Publications, third edition, 2012.

[56] Brett Daniel, Danny Dig, Kely Garcia, and Darko Marinov. Automated testing of refactoring engines. In *Proceedings of 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, September 2007. ACM.

[57] Massimiliano Di Penta, Gabriele Bavota, and Fiorella Zampetti. On the relationship between refactoring actions and bugs: A differentiated replication. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE '20, pages 556–567, Virtual, November 2020. ACM.

[58] Bradley Efron and Robert J. Tibshirani. *An introduction to the bootstrap*. Monographs on Statistics and Applied Probability. Chapman and Hall, 1993.

[59] Michael E. Fagan. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 15(3):182–211, 1976.

[60] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing, 1999.

[61] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, second edition, 2018.

[62] Johannes Fürnkranz and Tomáš Kliegr. A brief overview of rule learning. In *Proceedings of the 9th International Symposium on Rules and Rule Markup Languages for the Semantic Web*, RuleML '15, pages 54–69, Berlin, Germany, August 2015. Springer International Publishing.

[63] Tate Galbraith. Which code merging method should I use in GitHub? `https://t.ly/BDL9`, May 2020. Accessed on June 2020.

[64] Xi Ge, Saurabh Sarkar, and Emerson Murphy-Hill. Towards refactoring-aware code review. In *7th International Workshop on Cooperative and Human Aspects of Software Engineering*, CHASE '14, pages 99–102, Hyderabad, India, June 2014. ACM.

[65] Xi Ge, Saurabh Sarkar, Jim Witschey, and Emerson Murphy-Hill. Refactoring-aware code review. In *Proceedings of the 2017 IEEE Symposium on Visual Languages and Human-Centric Computing*, VL/HCC '17, pages 71–79, Raleigh, USA, 2017. IEEE Computer Society Press.

[66] Liqiang Geng and Howard J. Hamilton. Interestingness measures for data mining: A survey. *ACM Computing Surveys*, 38(3):9–es, September 2006.

[67] Bart Goethals. Frequent set mining. In Oded Maimon and Lior Rokach, editors, *Data Mining and Knowledge Discovery Handbook*, pages 321–338. Springer US, Boston, USA, 2010.

[68] 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, May 2014. ACM.

[69] 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, May 2016. ACM.

[70] Jiawei Han, Jian Pei, and Yiwen Yin. Mining frequent patterns without candidate generation. *SIGMOD Record*, 29(2):1–12, May 2000.

[71] 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, March 2018.

[72] Jez Humble and David Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation.* Addison-Wesley Professional, first edition, 2010.

[73] 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, May 2016. ACM.

[74] Tao Ji, Liqian Chen, Xin Yi, and Xiaoguang Mao. Understanding merge conflicts and resolutions in Git rebases. In *Proceedings of the 31st International Symposium on Software Reliability Engineering*, ISSRE '20, pages 70–80, Coimbra, Portugal, October 2020. IEEE Computer Society Press.

[75] 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, San Francisco, USA, May 2013. IEEE Computer Society Press.

[76] Philip M. Johnson. Reengineering inspection. *Communications of ACM*, 41(2):49–52, February 1998.

[77] 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, Montreal, Canada, October 2002. IEEE Computer Society Press.

[78] Miryung Kim, Thomas Zimmermann, and Nachiappan Nagappan. A field study of refactoring challenges and benefits. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, Cary, EUA, November 2012. ACM.

[79] 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.

[80] 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, May 2016. ACM.

[81] Oleksii Kononenko, Olga Baysal, Latifa Guerrouj, Yaxin Cao, and Michael W. Godfrey. Investigating code review quality: Do people and participation matter? In *Proceedings of the 2015 IEEE International Conference on Software Maintenance and Evolution*, ICSME '15, pages 111–120, Bremen, Germany, September 2015. IEEE Computer Society Press.

[82] 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, May 2018. ACM.

[83] Vladimir Kovalenko, Nava Tintarev, Evgeny Pasynkov, Christian Bird, and Alberto Bacchelli. Does reviewer recommendation help developers? *IEEE Transactions on Software Engineering*, 46(7):710–731, July 2020.

[84] Robert Layton. *Learning Data Mining with Python*. Packt Publishing, first edition, 2015.

[85] Carlene Lebeuf, Margaret-Anne Storey, and Alexey Zagalsky. Software bots. *IEEE Software*, 35(1):18–23, January/February 2018.

[86] 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, December 2003.

[87] 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.

[88] 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, January/February 2012.

[89] 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, February 2019. IEEE Computer Society Press.

[90] Hassan Mansour and Nikolaos Tsantalis. Refactoring-aware commit review chrome extension. `https://t.ly/J3Wr`, 2020. Accessed on: November, 2020.

[91] Martin N. Marshall. Sampling for Qualitative Research. *Family Practice*, 13(6):522–526, December 1996.

[92] Joseph A. Maxwell. Designing a qualitative study. In Leonard Bickman and Debra J. Rog, editors, *The SAGE Handbook of Applied Social Research Methods*, pages 214–253. Sage Publications, 1997.

[93] Kenneth O. McGraw and Seok P. Wong. A common language effect size statistic. *Psychological Bulletin*, 111(2):361–365, March 1992.

[94] 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, May 2014. ACM.

[95] Shane McIntosh, Yasutaka Kamei, Bram Adams, and Ahmed E. Hassan. An empirical study of the impact of modern code review practices on software quality. *Empirical Software Engineering*, 21(5):2146–2189, October 2016.

[96] M. B. Miles and A. M. Huberman. *Qualitative Data Analysis: An Expanded Sourcebook*. Sage Publications, second edition, 1994.

[97] Ehsan Mirsaeedi and Peter C. Rigby. Mitigating turnover with code review recommendation: Balancing expertise, workload, and knowledge distribution. In *ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, pages 1183–1195, Seoul, South Korea, June 2020. ACM.

[98] Audris Mockus and Lawrence G. Votta. Identifying reasons for software changes using historic databases. In *Proceedings of the 2000 International Conference on Software Maintenance*, ICSM '00, pages 120–130, Washington, USA, October 2000. IEEE Computer Society Press.

[99] Rodrigo Morales, Shane McIntosh, and Foutse Khomh. Do code review practices impact design quality? A case study of the Qt, VTK, and ITK projects. In *Proceedings of the 22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering*, SANER '15, pages 171–180, Montreal, Canada, March 2015. IEEE Computer Society Press.

[100] 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.

[101] 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, July 2013. Springer-Verlag.

[102] 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.

[103] Jonhnanthan Oliveira, Rohit Gheyi, Felipe Pontes, Melina Mongiovi, Márcio Ribeiro, and Alessandro Garcia. Revisiting refactoring mechanics from tool devel-

opers' perspective. In Gustavo Carvalho and Volker Stolz, editors, *Formal Methods: Foundations and Applications*, pages 25–42, Ouro Preto, Brazil, November 2020. Springer International Publishing.

[104] William F. Opdyke. *Refactoring: A Program Restructuring Aid in Designing Object-Oriented Application Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[105] 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.

[106] 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):1–53, June 2016.

[107] Matheus Paixão, Jens Krinke, DongGyun Han, Chaiyong Ragkhitwetsagul, and Mark Harman. The impact of code review on architectural changes. *IEEE Transactions on Software Engineering*, 47(5):1041–1059, May 2021.

[108] Matheus Paixão and Paulo H. Maia. Rebasing in code review considered harmful: A large-scale empirical investigation. In *Proceedings of the 19th International Working Conference on Source Code Analysis and Manipulation*, SCAM '19, pages 45–55, Cleveland, USA, September 2019.

[109] Matheus Paixão, Anderson Uchôa, Ana Carla Bibiano, Daniel Oliveira, Alessandro Garcia, Jens Krinke, and Emilio Arvonio. Behind the intents: An in-depth empirical study on software refactoring in modern code review. In *Proceedings of the 17th International Conference on Mining Software Repositories*, MSR '20, pages 125–136, Virtual, June 2020. ACM.

[110] Fabio Palomba, Andy Zaidman, Rocco Oliveto, and Andrea De Lucia. An exploratory study on the relationship between changes and refactoring. In *Pro-*

ceedings of the 25th International Conference on Program Comprehension, ICPC '17, pages 176–185, Buenos Aires, Argentina, May 2017. IEEE Computer Society Press.

[111] Sebastiano Panichella and Nik Zaugg. An empirical investigation of relevant changes and automation needs in modern code review. *Empirical Software Engineering*, 25:4833–4872, January 2020.

[112] 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):1–30, September 2020.

[113] 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):1–27, November 2018.

[114] Michael Q. Patton. *Qualitative Research Evaluation Methods: Integrating Theory and Practice*. Sage Publications, fourth edition, 2014.

[115] 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, September 2010. IEEE Computer Society Press.

[116] Mohammad M. 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, May 2017. IEEE Computer Society Press.

[117] Achyudh Ram, Anand A. 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 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, November 2018. ACM.

[118] 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, April 2018.

[119] Peter Rigby, Brendan Cleary, Frederic Painchaud, Margaret-Anne Storey, and Daniel M. German. Contemporary peer review in action: Lessons from open source development. *IEEE Software*, 29(6):56–61, November 2012.

[120] Peter C. Rigby and Christian Bird. Convergent contemporary software peer review practices. In *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE '13, pages 202–212, Saint Petersburg, Russia, August 2013. ACM.

[121] 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):1–33, September 2014.

[122] 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, May 2008. ACM.

[123] Peter C. Rigby and Margaret-Anne Storey. Understanding broadcast based peer review on open source software projects. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 541–550, Honolulu, USA, May 2011. ACM.

[124] 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.

[125] Per Runeson, Martin Höst, Austen Rainer, and Bjorn Regnell. *Case Study Research in Software Engineering: Guidelines and Examples.* Wiley Publishing, first edition, 2012.

[126] 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, May 2018. ACM.

[127] Chris Sauer, David R. 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.

[128] Danilo Silva, João P. da Silva, Gustavo Santos, Ricardo Terra, and Marco T. Valente. Refdiff 2.0: A multi-language refactoring detection tool. *IEEE Transactions on Software Engineering*, 47(12):2786–2802, December 2021.

[129] Danilo Silva, Nikolaos Tsantalis, and Marco T. 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, November 2016.

[130] Danilo Silva and Marco T. 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, May 2017. IEEE Computer Society Press.

[131] Gustavo Soares, Rohit Gheyi, and Tiago Massoni. Automated behavioral testing of refactoring engines. *IEEE Transactions on Software Engineering*, 39(2):147–162, February 2013.

[132] Gustavo Soares, Rohit Gheyi, Emerson Murphy-Hill, and Brittany Johnson. Comparing approaches to analyze refactoring activity on software repositories. *Journal of Systems and Software*, 86(4):1006–1022, April 2013.

[133] Ian Sommerville. *Software Engineering.* Pearson, tenth edition, 2015.

[134] Ramakrishnan Srikant. *Fast Algorithms for Mining Association Rules and Sequential Patterns*. PhD thesis, University of Wisconsin–Madison, 1996.

[135] Volker Stolz, Larissa Braz, Anna M. Eilertsen, Fernando Macías, and Rohit Gheyi. Modern refactoring. `https://tinyurl.com/373ec2m4`, 2017. Accessed on: December, 2021.

[136] Margaret-Anne Storey and Alexey Zagalsky. Disrupting developer productivity one bot at a time. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '16, pages 928–931, Seattle, USA, November 2016. ACM.

[137] E. Burton Swanson. The dimensions of maintenance. In *Proceedings of the 2nd International Conference on Software Engineering*, ICSE '76, pages 492–497, San Francisco, USA, October 1976. IEEE Computer Society Press.

[138] Gábor Szőke, Gábor Antal, Csaba Nagy, Rudolf Ferenc, and Tibor Gyimóthy. Empirical study on refactoring large-scale industrial systems and its effects on maintainability. *Journal of Systems and Software*, 129(C):107–126, July 2017.

[139] 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 14th International Conference on Computational Science and its Applications*, ICCSA '14, pages 524–540, Guimarães, Portugal, June 2014. Springer International Publishing.

[140] Pang-Ning Tan, Michael Steinbach, Anuj Karpatne, and Vipin Kumar. *Introduction to Data Mining*. Pearson, second edition, 2018.

[141] 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, November 2012. ACM.

[142] 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.

[143] 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.

[144] Garen Torikian, Brandon Black, Brooks Swinnerton, Hailey Somerville, David Celis, and Kyle Daigle. The GitHub GraphQL API. `https://github.blog/2016-09-14-the-github-graphql-api/`, 2016. Accessed on: November, 2021.

[145] Nikolaos Tsantalis, Ameya Ketkar, and Danny Dig. RefactoringMiner 2.0. *IEEE Transactions on Software Engineering*, pages 1–1, July 2020. Early access article.

[146] Nikolaos Tsantalis, Matin Mansouri, Laleh M. Eshkevari, Davood Mazinanian, 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, May 2018. ACM.

[147] 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, July 2019.

[148] Peter Weißgerber and Stephan Diehl. Are refactorings less error-prone than other changes? In *Proceedings of the 2006 International Workshop on Mining Software Repositories*, MSR '06, pages 112–118, Shanghai, China, May 2006. ACM.

[149] Dongkuan Xu and Yingjie Tian. A comprehensive survey of clustering algorithms. *Annals of Data Science*, 2:165–193, August 2015.

[150] 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, May 2015. IEEE Computer Society Press.

[151] Alice Zheng and Amanda Casari. *Feature Engineering for Machine Learning: Principles and Techniques for Data Scientists*. O'Reilly Media, Inc., first edition, 2018.

[152] 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, December 2012. IEEE Computer Society Press.

[153] Jiaxin Zhu, Minghui Zhou, and Audris Mockus. Effectiveness of code contribution: From patch-based to pull-request-based tools. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '16, pages 871–882, Seattle, USA, November 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 [146], 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 [52][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],

[1]The 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 PRs on GitHub [6], and

7. (Q4 2019) a brief ”case history” relating to refactoring-inducing code review in merged PRs 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 [18].

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 [19], such as the Apache Software Foundation, which has completely migrated its projects to GitHub, in February 2019 [16].

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 PRs.

Phase	***Rationale***	***Result***	***Impact for the research***
1	Find an accurate refactoring detection tool for source code written in the Java language	RefactoringMiner [11]	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 PRs	Brief ”case history”	Motivating results towards a case study on merged pull requests
7	Investigate a case study feasibility from GitHub merged PRs	Brief ”case history”	Decision for performing a case study on data from GitHub merged PRs

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

Appendix B

Descriptive Statistics of pull requests data

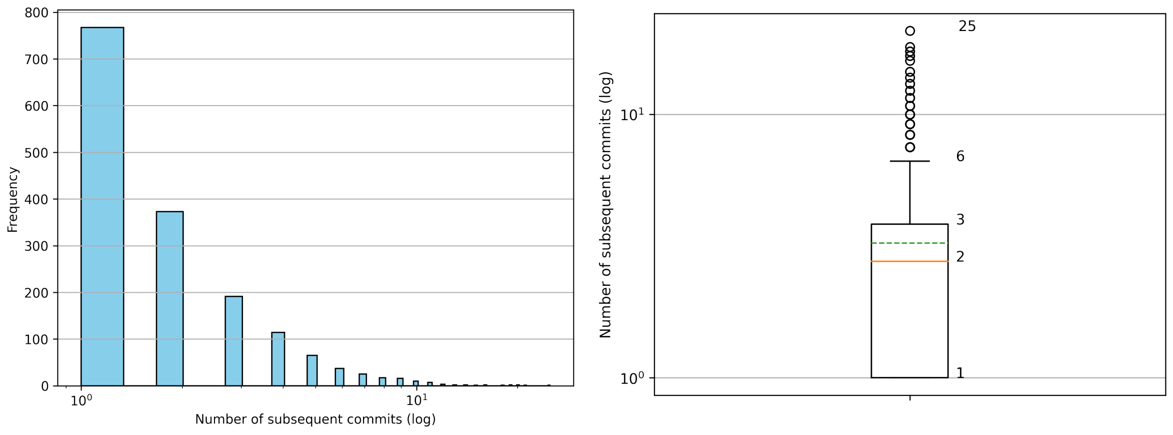
In this appendix, we present the descriptive statistics and data distribution (histogram and boxplot) of our sample of Apache's PRs, by considering the *code review dataset* (Chapter 3, Subsection 3.1.3). In the boxplots, a dashed line denotes the average value, whereas a solid line indicates the median value.

Specifically, we describe number of subsequent commits, number of added lines, number of deleted lines, number of file changes, number of reviewers, number of review comments, length of discussion, time to merge, and number of refactorings.

Number of subsequent commits

Table B.1: Descriptive statistics – number of subsequent commits

Statistic	***Value***
average	2.4
standard deviation	2.4
median	2
interquartile range	2



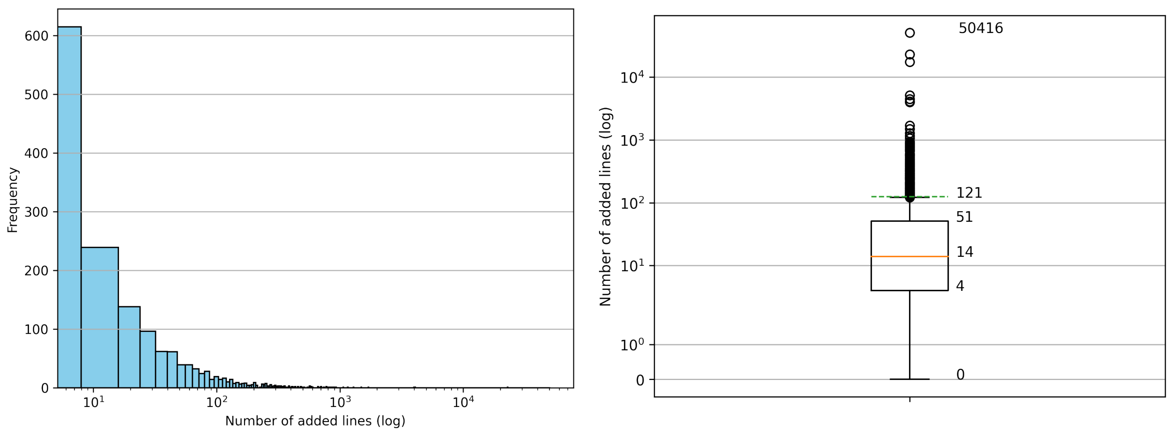
(a) Histogram (b) Boxplot

Figure B.1: Number of subsequent commits in the sample's PRs

Number of Added Lines

Table B.2: Descriptive statistics – number of added lines

Statistic	***Value***
average	125.5
standard deviation	1,453.7
median	14
interquartile range	47



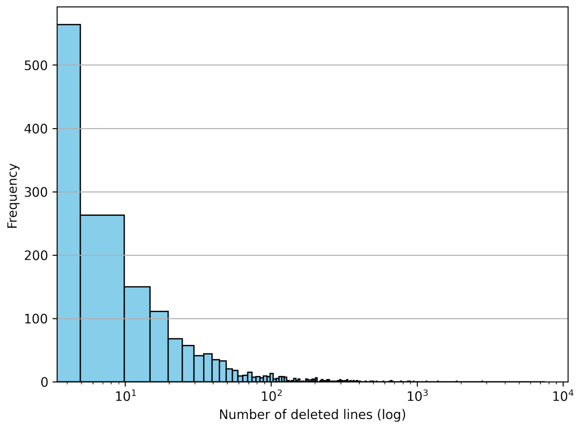
(a) Histogram (b) Boxplot

Figure B.2: Number of added lines in the sample's PRs

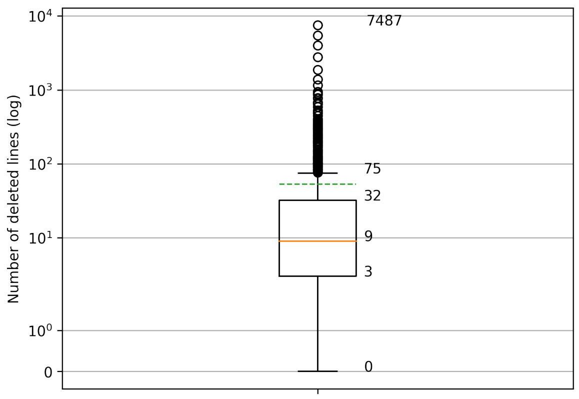
Number of Deleted Lines

Table B.3: Descriptive statistics – number of deleted lines

Statistic	***Value***
average	52.9
standard deviation	278.8
median	9
interquartile range	29



(a) Histogram



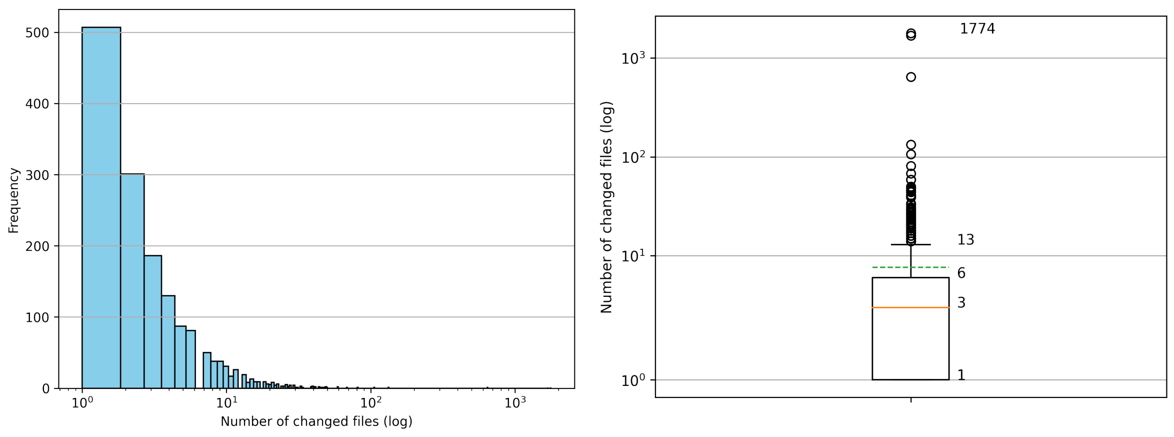
(b) Boxplot

Figure B.3: Number of deleted lines in the sample's PRs

Number of File Changes

Table B.4: Descriptive statistics – number of file changes

Statistic	***Value***
average	7.6
standard deviation	62.8
median	3
interquartile range	5



(a) Histogram

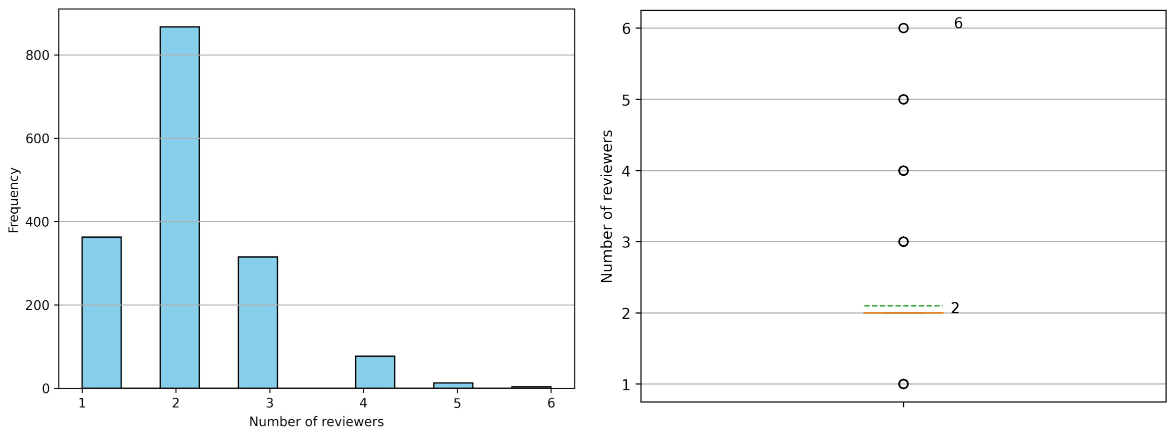
(b) Boxplot

Figure B.4: Number of file changes in the sample's PRs

Number of Reviewers

Table B.5: Descriptive statistics – number of reviewers

Statistic	***Value***
average	2.1
standard deviation	0.8
median	2
interquartile range	0



(a) Histogram

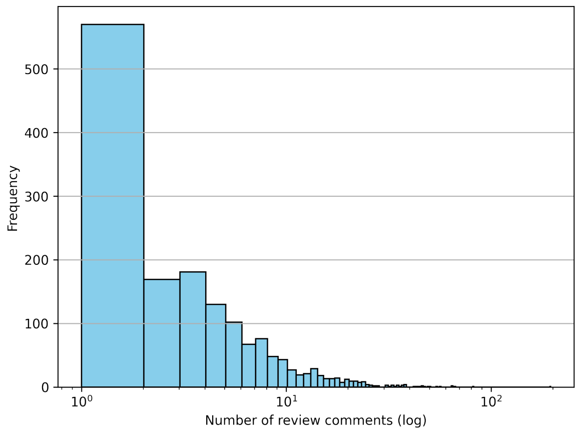
(b) Boxplot

Figure B.5: Number of reviewers in the sample's PRs

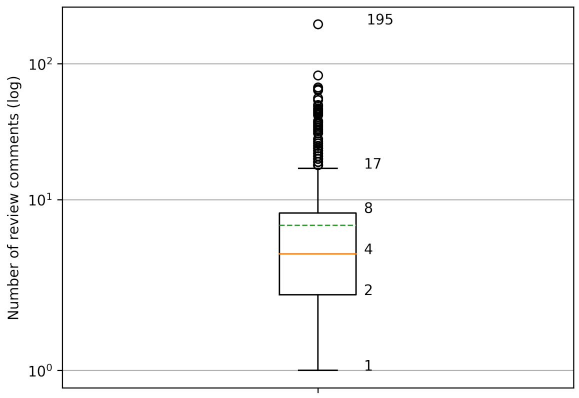
Number of Review Comments

Table B.6: Descriptive statistics – number of review comments

Statistic	***Value***
average	6.5
standard deviation	8.9
median	4
interquartile range	6



(a) Histogram



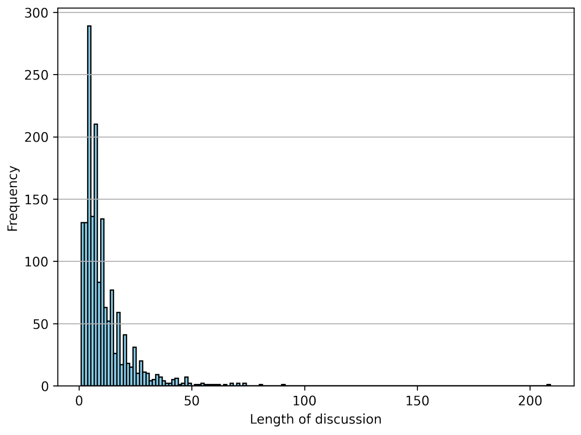
(b) Boxplot

Figure B.6: Number of review comments in the sample's PRs

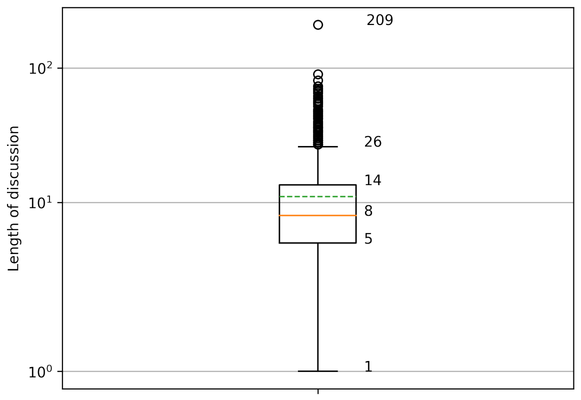
Length of Discussion

Table B.7: Descriptive statistics – length of discussion

Statistic	***Value***
average	11.1
standard deviation	11.4
median	8
interquartile range	8



(a) Histogram



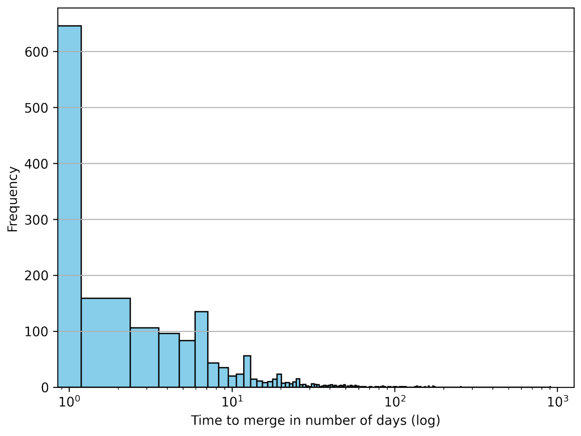
(b) Boxplot

Figure B.7: Length of discussion in the sample's PRs

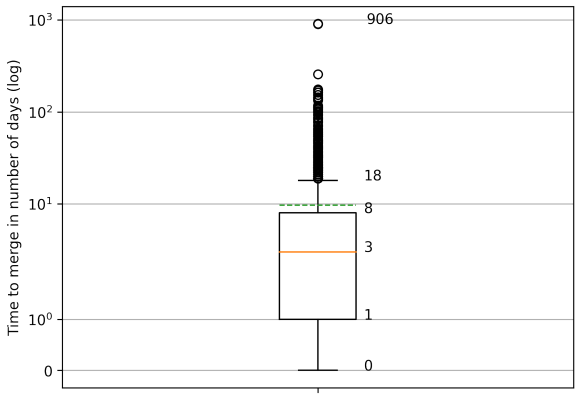
Time to Merge

Table B.8: Descriptive statistics – time to merge

Statistic	***Value***
average	9.6
standard deviation	36.9
median	3
interquartile range	7



(a) Histogram



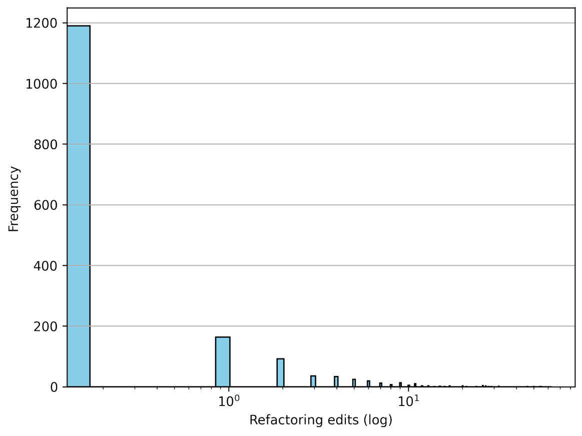
(b) Boxplot

Figure B.8: Time to merge in the sample's PRs

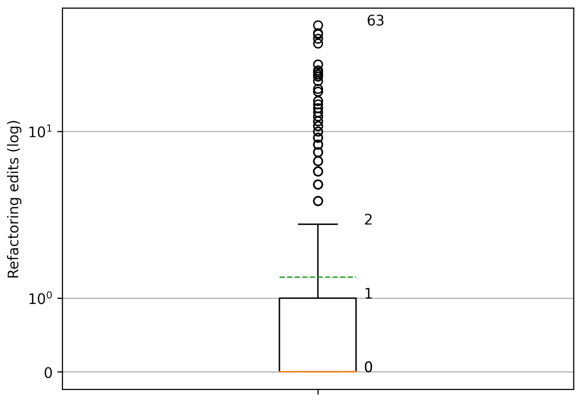
Number of Refactoring Edits

Table B.9: Descriptive statistics – number of refactoring edits

Statistic	***Value***
average	1.3
standard deviation	4.4
median	0
interquartile range	1



(a) Histogram



(b) Boxplot

Figure B.9: Number of refactoring edits in the sample's PRs

Appendix C

Checking for Parametric Tests Assumptions and Results of Statistical Testing of Hypotheses

Following, we present the results from checking for data normality and homogeneity of variance of refactoring-inducing and non-refactoring-inducing PRs, and statistical testing of hypotheses, according to this structure:

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 PRs and non-refactoring-inducing ones 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 PR is a refactoring-inducing PR or a non-refactoring-inducing PR. In case of exploration on the difference between refactoring-inducing PRs and non-refactoring-inducing ones, 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 higher/greater than a randomly selected observation from another sample [93].

3. Deciding whether the null hypothesis is supported or refused.

To check descriptive statistics of the analyzed features from refactoring-inducing and non-refactoring-inducing PRs, see Table 4.4.

H_1 Refactoring-inducing pull requests are more likely to have more added lines than non-refactoring-inducing pull requests

1. Null and alternative hypotheses:

   H_{1_0} Refactoring-inducing and non-refactoring-inducing pull requests are equally likely to have more added lines.

   H_{1_a} Refactoring-inducing pull requests are more likely to have more added lines than non-refactoring-inducing pull requests.

2. Statistical tests. Figure C.1 displays the distribution of number of added lines according to the presence of refactoring edits in the PRs. The computed 95% CI [35.0, 52.5] of the difference in medians does not contain 0, thus there is a statistically significant difference in the number of added lines between refactoring-inducing and non-refactoring-inducing PRs. Table C.1 presents the computed statistics and p-value for each applied statistical test.

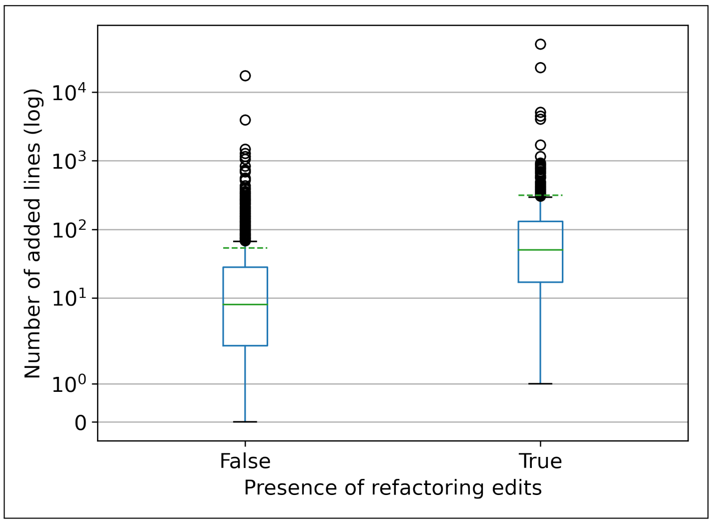


Figure C.1: Number of added lines by presence of refactorings

Table C.1: Statistical tests output – number of added lines by presence of refactorings

Statistical test	***Statistic***	***p-value***
Shapiro-Wilk	False: 0.05	0.0
	True: 0.07	$7.19 \times e^{-42}$
Levene's	9.23	0.002
Mann Whitney	$0.42 \times e^{+6}$	$9.04 \times e^{-74}$

3. Decision. Based on Mann Whitney one-sided test, the result provides support for reject the null hypothesis in favor to the alternative hypothesis ($U = 0.42 \times e^{+6}$, $p < .05$). Thus, there is evidence that refactoring-inducing PRs are significantly more likely to have a higher number of added lines than non-refactoring-inducing PRs. There is a 78.9% probability (CLES = 0.789) that a random observation of the number of added lines from refactoring-inducing PRs will be higher than one from non-refactoring-inducing PRs, at the population level.

H_2 Refactoring-inducing pull requests are more likely to have more deleted lines than the non-refactoring-inducing pull requests

1. Null and alternative hypotheses:

H_{2_0} Refactoring-inducing pull requests and non-refactoring-inducing pull requests are equally likely to have more deleted lines.

H_{2_a} Refactoring-inducing pull requests are more likely to have more deleted lines than the non-refactoring-inducing pull requests.

2. Statistical tests. Figure C.2 presents the distribution of number of deleted lines according to the presence of refactoring edits in the PRs. The computed 95% CI [20.0, 31.0] of the difference in medians does not contain 0, thus there is a statistically significant difference in the number of deleted lines between refactoring-inducing and non-refactoring-inducing PRs. Table C.2 presents the computed statistics and p-value for each applied statistical test.

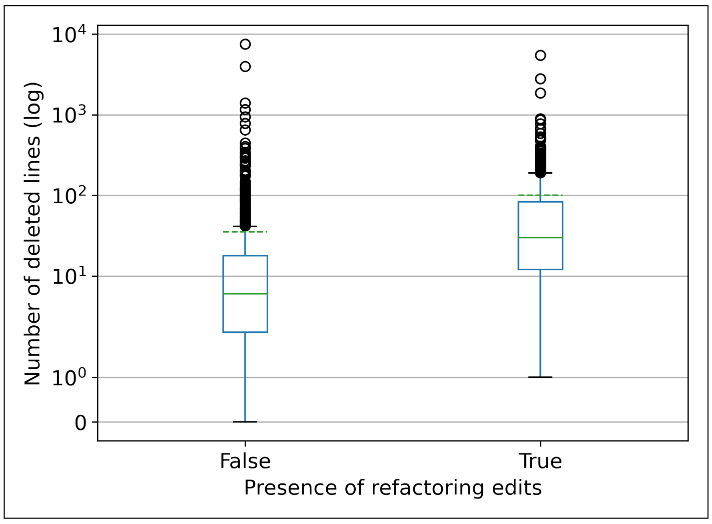


Figure C.2: Number of deleted lines by presence of refactorings

Table C.2: Statistical tests output – number of deleted lines by presence of refactorings

Statistical test	***Statistic***	***p-value***
Shapiro-Wilk	False: 0.08 True: 0.24	0.0 $4.32 \times e^{-39}$
Levene's	12.49	0.0004
Mann Whitney	$0.42 \times e^{+6}$	$7.02 \times e^{-69}$

3. Decision. Based on Mann Whitney one-sided test, the result provides support to reject the null hypothesis in favor to the alternative hypothesis ($U = 0.42 \times e^{+6}$,

$p < .05$). Therefore, there is evidence that refactoring-inducing PRs are significantly more likely to have a higher number of deleted lines than non-refactoring-inducing PRs. There is a 77.9% probability (CLES = 0.779) that a random observation of the number of deleted lines from refactoring-inducing PRs will be higher than one from non-refactoring-inducing PRs, at the population level.

H_3 Refactoring-inducing pull requests are more likely to have more file changes than the non-refactoring-inducing pull requests

1. Null and alternative hypotheses:

   H_{3_0} Refactoring-inducing and non-refactoring-inducing pull requests are equally likely to have more file changes.

   H_{3_a} Refactoring-inducing pull requests are more likely to have more file changes than the non-refactoring-inducing pull requests.

2. Statistical tests. Figure C.3 shows the distribution of number of file changes according to the presence of refactoring edits in the PRs. The computed 95% CI [3.0, 4.0] of the difference in medians does not contain 0, thus there is a statistically significant difference in the number of file changes between refactoring-inducing and non-refactoring-inducing PRs. Table C.3 presents the computed statistics and p-value for each applied statistical test.

Table C.3: Statistical tests output – number of file changes by presence of refactorings

Statistical test	***Statistic***	***p-value***
Shapiro-Wilk	False: 0.03	0.0
	True: 0.08	$9.67 \times e^{-42}$
Levene's	5.17	0.02
Mann Whitney	$0.41 \times e^{+6}$	$1.98 \times e^{-62}$

3. Decision. Based on Mann Whitney one-sided test, the result provides support to reject the null hypothesis in favor to the alternative hypothesis ($U = 0.41 \times e^{+6}$, $p < .05$). Therefore, there is evidence that refactoring-inducing PRs are significantly more likely to have a higher number of file changes than non-refactoring-

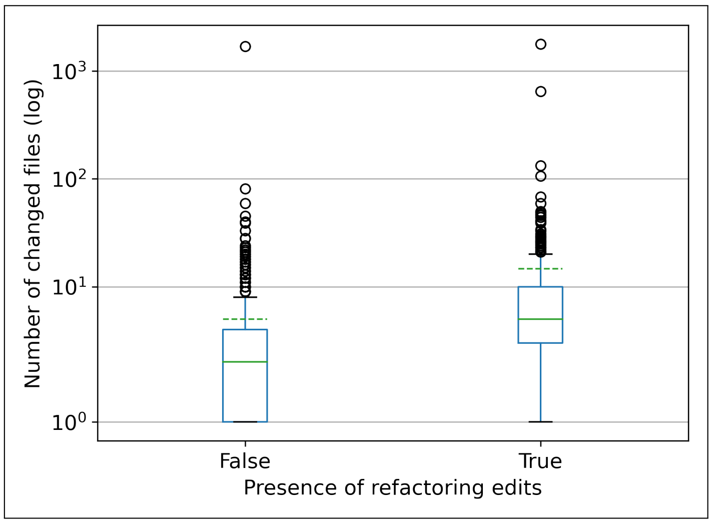


Figure C.3: Number of file changes by presence of refactorings

inducing PRs. There is a 76.1% probability (CLES = 0.761) that a random observation of the number of changed files from refactoring-inducing PRs will be higher than one from non-refactoring-inducing PRs, at the population level.

H_4 Refactoring-inducing pull requests are more likely to have more subsequent commits than non-refactoring-inducing pull requests

1. Null and alternative hypotheses:

   H_{4_0} Refactoring-inducing and non-refactoring-inducing pull requests are equally likely to have more subsequent commits.

   H_{4_a} Refactoring-inducing pull requests are more likely to have more subsequent commits than the non-refactoring-inducing pull requests.

2. Statistical tests. Figure C.4 displays the distribution of number of subsequent commits by the presence of refactoring edits in the PRs. The computed 95% CI [1.0, 2.0] of the difference in medians does not contain 0, so there is a statistically significant difference in the number of subsequent commits between refactoring-inducing and non-refactoring-inducing PRs. Table C.4 presents the computed statistics and p-value for each applied statistical test.

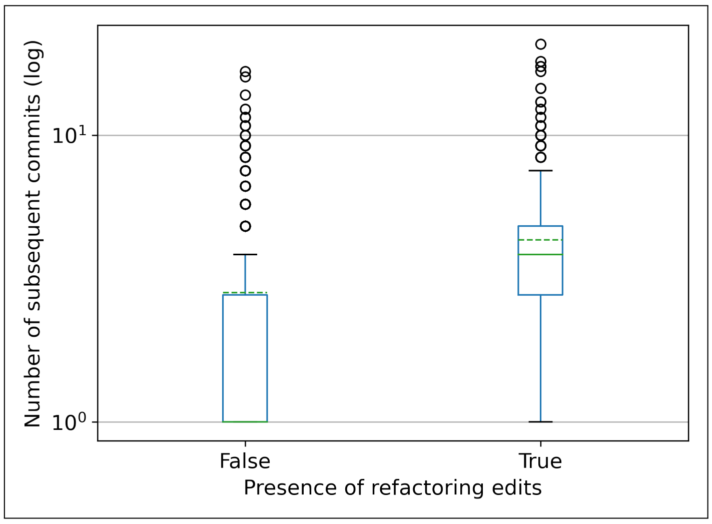


Figure C.4: Number of subsequent commits by presence of refactorings

Table C.4: Statistical tests output – number of subsequent commits by presence of refactorings

Statistical test	***Statistic***	***p-value***
Shapiro-Wilk	False: 0.60	0.0
	True: 0.71	$3.03 \times e^{-27}$
Levene's	62.96	$3.88 \times e^{-15}$
Mann Whitney	$0.37 \times e^{+6}$	$2.59 \times e^{-37}$

3. Decision. Based on Mann Whitney one-sided test, the result provides support for reject the null hypothesis in favor to the alternative hypothesis (U = $0.37 \times e^{+6}$, p < .05). Thus, there is evidence that refactoring-inducing PRs are significantly more likely to have a higher number of subsequent commits than non-refactoring-inducing PRs. There is a 69.1% probability (CLES = 0.691) that a random observation of the number of subsequent commits from refactoring-inducing PRs will be higher than one from non-refactoring-inducing PRs, at the population level.

H_5 Refactoring-inducing PRs are more likely to have more review comments than the non-refactoring-inducing pull requests

1. Null and alternative hypotheses:

   H_{5_0} Refactoring-inducing and non-refactoring-inducing pull requests are equally likely to have more review comments.

   H_{5_a} Refactoring-inducing pull requests are more likely to have more review comments than the non-refactoring-inducing pull requests.

2. Statistical tests. Figure C.5 displays the distribution of number of review comments according to the presence of refactoring edits in the PRs. The computed 95% CI [2.0, 4.0] of the difference in medians does not contain 0, therefore there is a statistically significant difference in the number of review comments between refactoring-inducing and non-refactoring-inducing PRs. Table C.5 presents the computed statistics and p-value for each applied statistical test.

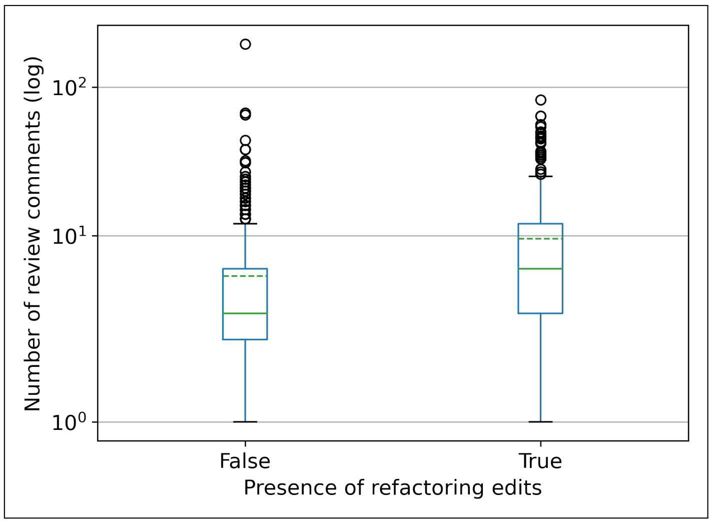


Figure C.5: Number of review comments by presence of refactorings

Table C.5: Statistical tests output – number of review comments by presence of refactorings

Statistical test	***Statistic***	***p-value***
Shapiro-Wilk	False: 0.41	0.0
	True: 0.71	$2.86 \times e^{-27}$
Levene's	44.22	$3.98 \times e^{-11}$
Mann Whitney	$0.35 \times e^{+6}$	$3.81 \times e^{-23}$

3. Decision. Based on Mann Whitney one-sided test, the result provides support for reject the null hypothesis in favor to the alternative hypothesis ($U = 0.35 \times e^{+6}$, $p < .05$). Thus, there is evidence that refactoring-inducing PRs are significantly more likely to have a higher number of review comments than non-refactoring-inducing PRs. There is a 65.6% probability (CLES = 0.656) that a random observation of the number of review comments from refactoring-inducing PRs will be higher than one from non-refactoring-inducing PRs, at the population level.

H_6 Refactoring-inducing pull requests are more likely to present a lengthier discussion than the non-refactoring-inducing pull requests

1. Null and alternative hypotheses:

   H_{6_0} Refactoring-inducing and non-refactoring-inducing pull requests are equally likely to present lengthier discussion.

   H_{6_a} Refactoring-inducing pull requests are more likely to present a lengthier discussion than the non-refactoring-inducing pull requests.

2. Statistical tests. Figure C.6 displays the distribution of length of discussion according to the presence of refactoring edits in the PRs. The computed 95% CI [3.0, 5.0] of the difference in medians does not contain 0, therefore there is statistically significant difference in the length of discussion between refactoring-inducing and non-refactoring-inducing PRs. Table C.6 presents the computed statistics and p-value for each applied statistical test.

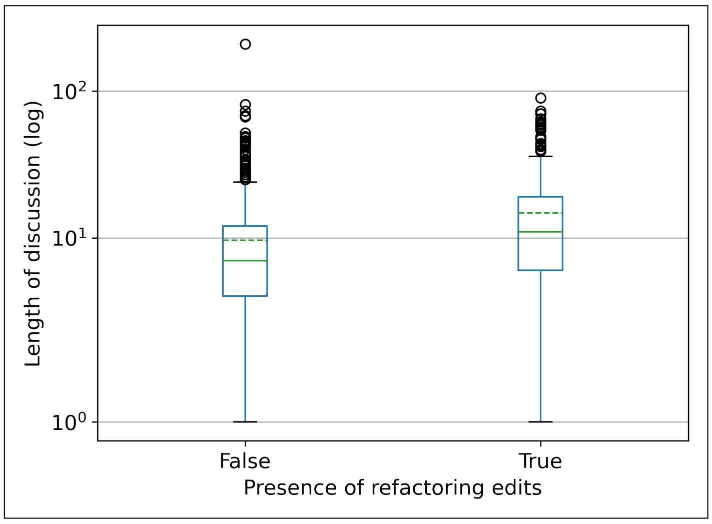


Figure C.6: Length of discussion by presence of refactorings

Table C.6: Statistical tests output – length of discussion by presence of refactorings

Statistical test	***Statistic***	***p-value***
Shapiro-Wilk	False: 0.58	0.0
	True: 0.79	$2.87 \times e^{-23}$
Levene's	32.41	$1.47 \times e^{-8}$
Mann Whitney	$0.35 \times e^{+6}$	$4.32 \times e^{-22}$

3. Decision. Based on Mann Whitney one-sided test, the result provides support to reject the null hypothesis ($U = 0.35 \times e^{+6}$, $p < .05$). In particular, there is evidence that refactoring-inducing PRs are significantly more likely to have a lengthier discussion than non-refactoring-inducing PRs in opposition to the alternative hypothesis. There is a 65.3% probability (CLES = 0.653) that a random observation of the length of discussion from refactoring-inducing PRs will be larger than one from non-refactoring-inducing PRs, at the population level.

H_7 Refactoring-inducing and non-refactoring-inducing are equally likely to have a higher number of reviewers

1. Null and alternative hypotheses:

H_{7_0} Refactoring-inducing and non-refactoring-inducing pull requests are not equally likely to have a higher number of reviewers.

H_{7_a} Refactoring-inducing and non-refactoring-inducing pull requests are equally likely to have a higher number of reviewers.

2. Statistical tests. Figure C.7 displays the distribution of number of reviewers according to the presence of refactoring edits in the PRs. The computed 95% CI [0.0, 0.0] of the difference in medians contains 0, thus there is no statistically significant difference in the number of reviewers between refactoring-inducing PRs and non-refactoring-inducing ones. Table C.7 presents the computed statistics and p-value for each applied statistical test.

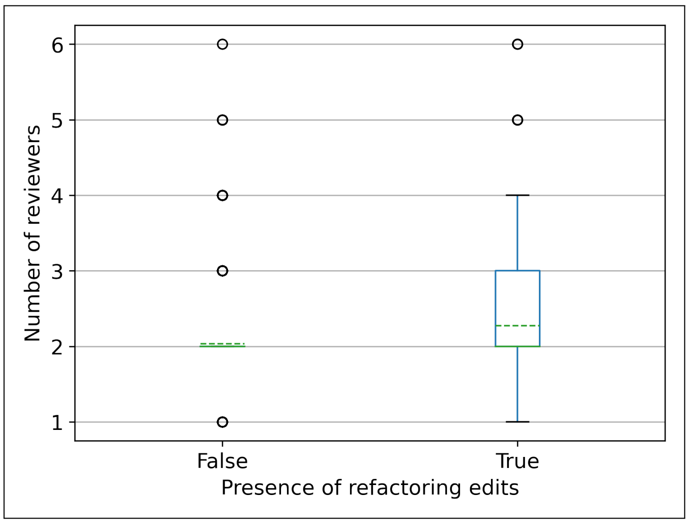


Figure C.7: Number of reviewers by presence of refactorings

Table C.7: Statistical tests output – number of reviewers by presence of refactorings

Statistical test	***Statistic***	***p-value***
Shapiro-Wilk	False: 0.82 True: 0.86	$2.18 \times e^{-34}$ $1.53 \times e^{-19}$
Levene's	19.24	$1.22 \times e^{-5}$
Mann Whitney	$0.30 \times e^{+6}$	$2.00 \times e^{-6}$

3. Decision. Based on Mann Whitney two-sided test, the result provides support to reject the null hypothesis in favor to the alternative hypothesis ($U = 0.30 \times e^{+6}$,

$p < .05$). Thus, there is evidence that refactoring-inducing and non-refactoring-inducing PRs are significantly likely to have a higher number of reviewers. There is a 56.7% probability (CLES = 0.567) that a random observation of the number of reviewers from refactoring-inducing PRs will be as high as one from non-refactoring-inducing PRs, at the population level.

H_8 Refactoring-inducing pull requests are more likely to take a longer time to merge than non-refactoring-inducing pull requests

1. Null and alternative hypotheses:

   H_{8_0} Refactoring-inducing and non-refactoring-inducing pull requests are equally likely to take a longer time to merge.

   H_{8_a} Refactoring-inducing pull requests are more likely to take a longer time to merge than the non-refactoring-inducing pull requests.

2. Statistical tests. Figure C.8 displays the distribution of time to merge according to the presence of refactoring edits in the PRs. The computed 95% CI [1.0, 3.0] of the difference in medians does not contain 0, therefore there is statistically significant difference in time to merge between refactoring-inducing and non-refactoring-inducing PRs. Table C.8 lists the computed statistics and p-value for each applied statistical test. It is worth noting that only one of the assumptions, homogeneity of variances, was met ($F = 0.09$, $p > .05$).

Table C.8: Statistical tests output – time to merge by presence of refactorings

Statistical test	***Statistic***	***p-value***
Shapiro-Wilk	False: 0.20	0.0
	True: 0.19	$6.19 \times e^{-40}$
Levene's	2.81	0.09
Mann Whitney	$0.31 \times e^{+6}$	$1.00 \times e^{-6}$

3. Decision. Based on Mann Whitney one-sided test, the result provides support for reject the null hypothesis ($U = 0.31 \times e^{+6}$, $p < .05$). Particularly, we found evidence that refactoring-inducing PRs are significantly more likely to take a longer

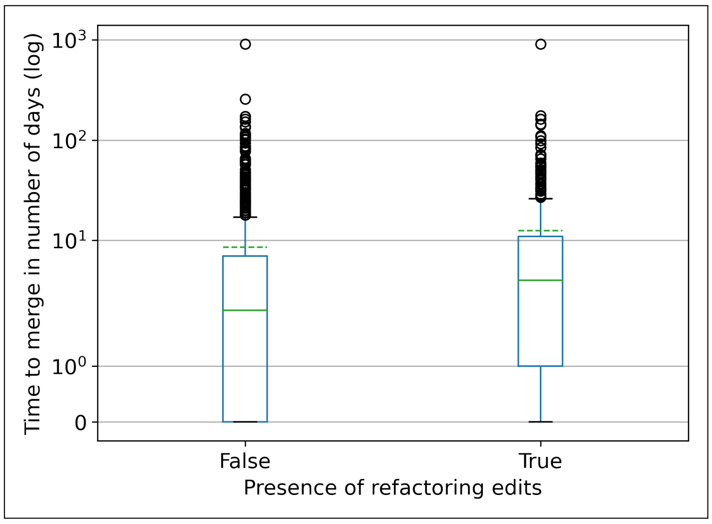


Figure C.8: Time to merge by presence of refactorings

time to merge than non-refactoring-inducing PRs. There is a 57.4% probability (CLES = 0.574) that a random observation of time to merge from refactoring-inducing PRs will be longer than one from non-refactoring-inducing PRs, at the population level.

Appendix D

Complements to the Characterization of Code Review in Refactoring-Inducing Pull Requests

Following, we provide more details regarding the characteristics of code review in refactoring-inducing PRs in complement to Chapter 5.

As displayed in Tables D.1 and D.2, the magnitude of both refactoring-inducing and non-refactoring-inducing PRs, in our purposive samples, increases in the following order: *sample 2* < *sample 1* < *sample 4* < *sample 3*, when considering size-related aspects (number of subsequent commits, number of file changes, number of added lines, and number of deleted lines).

Table D.1: Descriptive statistics (refactoring-inducing PRs)

Sample	***Attribute***	***Average***	***SD***	***Median***	***IQR***
Sample 1	*No. of reviewers*	2	0	2	0
	No. of review comments	5	0	5	0
	No. of subsequent commits	2.4	1.2	2	1.2
	Time to merge	3.2	6.1	1	2.2
	No. of file changes	8.2	8.7	5	7
	No. of added lines	93.1	170.5	31	56.7
	No. of deleted lines	43.3	58.3	13.5	39.5

Continued on next page

Table D.1 – continued from previous page

Sample	***Attribute***	***Average***	***SD***	***Median***	***IQR***
	No. of refactoring edits	5.7	7.2	2.5	5
Sample 2	*No. of reviewers*	1.8	0.6	2	0.5
	No. of review comments	4	4.3	2	2
	No. of subsequent commits	1	0	1	0
	Time to merge	3.5	6.6	1	3
	No. of file changes	1.7	0.6	2	1
	No. of added lines	35.4	63.5	6	21.5
	No. of deleted lines	26.9	56.4	6	13
	No. of refactoring edits	1	0	1	0
Sample 3	*No. of reviewers*	3.1	1.1	3	1
	No. of review comments	15.1	13.7	10	14
	No. of subsequent commits	5.8	3.6	5	6
	Time to merge	11.5	14.8	6	8
	No. of file changes	26.5	34.6	16	22
	No. of added lines	550	1198.8	143	332
	No. of deleted lines	297.9	490.7	104	281
	No. of refactoring edits	16.1	18.6	10	12
Sample 4	*No. of reviewers*	2.3	0.9	2	1
	No. of review comments	8.7	5.8	7	8.7
	No. of subsequent commits	3.8	3.3	3	2
	Time to merge	22.4	37	6.5	15.7
	No. of file changes	9.2	13.2	5	6
	No. of added lines	288.5	811.8	40	109.2
	No. of deleted lines	56.1	80.5	24.5	28.7
	No. of refactoring edits	2.9	1.1	2.5	2

Table D.2: Descriptive statistics (non-refactoring-inducing PRs)

Sample	***Attribute***	***Average***	***SD***	***Median***	***IQR***
Sample 1	*No. of reviewers*	2	0	2	0
	No. of review comments	5	0	5	0
	No. of subsequent commits	1.4	0.5	1	1

Continued on next page

Table D.2 – continued from previous page

Sample	***Attribute***	***Average***	***SD***	***Median***	***IQR***
	Time to merge	2.5	4	0.5	3
	No. of file changes	2.9	2	2	2
	No. of added lines	17.5	19.2	8.5	24.5
	No. of deleted lines	16.6	19.2	9	24.5
Sample 2	*No. of reviewers*	2.1	0.6	2	0
	No. of review comments	4.8	4.2	4	4
	No. of subsequent commits	1	0	1	0
	Time to merge	2.9	2.3	3	4
	No. of file changes	2.1	1.7	1	1
	No. of added lines	115.1	293	2	46
	No. of deleted lines	107.5	295.3	3	8
Sample 3	*No. of reviewers*	2.8	0.9	3	1
	No. of review comments	10	0	10	0
	No. of subsequent commits	3.8	1.9	3	2
	Time to merge	10.9	23.2	2	7
	No. of file changes	7.7	4.6	6	4
	No. of added lines	47.2	48.5	23	40
	No. of deleted lines	29.9	39.4	18	16
Sample 4	*No. of reviewers*	2.4	0.7	2	1
	No. of review comments	7	0	7	0
	No. of subsequent commits	2.3	2.2	2	2
	Time to merge	6.8	8.1	3	10.5
	No. of file changes	4.5	4.9	3	3.7
	No. of added lines	53.3	90.6	21	20.2
	No. of deleted lines	30.7	62.1	6.5	22

Refactoring-inducing and non-refactoring-inducing PRs comprise the three *primary types of change* (adaptive, corrective, and perfective), as indicated in Table D.3.

Table D.3: Type of changes by category of PRs

Sample	**Type of changes**		
	Adaptive	**Corrective**	**Perfective**
Refactoring-inducing PRs			
Sample 1	3/13 (23.1%)	5/13 (38.5%)	4/13 (30.8%)
Sample 2	4/11 (36.4%)	4/11 (36.4%)	2/11 (18.2%)
Sample 3	7/13 (53.8%)	3/13 (23.1%)	2/13 (15.4%)
Sample 4	11/28 (39.2%)	8/28 (28.6%)	8/28 (28.6%)
All samples	*25/65 (38.5%)*[1]	*20/65 (30.8%)*[2]	*16/65 (24.6%)*[3]
non-Refactoring-inducing PRs			
Sample 1	none	4/4 (100.0%)	none
Sample 2	none	6/6 (100.0%)	none
Sample 3	6/11 (54.5%)	4/11 (36.4%)	1/11 (9.1%)
Sample 4	9/17 (52.9%)	8/17 (47.1%)	none

Continued on next page

[1]Flink #7945, Hadoop #942, Samza #1030 (sample 1); Beam #4458, Incubator-Pinot #479, Samza #1051, Servicecomb-Java-Chassis #346 (sample 2); Beam #6261, Flink #8222, Incubator-Iceberg #119, Kafka #4735, Kafka #4757, Kafka #5590, Servicecomb-Java-Chassis #678 (sample 3); Avro #525, Dubbo #4099, Kafka #5501, Knox #69, Logging-log4j #213, Rocketmq-Externals #45, Sling-Org-Apache-Sling-Feature-Analyser #16, Struts #43, Tinkerpop #893, Tinkerpop #1110, Tomee #407 (sample 4).

[2]Commons-Text #39, Flink #7970, Flink #7971, Flink #9143, Fluo #837 (sample 1); Beam #4407, Brooklyn-Server #1049, Kafka #5784, Kafka #7132 (sample 2); Incubator-Iceberg #183, Kafka #6657, Usergrid #102 (sample 3); Dubbo #2445, Dubbo #3174, Dubbo #3257, Flink #7165, Kafka #4796, Kafka #5946, Kafka #6848, Kafka #6853 (sample 4).

[3]Beam #4460, Dubbo #3299, Incubator-Iceberg #254, Kafka #5194 (sample 1); Dubbo #3185, Kafka #5423 (sample 2); Cloudstack #2071, Cloudstack #3454 (sample 3); Accumulo-Examples #19, Brooklyn-Server #964, Cloudstack #2833, Flink #8620, Kafka #4574, Knox #74, Tika #234, Tomee #89 (sample 4).

Table D.3 – continued from previous page

Sample	Type of changes		
	Adaptive	**Corrective**	**Perfective**
All samples	*15/38 (39.5%)*[4]	*22/38 (57.9%)*[5]	*1/38 (2.6%)*[6]

We found self-affirmed minor PRs (Table D.4) and self-affirmed minor review comments (Table D.5) in both refactoring-inducing and non-refactoring-inducing PRs. We observed self-affirmed review comments that induced edits of *Rename* (Brooklyn-Server #964, Flink #7945, Flink #8620, Kafka #5784, Kafka #6848), *Split* (Brooklyn-Server #1049), *Inline* (Dubbo #3185), and *Extract* (Flink #8620, Kafka #4735).

Table D.4: Presence of self-affirmed minor PRs

Sample	**Refactoring-inducing pull requests**	**Pull requests**	**non-Refactoring-inducing pull requests**	**Pull requests**
Sample 1	1/13 (7.7%)	Kafka #5194	2/7 (28.6%)	Brooklyn-server #411, Kafka #5111
Sample 2	1/11 (9.1%)	Kafka #5423	0/9 (0%)	
Sample 3	1/13 (7.7%)	Kafka #5590	1/13 (7.7%)	Kafka #6438

Continued on next page

[4]Dubbo #3184, Dubbo #3447, Dubbo #4208, Incubator-Iotdb #67, Servicecomb-Java-Chassis #691, Servicecomb-Java-Chassis #744 (sample 3); Accumulo-Testing #21, Beam #6317, Flink #2096, Incubator-Pinot #880, Kafka #4430, Plc4x #9, Servicecomb-Java-Chassis #969, Tinkerpop #282, Tomee #283 (sample 4).

[5]Brooklyn-server #411, Dubbo #4870, Flink #91, Kafka #5111 (sample 1); Beam #6050, Dubbo #3317, Kafka #5219, Kafka #6565, Kafka #6818, Tinkerpop #524 (sample 2); Cloudstack #2553, Cloudstack #2714, Cloudstack #3276, Fluo #929 (sample 3); Accumulo-Examples #50, Cloudstack #3333, Cloudstack #3430, Dubbo #3331, Dubbo #3748, Kafka #6427, Servicecomb-Java-Chassis #698, Struts #191 (sample 4).

[6]Kafka #6438 (sample 3).

Table D.4 – continued from previous page

Sample	Refactoring-inducing PRs	Pull requests	non-Refactoring-inducing pull requests	Pull requests
Sample 4	2/28 (7.1%)	Kafka #4574, Kafka #6853	3/24 (12.5%)	Kafka #5368, Kafka #6427, Kafka #6758
All samples	*5/65 (7.7%)*		*6/53 (11.3%)*	

Table D.5: Presence of self-affirmed minor review comments

Sample	Refactoring-inducing pull requests	Pull requests	non-Refactoring-inducing pull requests	Pull requests
Sample 1	1/13 (7.7%)	Flink #7945	2/7 (28.6%)	Brooklyn-server #4111, Flink #91
Sample 2	3/11 (27.3%)	Brooklyn-Server #1049, Dubbo #3185, Kafka #5784	2/9 (22.2%)	Beam #5785, Flink #9451
Sample 3	3/13 (23.1%)	Kafka #4735, Kafka #4757, Kafka #6657	0/13 (0%)	

Continued on next page

Table D.5 – continued from previous page

Sample	Refactoring-inducing pull requests	Pull requests	non-Refactoring-inducing pull requests	Pull requests
Sample 4	8/28 (28.6%)	Brooklyn-Server #964, Cloudstack #2833, Flink #8620, Kafka #4574, Kafka #4796, Kafka #5501, Kafka #6848, Logging-Log4j #213	2/24 (8.3%)	Beam #6317, Kafka #4430
All samples	*15/65 (23.1%)*		*6/53 (11.3%)*	

We identified that 8/65 (12.3%) of refactoring-inducing PRs and 1/53 (1.9%) of non-refactoring-inducing PRs ran a *code review bot* (Table D.6).

Table D.6: Presence of code review bot in PRs

Sample	Proportion	PRs
Refactoring-inducing PRs		
Sample 1	5/13 (38.5%)	Flink #7970, Flink #7971, Flink #7945, Flink #9143, Hadoop #942
Sample 2	none	
Sample 3	1/13 (7.7%)	Flink #8222
Sample 4	2/28 (7.1%)	Flink #8620, Struts #43
All samples	*8/65 (12.3%)*	
non-Refactoring-inducing PRs		
Sample 1	none	

Continued on next page

Table D.6 – continued from previous page

Sample	Proportion	PRs
Sample 2	1/9 (11.1%)	Flink #9451
Sample 3	none	
Sample 4	none	
All samples	*1/53 (1.9%)*	

We investigated the number of contributions (Table D.7) and Apache's roles (Table D.8) of the PR participants.

Table D.7: Descriptive statistics of participants of PRs by category

Sample	Category	Stats	Author	Reviewer
Sample 1	*Refactoring-inducing PRs*	Average	668.9	2801.6
		SD	853.3	3417.6
		Median	424	1414
		IQR	794	3698
		Outliers	3127	14442
	Non-refactoring-inducing PRs	Average	5375.6	2907.6
		SD	10982.5	4358.2
		Median	511	1373
		IQR	12862.5	1228
		Outliers	none	12678
Sample 2	*Refactoring-inducing PRs*	Average	758.2	956.1
		SD	547.5	974.9
		Median	1032	438
		IQR	1024.5	1909
		Outliers	3699, 28964	10853, 14688
	Non-refactoring-inducing PRs	Average	3137	1645.5
		SD	4058.6	1311.3
		Median	1674.5	1639
		IQR	3499	2765
		Outliers	11965	none
Sample 3	*Refactoring-inducing PRs*	Average	723.5	2683.1
		SD	1447.2	4637.7

Continued on next page

Table D.7 – continued from previous page

Sample	Category	Stats	Author	Reviewer
		Median	99	912
		IQR	697	2543.5
		Outliers	5254	14216, 15197, 18995
	Non-refactoring-inducing PRs	Average	2091.1	2432.4
		SD	3114.8	4872.1
		Median	380.5	828
		IQR	3752	1190
		Outliers	none	5808, 18728, 18866
Sample 4	*Refactoring-inducing PRs*	Average	839.3	1657.1
		SD	978.1	1532.3
		Median	476	1302
		IQR	1329	2387
		Outliers	7171, 12467, 13205, 13955, 14288	12305, 14188, 14215, 28364
	Non-refactoring-inducing PRs	Average	940.3	783.5
		SD	1312.1	522.8
		Median	215	755
		IQR	1728.5	903
		Outliers	11068, 30032	2715, 3009, 3917, 4614, 4716, 9892, 30104

Table D.8: Apache roles of the participants of PRs by category

Sample	Category	Author	Reviewer
Sample 1	*Refactoring-inducing PRs*	2 committers 1 commiter/PMC	6 PMC 2 committers 3 commiters/PMC

Continued on next page

Table D.8 – continued from previous page

Sample	Category	Author	Reviewer
	Non-refactoring-inducing PRs	1 committer/PMC	1 PMC 2 committer/PMC
Sample 2	*Refactoring-inducing PRs*	1 committer 1 commiter/PMC	4 PMC 3 commiters/PMC
	Non-refactoring-inducing PRs	3 committer/PMC	4 PMC 4 committers/PMC
Sample 3	*Refactoring-inducing PRs*	none	1 committer 10 PMC 3 commiters/PMC
	Non-refactoring-inducing PRs	none	1 committer 14 PMC 1 committers/PMC
Sample 4	*Refactoring-inducing PRs*	1 committer 2 committers/PMC	10 PMC 12 committers/PMC
	Non-refactoring-inducing PRs	1 PMC 2 committers/PMC	7 PMC 2 committers/PMC

We also explored the number of contributions in the three subgroups of refactoring-inducing PRs: with refactorings led by the authors, with refactorings induced by code review, and with refactorings both led by the authors and induced by code review (Table D.9).

Table D.9: Descriptive statistics of participants by subgroup of refactoring-inducing PRs

Subgroup	Stats	Author	Reviewer
Authors led refactorings	Average	1621.4	2642.7
	SD	3242.1	3549.3
	Median	904	1388
	IQR	1293.5	2443.5
	Outliers	5334, 13205	12305, 14188
Code review induced refactorings	Average	3481.1	3204.6

Continued on next page

Table D.9 – continued from previous page

Subgroup	Stats	Author	Reviewer
	SD	6036.6	5370.8
	Median	1123	1280
	IQR	3111.5	2721
	Outliers	12467, 13955, 14288, 28964	8132, 10853, 14215, 14422, 14688, 15197, 18995, 28364
Authors and reviewers led/induced refactorings	Average	458.5	1800.1
	SD	703.1	2647.1
	Median	173	871
	IQR	486	1720
	Outliers	1451, 2610	4706, 4910, 5906, 14216

We observed the age of the PRs, by computing the difference between the creation date of repositories and the creation date of PRs, in number of years (Table D.10).

Table D.10: Age of PRs (in number of years)

Sample	Statistic	Refactoring-inducing PRs	non-Refactoring-inducing PRs
Sample 1	*Average*	6	4.8
	SD	2.6	1.8
	Median	6.2	5.4
	IQR	4.7	3
	Outliers	none	none
Sample 2	*Average*	5.4	5.3
	SD	3.3	1.8
	Median	5.7	5.5
	IQR	3.5	2.9
	Outliers	12.9	none
Sample 3	*Average*	4.9	5.6
	SD	2.6	2.7

Continued on next page

Table D.10 – continued from previous page

Sample	**Statistic**	**Refactoring-inducing PRs**	**non-Refactoring-inducing PRs**
	Median	5.2	6.6
	IQR	4.4	4.8
	Outliers	none	None
Sample 4	*Average*	6.6	4.9
	SD	2.8	3.2
	Median	6.3	4.6
	IQR	2.7	4.0
	Outliers	0.9, 1.2,13.0	12.8