



**UNIVERSIDADE FEDERAL DE CAMPINA GRANDE  
CENTRO DE ENGENHARIA ELÉTRICA E INFORMÁTICA  
UNIDADE ACADÊMICA DE SISTEMAS E COMPUTAÇÃO  
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO**

**JOSÉ GLAUBER BRAZ DE OLIVEIRA**

**AN EMPIRICAL STUDY OF THE RELATIONSHIP BETWEEN  
REFACTORINGS AND MERGE CONFLICTS IN JAVASCRIPT  
REPOSITORIES**

**CAMPINA GRANDE - PB**

**2024**

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

An Empirical Study of the Relationship Between  
Refactorings and Merge Conflicts in Javascript  
Repositories

José Glauber Braz de Oliveira

Dissertação 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 Mestre em Ciência da Computação.

Área de Concentração: Ciência da Computação

Linha de Pesquisa: Software Engineering

Melina Mongiovi

Sabrina Souto

Campina Grande, Paraíba, Brasil

©José Glauber Braz de Oliveira, 14/12/2023

O48e

Oliveira, José Glauber Braz de.

An empirical study of the relationship between refactorings and merge conflicts in javascript repositories / José Glauber Braz de Oliveira – Campina Grande, 2024.

86 f. : il. color.

Dissertação (Mestrado em Ciência da Computação) - Universidade Federal de Campina Grande, Centro de Engenharia Elétrica e Informática, 2023.

"Orientação: Profa. Dra. Melina Mongiovi Cunha Lima Sabino, Profa. Dra. Sabrina de Figueiredo Souto."

Referências.

1. Computer Software Program. 2. Software Engineering. 3. Refactorings. 4. Merge Conflicts. 5. Javascript. I. Sabino, Melina Mongiovi Cunha Lima. II. Souto, Sabrina de Figueiredo. III. Título.

CDU 004.4(043)



MINISTÉRIO DA EDUCAÇÃO  
**UNIVERSIDADE FEDERAL DE CAMPINA GRANDE**  
POS-GRADUACAO EM CIENCIA DA COMPUTACAO  
Rua Aprígio Veloso, 882, Edifício Telmo Silva de Araújo, Bloco CG1, - Bairro  
Universitário, Campina Grande/PB, CEP 58429-900  
Telefone: 2101-1122 - (83) 2101-1123 - (83) 2101-1124  
Site: <http://computacao.ufcg.edu.br> - E-mail: secretaria-  
copin@computacao.ufcg.edu.br / copin@copin.ufcg.edu.br

## FOLHA DE ASSINATURA PARA TESES E DISSERTAÇÕES

**JOSÉ GLAUBER BRAZ DE OLIVEIRA**

### **AN EMPIRICAL STUDY OF THE RELATIONSHIP BETWEEN REFACTORINGS AND MERGE CONFLICTS IN JAVASCRIPT REPOSITORIES**

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

Aprovada em: 14/12/2023

Profa. Dra. MELINA MONGIOVI BRITO LIRA, UFCG, Orientadora

Profa. Dra. SABRINA DE FIGUEIRÊDO SOUTO, UEPB, Orientadora

Prof. Dr. EVERTON LEANDRO GALDINO ALVES, UFCG, Examinador Interno

Prof. Dr. LEOPOLDO MOTTA TEIXEIRA, UFPE, Examinador Externo



---

Documento assinado eletronicamente por **MELINA MONGIOVI CUNHA LIMA SABINO, PROFESSOR(A) DO MAGISTERIO SUPERIOR**, em 19/12/2023, às 12:19, 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 **EVERTON LEANDRO GALDINO ALVES, PROFESSOR 3 GRAU**, em 19/12/2023, às 14:25, 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 **Leopoldo Motta Teixeira, Usuário Externo**, em 20/12/2023, às 11:36, 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 **Sabrina de Figueiredo Souto, Usuário Externo**, em 20/12/2023, às 16:44, 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 **4090330** e o código CRC **4840CCD6**.

---

**Referência:** Processo nº 23096.092249/2023-11

SEI nº 4090330

## Abstract

Maintenance activities are crucial to prolong the lifecycle of a software. An important activity during software maintenance is refactoring, which is a transformation that improves the quality of the internal structure of the code without changing its behavior. During software development, Version Control Systems (VCS) are used to integrate changes made by developers. These integration procedures, known as merge processes, may result in conflicts if changes are made in the same place in the code. This work aims to analyze the possible relationship between refactorings and merge conflicts in JavaScript code. We analyzed 76 JavaScript repositories, including 81,856 merge scenarios, which 6,356 of them have conflicts. We discovered a moderate positive correlation between the number of conflicts files/-conflicting regions and relationship/number of refactoring. For the second research question we found that the refactoring types Internal move, Move and Rename are more related to the conflicting areas, as well as a moderate correlation between the number of conflicts and the number of types of refactoring performed. 8 types of refactorings were identified at the conflicting file level and also at the conflict region level. Through statistical analysis, the relationship between the number of refactoring types and the number of conflicts was the strongest found result in our study. In addition to our automatic analysis, a manual study was conducted that analyzed 535 evolutionary commits, verifying that 447 (84%) of these were classified as floss refactoring because they had other types of modifications involved in the process. 88 evolutionary commits analyzed were classified as pure refactoring, representing 16% of evolutionary commits that only have refactoring actions.

## Resumo

Atividades de manutenção são cruciais para prolongar o ciclo de vida de um software. Uma atividade importante durante a manutenção de software é a refatoração, que é uma transformação que melhora a qualidade de um programa sem alterar seu comportamento. Durante o desenvolvimento de software, Sistemas de Controle de Versão (SCV) são utilizados para integrar as mudanças feitas pelos desenvolvedores. Esses procedimentos de integração, conhecidos como processos de mesclagem, podem resultar em conflitos se forem feitas alterações no mesmo lugar do código. Este trabalho tem por objetivo analisar a possível relação entre refatorações e conflitos de mesclagem em código JavaScript. Analisamos 76 repositórios JavaScript, incluindo 81.856 cenários de mesclagem, dos quais 6.356 apresentam conflitos. Nós descobrimos uma correlação positiva moderada entre o número de arquivos de conflitos/regiões em conflito e relação/número de refatorações. Para a segunda questão de pesquisa descobrimos que os tipos de refatoração Internal move, Move e Rename estão mais relacionados às áreas conflitantes, bem como correlação moderada entre o número de conflitos e o número de tipos de refatoração realizadas. Através de análises estatísticas, a relação entre o número de tipos de refatorações e o número de conflitos foi o mais forte encontrado em nosso estudo. 8 tipos de refatorações foram identificados ao nível dos arquivos conflitantes e a nível de região de conflito. Além da nossa análise automática para as QP1 e QP2, foi realizado um estudo manual para a QP3 que analisou 535 commits, verificando que 447 (84%) destes foram classificados como floss refactoring, possuindo outros tipos de modificações envolvidas no processo. 88 commits evolutivos analisados foram classificados como pure refactoring, representando 16% dos commits evolutivos.

## Agradecimentos

Toda essa jornada foi desafiadora. Foram muitos dias de altos e baixos, descobertas e inseguranças, memórias de aprendizado e carinho. Início agradecendo a Deus e a mim mesmo por não ter desistido desse caminho, o caminho da educação que liberta e faz crescer. Cresce dentro de mim o orgulho e a vontade de alcançar sempre mais. Foi uma experiência incrivelmente gratificante!

Não posso deixar de expressar minha profunda gratidão à minha família, a base de tudo. Desde cedo, ela me ensinou os valores da vida e a correr atrás do que é meu, sendo sem dúvida a minha maior rede de apoio. Obrigado, pai, Gil, minha Cosma, minhas irmãs Kallyse e Camila, e meus sobrinhos Duda, Aninha e Bê, por todo o amor recebido durante esse processo. Esta vitória é nossa!

Quero estender meus agradecimentos às minhas orientadoras, Professora Melina Mongiovi e Professora Sabrina Souto, por toda a ajuda, ensinamento, paciência e suporte nessa jornada. Esta vitória é nossa, e foi um prazer trabalhar junto com vocês. Vocês são excepcionais no ensino e orientação, e espero que o futuro nos reserve muitas trocas de conhecimento! Obrigado por essa rede de apoio.

Agradeço também a todos que me acompanharam nessa jornada, tanto profissional quanto pessoal, em especial Helder e Samara, obrigado por todo o amor recebido! Vocês tornam esse caminho muito mais leve. Sem o apoio e as palavras de encorajamento, essa jornada seria muito mais difícil. Quem tem um amigo tem tudo, e saibam que guardo cada um de vocês em meu coração.

Por fim, agradeço ao universo e a todos que enviaram boas energias e força para a conclusão deste ciclo. Que venham mais experiências no ensino e pesquisa, levando e gerando conhecimento por onde passar.



# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Problem . . . . .	2
1.2	Objectives . . . . .	4
1.3	Contributions . . . . .	6
1.4	Structure . . . . .	6
<b>2</b>	<b>Background</b>	<b>8</b>
2.1	Merge . . . . .	8
2.1.1	How does the GitHub merge? A three-way merge . . . . .	8
2.1.2	Merge Conflicts . . . . .	9
2.2	Refactorings . . . . .	10
2.2.1	Refactoring actions in JavaScript . . . . .	12
2.3	Refactorings versus Merge Conflicts . . . . .	33
<b>3</b>	<b>Exploring the relationship between Refactorings and Merge Conflicts</b>	<b>35</b>
3.1	Methodology . . . . .	36
3.1.1	Methodological study for RQ1: Analyzing the Relationship between Refactorings and Merge Conflict . . . . .	36
3.1.2	Methodological study for RQ2: Analyzing the Relationship between Refactorings Types and Merge Conflicts . . . . .	40
3.2	Study Setup . . . . .	41
3.2.1	Selection of JavaScript repositories . . . . .	41
3.2.2	Creation of scripts to collect text data information . . . . .	42
3.2.3	Initialization and configuration of the RefDiff 2.0 refactoring tool . . . . .	42

---

3.2.4	Execution environment . . . . .	42
3.3	Results . . . . .	43
3.3.1	Descriptive analysis . . . . .	43
3.3.2	Answering RQ1: Is there a relationship between refactoring and merge conflicts in JavaScript programs? . . . . .	52
3.3.3	Answering RQ2: What refactoring patterns relate most to merge conflicts in JavaScript programs? . . . . .	56
3.4	Discussion . . . . .	60
3.5	Threats to Validity . . . . .	62
3.5.1	Internal Validity . . . . .	62
3.5.2	External Validity . . . . .	63
3.5.3	Constructor Validity . . . . .	63
3.5.4	Conclusion Validity . . . . .	64
<b>4</b>	<b>An examination of commit evolutionary: floss or pure refactoring?</b>	<b>65</b>
4.1	Methodology . . . . .	66
4.1.1	Methodological study for RQ3: Analyzing the content of evolution- ary commit (floss and pure refactoring) . . . . .	66
4.2	Results . . . . .	68
4.2.1	Answering RQ3: The evolutionary commits that made conflicting code contain only refactorings (pure refactoring) or other modifica- tions (floss refactoring)? . . . . .	70
4.3	Discussion . . . . .	71
4.4	Threats to Validity . . . . .	72
4.4.1	Internal Validity . . . . .	73
4.4.2	External Validity . . . . .	73
4.4.3	Constructor Validity . . . . .	73
4.4.4	Conclusion Validity . . . . .	73
<b>5</b>	<b>Related Work</b>	<b>75</b>

---

<b>6</b>	<b>Conclusions</b>	<b>78</b>
6.1	Future work . . . . .	81
<b>A</b>	<b>Appendix of study</b>	<b>86</b>

# List of Figures

2.1	Common ancestor. . . . .	9
2.2	Branch. . . . .	9
2.3	Long method and Extract refactoring example . . . . .	11
2.4	Speculative Generality and Inline refactoring example . . . . .	11
2.5	Code smells and respectively refactoring examples . . . . .	12
2.6	Rename function refactoring example . . . . .	15
2.7	Another rename function refactoring example . . . . .	15
2.8	Inline function refactoring example . . . . .	17
2.9	Another inline function refactoring example . . . . .	18
2.10	Extract function refactoring example . . . . .	19
2.11	Move function refactoring example . . . . .	21
2.12	Move rename function refactoring example . . . . .	23
2.13	Extract Move function refactoring example . . . . .	24
2.14	Internal Move function refactoring example . . . . .	26
2.15	Another Internal Move function refactoring example . . . . .	27
2.16	Internal Move Rename function refactoring example . . . . .	28
2.17	Move file refactoring example . . . . .	29
2.18	Rename file refactoring example . . . . .	30
2.19	Move Rename file refactoring example . . . . .	31
2.20	Move class refactoring example . . . . .	32
2.21	Rename class refactoring example . . . . .	33
3.1	Methodology for analyzing the relationship between refactorings and merge conflicts . . . . .	37

---

3.2	Methodology for analyzing the relationship between refactoring types and merge conflicts . . . . .	40
3.3	Initial metrics of selection repositories Javascript . . . . .	45
3.4	Dispersion metrics about merge commits and merge commits with conflicts	46
3.5	Correlation Matrix with metrics selection . . . . .	47
3.6	Dispersion metrics about conflicting files and conflicting regions . . . . .	48
3.7	Dispersion metrics about relationship research variables for RQ1 . . . . .	49
3.8	Dispersion metrics about the number of refactorings in research variables for RQ1 . . . . .	50
3.9	Metric: number of conflicts . . . . .	51
3.10	Example of collected conflict . . . . .	51
3.11	Example of collected region conflict . . . . .	52
3.12	Example of collected refactoring in conflict file and region conflict . . . . .	52
3.13	Dispersion graph of variables (relationship/conflicts) of QP1 . . . . .	53
3.14	Dispersion graph of variables (n° of refactorings/conflicts) of QP1 . . . . .	54
3.15	Dispersion graph of scenarios involved in merge conflict . . . . .	55
3.16	Violinplot to types of refactoring involved in conflicting file . . . . .	57
3.17	Violinplot to types of refactoring involved in conflicting regions . . . . .	58
3.18	Dispersion graph of the relationship between the type of refactorings and conflicting variables . . . . .	59
3.19	Correlation between quantity type of refactorings and conflicting file/region	60
4.1	Methodology for manual analysis of content by evolutionary commit . . . . .	67
4.2	Example of pure evolutionary commit involved in conflict . . . . .	69
4.3	Example of floss evolutionary commit involved in conflict . . . . .	70
4.4	Dispersion of evolutionary commits floss/pure of QP3 . . . . .	71

# List of Tables

3.1	Descriptive analysis of metrics repository selection . . . . .	37
3.2	Descriptive analysis of metrics repository selection . . . . .	40
3.3	Descriptive analysis of metrics repository selection . . . . .	44
3.4	Descriptive analysis of variables refactorings and conflicts . . . . .	44
3.5	Descriptive analysis of variables of study . . . . .	55
3.6	Number of Relationship x Conflict . . . . .	56
3.7	Number of refactorings x Conflict . . . . .	56
3.8	Descriptive refactorings relationship founded in conflicting file and conflict- ing regions . . . . .	59
3.9	Number of Type of refactorings x Conflict . . . . .	60
4.1	Descriptive analysis of metrics repository selection . . . . .	67

# Chapter 1

## Introduction

Maintenance activities are essential throughout the software life cycle to prolong its usability. Meir Lehman [16] emphasizes the constant need for software adaptation. Failure to adapt leads to software being unable to meet its intended demands, resulting in a loss of quality over time. Lehman also asserts that preventive maintenance in the source code is necessary to enhance systems for future maintenance. This can involve replacing poorly structured code and implementing design patterns to improve scalability and minimize errors in the system [16].

According to William Opdyke [21], one example of maintenance activity is refactoring, which is the process of code reorganization to improve quality without altering its behavior, so if the program had some functionalities before the refactoring these may have the same result that had before the refactoring. These modifications to the source code are implemented throughout the software evolution process. To integrate changes into the product and facilitate its growth and evolution over time, it is crucial to track and document every action taken during the process. This ensures that the entire code change history is preserved at each stage of development.

Version Control Systems (VCS) play a crucial role in software evolution. In the study conducted by Santos and Murta [6], VCS are highlighted as dedicated tools for managing software development, offering various benefits such as storing development history and facilitating version recovery. They also enable developers to integrate local changes into a global environment, simplifying the code integration process.

GitHub, as mentioned by Cosentino et al. [5], is a widely adopted VCS that has experi-

enced significant growth, from 150,000 hosted projects in 2009 to 35 million hosted projects in 2015. They also highlighted that GitHub brings many resources that facilitate contribution and social integrations in the project. Achilleas Pipinellis [22] explains that GitHub's functionality revolves around branches. These branches serve as copies of the main repository and provide a space to implement sets of changes without affecting the main version of the product located in the main branch.

During the code integration process, merge operations are performed to combine changes made in different branches by individual developers into the final product [28]. However, these merge operations are not always successful, and conflicts can arise if developers attempt to integrate changes that modify the same portion of the code. Such conflicts can have a direct impact on the productivity of the development team because it is necessary extra effort to fix these problems, which sometimes may be simple, but in bigger systems might be complex [26]. To address this issue, GitHub employs mechanisms to detect and notify developers of conflicts that arise during code integration, thereby facilitating the resolution of conflicting changes.

In the given context, the study conducted by Mahmoudi et al. [17] focused on examining whether altering the code structure through refactoring actions could potentially result in merge conflicts. This is because the non-structural merge process typically considers the textual positions of the changes made. The study specifically analyzed Java programs and discovered that approximately 22% of the investigated refactorings were associated with merge conflicts.

There is a substantial amount of research dedicated to refactorings and merge processes specifically in the context of Java. The language's versatility and robustness have generated significant academic interest, leading to a substantial number of studies and research in this area [19], [11], [23], [12], [1], [14].

## 1.1 Problem

As new programming languages emerge with distinct characteristics, such as type checking, execution environments, and other factors, it becomes crucial to analyze these variables about the specific context of the emerging languages. This analysis allows for a deeper



understanding of their unique characteristics and facilitates the application of appropriate approaches in each specific language context.

At the same time, JavaScript has been gaining popularity as a well-accepted language among development teams, becoming a favored language in many development projects and being heavily utilized for web programming. It is currently among the top 10 most popular programming languages and was even declared the "Language of the Year" in 2014<sup>1</sup>. According to Johannes et al. [15], the fact that JavaScript is used both server-side and client-side has increased its popularity and highlighted the need for studies on code smells and refactorings specific to this language.

Despite JavaScript gaining significant popularity and being a widely adopted language among development teams, the research landscape in the field of JavaScript is not in the same rhythm observed in adoption development language numbers, gaining greater significance in recent years. The increasing popularity of JavaScript has led to the observation that developers often adopt poor programming practices within the language. Barros and Adachi [4] conducted a study that examined 26 different types of code smells across eight studies published between 2013 and 2020. Their findings showed that studies are needed to analyze the impact of bad design choices on systems developed in this language. The study by Silva et al. [24] addressed a specific refactoring type found in JavaScript files, known as "Internal Move."

These works further emphasize the importance of conducting comprehensive studies on software quality for other programming languages, especially from different programming paradigms. JavaScript, for example, has different challenges compared to other languages, scope-related issues like closures, a fragmented execution and development ecosystem due to the multitude of frameworks and libraries supporting the language, asynchronous management, and more. The high popularity results in the availability of numerous frameworks that contribute to the development environment in the language. Moreover, JavaScript is an interpreted, dynamic language widely used in web browsers [9], factors that contribute to a variety of programming practices in this environment. All these factors that differentiate JavaScript from other languages make both the merge processes and the continuous improvement of the system's design challenging. It highlights the need to comprehend the unique

---

<sup>1</sup><https://www.tiobe.com/tiobe-index/javascript/>

characteristics of each language and adapt refactoring and merge approaches accordingly to effectively address their specific challenges.

Considering the growing adoption of JavaScript among development teams, it is essential to investigate the correlation between refactoring and merging conflicts in this language. Such exploration can provide valuable insights to academia and developers regarding potential refactorings that may lead to conflicts during integration. Researchers can analyze whether methodologies previously applied in other studies for other programming languages can be replicated for new languages. Simultaneously, they can identify new research problems stemming from this work, addressing both merge conflicts and refactoring actions, and exploring the correlation between these variables. For developers, this study can serve as a guide for discussions on refactoring and code merging practices, considering limits on the number of refactorings to be performed in a single commit and identifying patterns of refactoring that may pose increased risks when executed together in a code integration context. By analyzing these conflict regions and to recognizing performed refactorings within them, our study contributes to verifying whether the refactoring indeed caused the merge problem, offering a starting point for further investigation into the relationship between refactorings and merge conflicts in future work.

## 1.2 Objectives

Given the complexity of our research problem, our objective is to conduct an empirical study to investigate the presence of refactorings in conflicting codes and to examine the types of refactorings involved in this process. For our study, we selected 76 JavaScript repositories from a list of repositories mentioned in the studies conducted by Silva et al. [24] and Tavares et al [26], and other random repositories found by quickly searching for JavaScript repositories. We collected information on merge conflicts and identified refactoring actions within the conflicting files and their corresponding exact local conflict, which we will call by conflict regions. The collected variables of this relationship are intended to address the research questions outlined below.

- **Is there a relationship between refactoring and merge conflicts in JavaScript programs?**

By identifying refactoring actions within conflicting files and their respective conflicting regions, it may be possible to gather evidence and initiate discussions that can substantiate the relationship between these variables. This analysis allows for a deeper understanding of how refactoring activities may contribute to or interact with merge conflicts in JavaScript programs.

- **What refactoring patterns relate most to merge conflicts in JavaScript programs?**

Given the presence of refactorings within conflicting JavaScript files and their respective conflicting regions, it is important to discuss which types of refactorings occur and how often they occur within the analyzed region. This study can suggest potentially more risky refactorings to be performed in a code integration context.

- **The evolutionary commits that made conflicting code contain only refactorings (pure refactoring) or other modifications (floss refactoring)?**

When analyzing the commits evolution, through evolutionary commits, it is possible to trace the modifications in each commit that have contributed to one or more conflict regions. The main objective is to discuss evolutionary commits that only have refactorings (pure refactoring), as opposed to those that have other changes, with refactoring actions (floss refactoring). The aim is to understand how the content of evolutionary commits effectively influenced conflicts, determining whether refactoring actions present in these commits have a significant role in merge conflicts.

This study provides valuable insights into the impact of refactorings on JavaScript code integration processes. By analyzing the potential relationship between refactorings and merge conflicts, the study contributes to a robust discussion on the factors that can influence conflict occurrences. Additionally, by identifying the types of refactorings more commonly associated with conflict regions, the study highlights the importance of careful consideration when performing refactorings. These findings prompt developers to pay closer attention to the types of refactorings they apply, leading to improved code integration practices.

## 1.3 Contributions

In summary, the main contributions of this work are:

- Empirical analysis of the relationship between refactorings and merge conflicts in JavaScript code [20];
- Identification of refactorings patterns in JavaScript that is more related with merge conflicts [20];
- Analysis of floss and pure refactorings in evolutionary commits that make region conflicts;
- Discussion and examples about JavaScript refactorings in Background Section 2.2;

In this work, we executed an empirical analysis between refactoring actions and merge conflicts. We analyzed 76 JavaScript repositories with at least one conflict in the .js file and found 81,856 merge commits with a subset of 6,356 merge commits with conflicts. We found 4,206 conflicts and 7,821 conflict regions.

Subsequently, our study found the most common types of refactorings within conflict regions, which are Internal move, Move and Rename. This study contributed to the discussion about the number of types of refactoring performed in only commit, showing results that suggest a correlation between this and the occurrence of conflicts.

A sample of evolutionary commits shows us that floss refactoring is applied in 84% of evolutionary commits analyzed and pure refactoring is applied in 16%, showing that a majority of repositories made other modifications together with refactoring actions.

## 1.4 Structure

Our study is structured as follows: Chapter 2 provides an overview of the background of this work, exploring concepts such as merge, refactorings, and their relationship. Chapter 3 introduces our first study, which explores the relationship between merge conflicts and refactoring. Chapter 4 discusses our second study, focusing on the manual analysis of evolutionary commits. Within these two chapters, we will discuss the adopted methodology,

---

execution setup, obtained results, and identified threats to validity. Chapter 5 examines related academic works that are closely aligned with our study. Finally, Chapter 6 concludes our work by showing its contributions and discussing plans for future research.

# Chapter 2

## Background

To enhance comprehension of the research field, this section will be expanded through a narrative review. The following sections discuss merge processes, refactorings, as well as studies that have already investigated the relationship between refactoring and merge conflicts. The research of the readings was made based on the knowledge already acquired about relevant articles in the area.

### 2.1 Merge

Version Control Systems (VCS) manage and merge different code versions through some algorithms and approaches, with the so-called merge processes [6]. Each new change in the code is developed in a new branch, and it is sent to this branch through the commit, which represents the action of sending the local modifications to the VCS. How the developed code is merged with the main branch is done by Git [5] through the Three-way merge algorithm, which according to Mens [18] is a code merge process that has more than two artifacts to be merged, a common ancestor, bringing more precision in the merge result.

#### 2.1.1 How does the GitHub merge? A three-way merge

This process is based on three main artifacts, as shown in Figure 2.1, base, left and right, where left and right are the parent commits of the merge commit. The base represents the main branch code at the time the secondary branches were performed, and the right and left

represent everything that was developed in the branches.

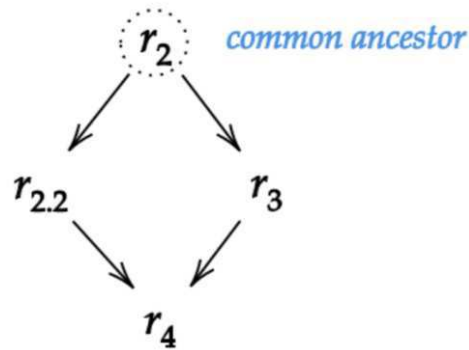


Figure 2.1: Common ancestor.

For each modified code entity there is a merge scenario which is the set (base, right and left). As we can see in Figure 2.2, branch  $r_2$  is called the base as well as the common ancestor. The left is identified as  $r_{2.2}$  and the right as  $r_3$ , resulting in the merge being  $r_4$ .

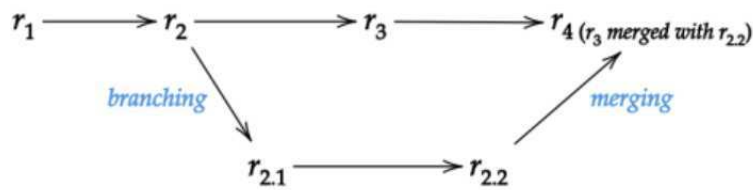


Figure 2.2: Branch.

### 2.1.2 Merge Conflicts

During the code integration process, there is a possibility that this activity may not be successful, bringing merge conflicts results and requiring extra effort to resolve these issues. As previously discussed, the most widely used Version Control System, GitHub, employs an unstructured merge approach, which identifies conflicts when changes in the same file and region lines are made by different developers. According to Mahmoudi et al. [17], current VCS may not be capable of detecting and resolving conflicts automatically, leading to the well-known merge conflicts that can be classified into six types:

- Add/Add: When both parents (left and right) of merge commit add a new file with the same name but different context;

- Content: When both parents (left and right) of merge commit applied changes in the same file at the same position;
- Modify/Delete: When P1 modify one file and P2 delete this file;
- Rename/Add: When P1 rename one file and P2 creates new file with the same name;
- Rename/Delete: When P1 rename one file and P2 delete this file;
- Rename/Rename: When both parents of merge commit rename the same file with different names.

## 2.2 Refactorings

To discuss software evolution it is necessary to show that refactorings are essential to a better growth of software. Opdyke is the first researcher to define the term "refactoring", characterizing this as an evolutionary change that will prepare the software for future changes, making it more effective and secure to do what it needs to do. He also discusses that having software with reusable design is a result of many improved actions of design, in other words, it is a continuous process that always must exist in software development [21].

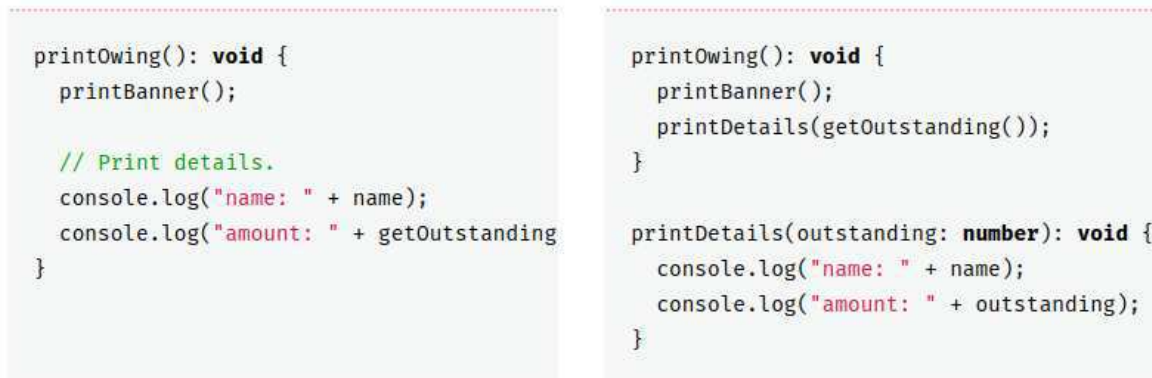
The term "refactoring" quickly became popular in the computational field. Flower made a significant contribution to popularizing the term with their study [10], which defined the refactoring process, defined best practices and specified the appropriate time and place to improve code. By analyzing a lot of code of various projects, Flower began to identify structures that "called out" to be restructured, and thus identified that the correct place to start refactoring is where the code has code smells.

In their work, Walter et al. [29] define code smells as symptoms of design problems in the code structure that can hinder software maintenance. The study provides a solid analysis of the relationship between introducing bad design choices in the system and the consequences that arise as the code grows and needs to be integrated with changes made by developers. Previous studies have already shown that conflicts during the integration process can introduce bugs into the system, but it has not yet been discussed how bad design choices can lead to possible merge conflicts. The results demonstrate that entities with certain types of code



smells are more likely to have errors. This study emphasizes the importance of considering design choices during development to reduce the likelihood of merge conflicts and improve software maintainability.

As discussed earlier, code smells are symptoms of design problems, and the article cited above provides a foundation for introducing the discussion on refactoring. In this regard, the figures below illustrate examples of code smells taken from the website "Refactoring Guru"<sup>1</sup>, found in code, and the respective refactorings performed to remove these problems.



```

printOwing(): void {
  printBanner();

  // Print details.
  console.log("name: " + name);
  console.log("amount: " + getOutstanding
}

printOwing(): void {
  printBanner();
  printDetails(getOutstanding());
}

printDetails(outstanding: number): void {
  console.log("name: " + name);
  console.log("amount: " + outstanding);
}

```

Figure 2.3: Long method and Extract refactoring example



```

class PizzaDelivery {
  // ...
  getRating(): number {
    return moreThanFiveLateDeliveries() ? 2 : 1;
  }
  moreThanFiveLateDeliveries(): boolean {
    return numberOfLateDeliveries > 5;
  }
}

class PizzaDelivery {
  // ...
  getRating(): number {
    return numberOfLateDeliveries > 5 ? 2 : 1;
  }
}

```

Figure 2.4: Speculative Generality and Inline refactoring example

In Figure 2.3 we can see an example of Long Method code smell, that defined as a method with many functionalities. By side, there is an example of refactoring the Extract Method that can be done to resolve the Long Method code smell. Another example of code smell and refactoring applied can be analyzed in Figure 2.4 which shows a Speculative Generality

<sup>1</sup><https://refactoring.guru/pt-br/>

code smell and an Inline refactoring, that simplifies your code by keeping only the essential methods, making it easier to understand.

The study by Sousa et al. [25] mined 50 projects, discussing and presenting results on types of structural refactorings, and analyzing when and for what purpose they are applied. The study presents Figure 2.5, which identifies types of code smells and refactorings that can be used to address them.

Code Smell	Common Refactorings
Complex Class	Extract Method, Move Method, Extract Class [2]
Dispersed Coupling	Extract Method
Divergent Change	Extract Class [9]
Feature Envy	Move Method, Move Field, Extract Field [9]
God Class	Extract Class, Move Method, Move Field [2]
Invasive Coupling	Move Method, Extract Method
Lazy Class	Inline Class, Collapse Hierarchy [9]
Long Method	Extract Method [9]
Shotgun Surgery	Move Method, Move Field, Inline Class [9]
Speculative Generality	Collapse Hierarchy, Inline Class, Remove Parameter, Rename Method [9]

Figure 2.5: Code smells and respectively refactoring examples

To analyze the impact of refactoring on developing systems, there are studies focused on discussing refactoring collection tools. One such example is the study by Silva et al. [24], which presents a multi-language refactoring detection tool called RefDiff 2.0. The paper presents excellent results regarding the correctness of the tool, as well as several types of refactorings for various languages, without being limited to the syntax of the language.

### 2.2.1 Refactoring actions in JavaScript

In this section, we explore some examples and discussions about the types of refactorings collected in JavaScript language.

#### REFACTORINGS RELATED TO FUNCTIONS

Languages that allow object-oriented programming are based on abstracting real-world concepts into the computational world. JavaScript is an example of language that enables this

implementation, and each object is represented by a combination of properties and methods. In JavaScript, methods are known as functions, which are created to encapsulate a set of instructions and perform a specific task within the code. The construction of a function involves specifying a name and a set of parameters that the function may or may not take.

In this section, we will discuss a set of refactorings that were identified by the RefDiff 2.0 tool [24] and performed in JavaScript code repositories, which are the same as in this study. The following refactorings are related to `RENAME`, `EXTRACT`, `INLINE`, and `MOVE FUNCTION`. In addition to detecting these four types of refactorings mentioned above, the tool also identified the composite refactoring `MOVE RENAME FUNCTION`, which involves combining the `MOVE` and `RENAME` refactorings. Furthermore, within refactorings involving functions, the tool detected a type of refactoring called `INTERNAL MOVE`.

The refactorings mentioned above, found in JavaScript code, will be discussed and analyzed in each subsection below, in comparison with Fowler's literature, where he demonstrates the motivation, instructions, and illustrative examples of refactorings he identified in his work [10].

## **RENAME FUNCTION**

### **1. Motivation**

Choosing a name for a function is a significant task, as it is very beneficial for developers to look at a function's name and identify its role within the analyzed class/file. Just like naming, identifying attributes that will be used within the scope of a function is an essential activity. Fowler [10] characterizes function attributes as the gateway to the rest of the code of this function, and through them, it becomes possible to identify the function's scope.

Renaming functions is a necessary activity for software maintenance, as functions represent actions performed within contexts, and these contexts may change during the software's lifecycle due to evolving requirements. Consequently, functions need to be modified as part of this evolution.

### **2. Step-by-step process of applying refactoring**

As Fowler explains in his work [2], a `RENAME` in JavaScript follows the same appli-

cation pattern. Fowler argues that this refactoring is generally simple, but depending on the nomenclature of the method for modification, it can be best carried out in two main ways: the simple procedure and the migration procedure.

In the simple step-by-step procedure, the following points are:

- (a) If the refactoring is just changing the name of the function, change the method declaration to the new desired name;
- (b) Find all references to the old statement and replace them with the new call;
- (c) Test.

If changing a function declaration involves removing an attribute:

- (a) Check whether the attribute to be removed is referenced in the function body, if so, evaluate the impact of removing the attribute and the code snippet;
- (b) Repeat procedures (b) and (c) of the simple procedure;
- (c) Test.

If changing the declaration of a function involves adding an attribute:

- (a) Check the impact of adding a new attribute to the function body;
- (b) Repeat procedures (b) and (c) of the simple procedure;
- (c) Test.

If the refactoring to be carried out is done in a function that is heavily referenced in the code, which makes it difficult to carry out quickly, Fowler argues that the process to be carried out is through migration, which is highlighted in the following points:

- (a) Creating a new role with a provisional name;
- (b) Perform the EXTRACT FUNCTION refactoring to remove the content for the new function from the function body;
- (c) If the new function requires the addition or removal of new parameters, use steps (b) and (c) of the simple procedures;

- (d) Apply INLINE FUNCTION refactoring to the old function;
- (e) Replace calls to the new function gradually, observing each context in which it applies;
- (f) Test;

### 3. Example of JavaScript

In the example in Figure 2.6<sup>2</sup> below, we can find a refactoring in the name of the function into config.js file. This refactoring involved changing the function's name from "getChannelDisplayName" to "getAppName."



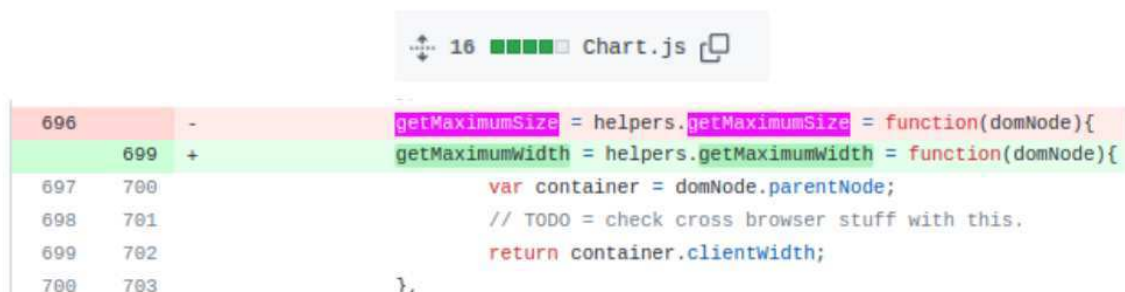
```

11 script/config.js
59 - function getChannelDisplayName (channel) {
60 -   if (channel === 'stable' || channel === 'dev') return null
61 -   return process.env.ATOM_CHANNEL_DISPLAY_NAME || channel.charAt(0).toUpperCase() + channel.slice(1)
59 + function getAppName(channel) {
60 +   return channel === 'stable'
61 +     ? 'Atom'
62 +     : `Atom ${process.env.ATOM_CHANNEL_DISPLAY_NAME || channel.charAt(0).toUpperCase() + channel.slice(1)}`

```

Figure 2.6: Rename function refactoring example

As previous example, Figure 2.7<sup>3</sup> illustrates another application of the RENAME FUNCTION refactoring in JavaScript code, changing name of function from "getMaximumSize" to "getMaximumWidth."



```

16 Chart.js
696 - getMaximumSize = helpers.getMaximumSize = function(domNode){
699 + getMaximumWidth = helpers.getMaximumWidth = function(domNode){
697 700     var container = domNode.parentNode;
698 701     // TODO = check cross browser stuff with this.
699 702     return container.clientWidth;
700 703 },

```

Figure 2.7: Another rename function refactoring example

<sup>2</sup><https://github.com/atom/atom/commit/bf9fac27cf626a2d0a46de526af6662199edc984>

<sup>3</sup><https://github.com/chartjs/Chart.js/commit/997a216b5008e33c9a9e01b5b5ac89c6536b9883>

## INLINE FUNCTION

### 1. Motivation

Identifying the context of a function and deciding what will be implemented within that context is a process that requires a programmer's careful attention. This needs to avoid implementing too many functionalities within the function's scope, which can lead to excessive indirection within the developed function. Indirection is the act of referencing something indirectly, and in the programming context, we can use delegation as an example of indirection [10]. Functions with excessive delegation to other functions can confuse the flow of data, as well as the comprehension and readability of the code.

The `INLINE` refactoring is performed when a function delegates a lot of its work to other functions, which have contexts that could be integrated into the delegating function without compromising its readability and functionality [10]. The function invokes the delegated function that can be merged with it without compromising code quality and functionality. Similar to Fowler's study, which demonstrates the applications of `INLINE` refactorings in Java code, the following sections illustrate `INLINE` refactorings in JavaScript code, highlighting that the motivation for refactoring and the steps involved are consistent across both languages.

### 2. Step-by-step process of applying refactoring

According to Fowler, it is important to follow a step-by-step guide to perform this refactoring. We can follow the same logic for JavaScript, highlighted in the following points:

- (a) Check the responsibility that the function to be removed has within the code structure. If the method is polymorphic, this type of refactoring is not appropriate, since polymorphic methods tend to have different responsibilities, which makes it difficult to carry out this type of refactoring.
- (b) Identify how the context of the function removed can be introduced into the target context.

- (c) Carry out the introduction of the new context carefully. If the source function is large, it is recommended that the code is tested when making each change;
- (d) After introducing the contexts into the new function, test and replace the calls to the removed function with calls to the target function, to correctly verify that the data flow has not been changed.

### 3. Example of JavaScript

Figure 2.8<sup>4</sup> illustrates the application of an INLINE refactoring in JavaScript code. We can observe that the constructor of the EventEmitter class made a call on line 63 to the `loadDataOverProcessBoundary` function, which existed from lines 154 to 162. The refactoring process involved integrating the content of the delegated function, `loadDataOverProcessBoundary`, into the constructor's body and eliminating the function, transferring its responsibility to the new location in the code, which is the constructor.

```

src/main-process/atom-window.js
@@ -56,11 +56,13 @@ class AtomWindow extends EventEmitter {
56 56   if (this.shouldHideTitleBar()) options.frame = false
57 57   this.browserWindow = new BrowserWindow(options)
58 58
59 -   if (this.atomApplication.projectSettings != null) {
60 -     this.projectSettings = this.atomApplication.projectSettings
61 -   }
62 +   Object.defineProperty(this.browserWindow, 'loadSettingsJSON', {
63 +     get: () => JSON.stringify(Object.assign({
64 +       userSettings: this.atomApplication.configFile.get(),
65 +       projectSettings: this.projectSettings
66 +     }, this.loadSettings))
67 +   })
68
69 62
70 63   this.loadDataOverProcessBoundary()
71 64   this.handleEvents()
72 65
73 66   this.loadSettings = Object.assign({}, settings)
74 67
75 68
@@ -151,16 +153,6 @@ class AtomWindow extends EventEmitter {
151 153   return paths.every(p => this.containsPath(p))
152 154 }
153 155
154 - loadDataOverProcessBoundary () {
155 -   Object.defineProperty(this.browserWindow, 'loadSettingsJSON', {
156 -     get: () => JSON.stringify(Object.assign({
157 -       userSettings: this.atomApplication.configFile.get(),
158 -       projectSettings: this.projectSettings
159 -     }, this.loadSettings)),
160 -     configurable: true
161 -   })
162 - }
163 -

```

Figure 2.8: Inline function refactoring example

<sup>4</sup><https://github.com/refdiff-study/atom/commit/7ce5b000e448552bb4ba9556c8f38ccfef127162>

Another example of inline refactoring can see in Figure 2.9<sup>5</sup> in which the function "getOnlyList" was removed, and its content was incorporated into the place where it was previously called, now passed as a parameter to the "babelRegisterOnly" function.



```
45 - function getOnlyList() {
46 -   return buildRegExps(__dirname, BABEL_ENABLED_PATHS);
47 - }

54 19 function setupBabel() {
55 +   babelRegisterOnly(getOnlyList());
20 +   babelRegisterOnly(babelRegisterOnly.buildRegExps(__dirname, BABEL_ENABLED_PATHS));
```

Figure 2.9: Another inline function refactoring example

## EXTRACT FUNCTION

### 1. Motivation

Extracting a part of code from an inappropriate context is a common task for almost every programmer. This happens because requirements change and evolve throughout the software's lifecycle, resulting in improvements to existing code. Functions encapsulate parts of the code, as discussed earlier. Just as it's possible to delegate too much within a function, it's also possible for a function to have too many responsibilities within a class or context, resulting in rigid and much harder-to-maintain code. According to Fowler [10], if you spend too much time figuring out what a function does, it's time to break that function into smaller parts.

The EXTRACT refactoring is performed to avoid a method having too many responsibilities and to enhance its readability. The refactorings identified by the RefDiff 2.0 tool [24] for JavaScript code follow a similar pattern to the contexts outlined in Fowler's study [10].

### 2. Step-by-step process of applying refactoring

We can follow the same logic for JavaScript, highlighted in the following points by Fowler[10]:

- (a) Create a new function with a name that clearly defines its context;

<sup>5</sup><https://github.com/refdiff-study/atom/commit/7ce5b000e448552bb4ba9556c8f38ccfef127162>



- (b) Copy the code from the source function to the destination function;
- (c) Check the code snippets in the function that need information that is in the scope of the source function and pass them as parameters;
- (d) Replace the code extracted with the call to the new function in the source function;
- (e) Test.
- (f) Look for other code snippets with similar behavior to the extracted code and check if it is possible to apply INLINE FUNCTION refactoring;

### 3. Example of JavaScript

Figure 2.10<sup>6</sup> show an example of refactoring Extract. In this example we can see a part of the code in red is removed from the source local and pasted in the target local, the function `checkRight`. We can see a call by this new function that already has a code extracted in the green local to the source local.

```

518 - var tooltipLayerStyleLeft = targetOffset.width / 2 - tooltipOffset.width / 2;
519 - // off the right side of the window
520 - if (targetOffset.left + tooltipLayerStyleLeft + tooltipOffset.width > windowSize.width)
521 -   tooltipLayer.style.left = (windowSize.width - tooltipOffset.width - targetOffset.left) + 'px';
522 - else
523 -   tooltipLayer.style.left = tooltipLayerStyleLeft + 'px';
524 - tooltipLayer.style.top = (targetOffset.height + 20) + 'px';

521 +
522 + var tooltipLayerStyleLeftRight = targetOffset.width / 2 - tooltipOffset.width / 2;
523 + if (_checkLeft(targetOffset, tooltipLayerStyleLeftRight, tooltipOffset, tooltipLayer)) {
524 +   tooltipLayer.style.right = null;
525 +   _checkRight(targetOffset, tooltipLayerStyleLeftRight, tooltipOffset, windowSize, tooltipLayer);
526 + }
527 + tooltipLayer.style.top = (targetOffset.height + 20) + 'px';

549 + function _checkRight(targetOffset, tooltipLayerStyleLeft, tooltipOffset, windowSize, tooltipLayer) {
550 +   if (targetOffset.left + tooltipLayerStyleLeft + tooltipOffset.width > windowSize.width) {
551 +     // off the right side of the window
552 +     tooltipLayer.style.left = (windowSize.width - tooltipOffset.width - targetOffset.left) + 'px';
553 +     return false;
554 +   }
555 +   tooltipLayer.style.left = tooltipLayerStyleLeft + 'px';
556 +   return true;
557 + }

```

Figure 2.10: Extract function refactoring example

<sup>6</sup><https://github.com/usablica/intro.js/commit/cd2ec800d52c69604f5e5545e125d377e1e73267>

## MOVE FUNCTION

### 1. Motivation

A good code design practice consists of promoting the modularization of software parts, that results in more reusable code (modules) that relate to each other, facilitating the division of functions within the code and better error detection. The developer needs to understand the context in which each code entity was created, its composition, e.g. attributes and methods. With software evolution, these contexts can be changed and MOVE type refactorings may be necessary to promote better code, adapting functions and even files to new contexts.

### 2. Step-by-step process of applying refactoring

We can follow the same logic for JavaScript, highlighted in the following points by Fowler [10]:

- (a) Identification of regions that use this function;
- (b) This step consists of deciding whether only this function will be subject to this type of refactoring or whether the elements that use it will also need to be moved;
- (c) Check whether the function to be moved is polymorphic, if so, it is necessary to be careful with super and sub classes when performing refactoring;
- (d) Move the function to the new context;
- (e) Carry out all necessary adaptations to the new job location. If the function has parameters, these are passed when calling the function and the name can be modified if necessary to adapt to a new context, but it would be a compound refactoring: MOVE RENAME;
- (f) Identify source contexts that reference the location of the function and that will reference the location of the newly moved function;
- (g) Test.

### 3. Example of JavaScript

The MOVE refactoring performed in the react repository in the Figure 2.11 <sup>7</sup> demonstrates the addComands function being moved from the hash\_handler.js file to the command\_manager.js file.

```
lib/ace/keyboard/hash_handler.js
47 -   this.addCommands = function(commands) {
48 -     Object.keys(commands).forEach(function(name) {
49 -       var command = commands[name];
50 -       if (typeof command === "string")
51 -         return this.bindKey(command, name);
52 -
53 -       if (typeof command === "function")
54 -         command = { exec: command };
55 -
56 -       if (!command.name)
57 -         command.name = name;
58 -
59 -       this.addCommand(command);
60 -     }, this);
61 -   };

lib/ace/commands/command_manager.js
81 +   this.addCommands = function(commands) {
82 +     commands && Object.keys(commands).forEach(function(name) {
83 +       var command = commands[name];
84 +       if (typeof command === "string")
85 +         return this.bindKey(command, name);
86 +
87 +       if (typeof command === "function")
88 +         command = { exec: command };
89 +
90 +       if (!command.name)
91 +         command.name = name;
92 +
93 +       this.addCommand(command);
94 +     }, this);
95 +   };
```

Figure 2.11: Move function refactoring example

## MOVE RENAME FUNCTION

### 1. Motivation

As previously discussed, it is common for part of the code not to have more scope into

<sup>7</sup><https://github.com/ajaxorg/ace/commit/6381f3e048506d5f0e2b8b1da81551d6ff1bd9a4>

the context initially inserted due to the constant evolution of the software. This refactoring is a combination of two previously discussed refactorings MOVE FUNCTION and RENAME FUNCTION and is applied when it is wanted to change a function location and rename the composition of its name.

## 2. Step-by-step process of applying refactoring

The application of this refactoring consists of combining the MOVE FUNCTION refactoring and the RENAME FUNCTION:

- (a) Identify the region from which the source function will be extracted. It is important to check the entire context to know whether elements that the function interacts with will also need to be moved;
- (b) Check whether the function chosen to be moved and renamed is polymorphic, if so, it is important to check all the places where the old function was called;
- (c) Perform MOVE refactoring;
- (d) Perform RENAME refactoring;
- (e) Test.

## 3. Example of JavaScript

In Figure 2.12<sup>8</sup> it is possible to see an example of refactoring Move Rename. We can see a function `saveAsUnipackage` that was moved from `tools/package.js` to `tools/unipackageclass.js` and renamed to `saveToPath`.

---

<sup>8</sup><https://github.com/meteor/meteor/commit/1bf4ffba803f95f9383f5d2ed5929726b659670c>

The image shows two code snippets from VS Code illustrating a refactor. The top snippet, from `tools/packages.js` (lines 2407-2423), shows a function `saveAsUnipackage` with a blue highlight on its name. A blue plus sign is next to line 2408. The bottom snippet, from `tools/unipackage-class.js` (lines 668-680), shows a new function `saveToPath` with a blue highlight on its name and a blue plus sign next to line 668. The code in the bottom snippet is highlighted in green, while the code in the top snippet is highlighted in red.

```

2407 - saveAsUnipackage: function (outputPath, options) {
2408 +   var self = this;
2409 -   options = options || {};
2410 -
2411 -   if (! self.pluginsBuilt || ! self.slicesBuilt)
2412 -     throw new Error("Unbuilt packages cannot be saved");
2413 -
2414 -   if (! self.version) {
2415 -     // XXX is this going to work? may need to relax it for apps?
2416 -     // that seems reasonable/useful. I guess the basic rules then
2417 -     // becomes that you can't depend on something if it doesn't have
2418 -     // a name and a version
2419 -     throw new Error("Packages without versions cannot be saved");
2420 -   }
2421 -
2422 -   var builder = new Builder({ outputPath: outputPath });
2423 -
857 - tools/unipackage-class.js
668 + saveToPath: function (outputPath, options) {
669 +   var self = this;
289 -   var handlers = self._allHandlers(packageloader);
290 -   var parts = filename.split('.');
291 -   for (var i = 0; i < parts.length; i++) {
292 -     var extension = parts.slice(i).join('.');
293 -     if (_.has(handlers, extension))
294 -       return handlers[extension];
670 +   options = options || {};
671 +
672 +   if (! self.version) {
673 +     // XXX is this going to work? may need to relax it for apps?
674 +     // that seems reasonable/useful. I guess the basic rules then
675 +     // becomes that you can't depend on something if it doesn't have
676 +     // a name and a version
677 +     throw new Error("Packages without versions cannot be saved");
678 +   }
679 +
680 +   var builder = new Builder({ outputPath: outputPath });

```

Figure 2.12: Move rename function refactoring example

## EXTRACT MOVE FUNCTION

### 1. Motivation

The motivation for this refactoring is extracting a portion of code that no longer belongs to the context of a previously defined function and moving it to a new context. This process also involves changing the function's name to better align with the evolved code in new contexts.

### 2. Step-by-step process of applying refactoring

The application of this refactoring consists of combining the EXTRACT FUNCTION

refactoring and the MOVE FUNCTION:

- (a) Identify the region from which the source function will be extracted. It is important to check the entire context to know whether elements that the function interacts with will also need to be moved;
- (b) Check whether the function chosen to be extracted and moved is polymorphic, if so, it is important to check all the places where the old function was called;
- (c) Perform EXTRACT refactoring;
- (d) Perform MOVE refactoring;
- (e) Test.

### 3. Example of JavaScript

In Figure 2.13<sup>9</sup> it is possible to see an example of refactoring Extract Move Rename. We can see part of the code function location going to function toKeyValue that its new name in different files.

The image shows two code snippets from a JavaScript file. The top snippet, from `src/services.js`, shows a function that processes location parameters. Lines 21-25 are marked with a red background, indicating the source code to be extracted. Lines 23-26 are marked with a green background, indicating the code to be moved to a new location. The bottom snippet, from `src/Angular.js`, shows the `parseKeyValue` function (lines 398-400) and the newly created `toKeyValue` function (lines 401-408). The `toKeyValue` function is a copy of the code from the red region in the top snippet, now located in a different file.

```

42 src/services.js
21 - var params = [];
22 - foreach(location.param, function(value, key){
23 -   params.push(encodeURIComponent(key) + '=' + encodeURIComponent(value));
24 - });
25 - return (location.path ? location.path : '') + (params.length ? '?' + params.join('&') : '');
23 + var hashKeyValue = toKeyValue(location.hashSearch);
24 + return location.href +
25 + (location.hashPath ? location.hashPath : '') +
26 + (hashKeyValue ? '?' + hashKeyValue : '');

src/Angular.js
@@ -398,6 +398,14 @@ function parseKeyValue(keyValue) {
398 398   return obj;
399 399 }
400 400
401 + function toKeyValue(obj) {
402 +   var parts = [];
403 +   foreach(obj, function(value, key){
404 +     parts.push(encodeURIComponent(key) + '=' + encodeURIComponent(value));
405 +   });
406 +   return parts.length ? parts.join('&') : '';
407 + };
408 +

```

Figure 2.13: Extract Move function refactoring example

<sup>9</sup><https://github.com/angular/angular.js/commit/d717020911a350a5ea3c0a985c57d56c8fcad607>

During data extraction, a refactoring caught in JavaScript codes was observed, which is captured by the tool but there is no discussion in the study by Silva et al. [24]. This refactoring is called by tool RefDiff 2.0 as INTERNAL MOVE and its results will be discussed in the next subsection.

## INTERNAL MOVE FUNCTION

### 1. Motivation

This refactoring consists of removing a specific piece of code that is in a code scope and inserting it into a more internal/external scope. According to the analysis of the study by Silva et al. [24] and the analysis of those researched in the study, we verified that this refactoring is due to the need to adapt only a part of the code to a new scope, within the same file. This type of refactoring happens a lot in nested functions, being more common in languages that allow this type of code scope.

### 2. Step-by-step process of applying refactoring

The application of this refactoring consists of the following steps:

- (a) Identify the region from which the innermost code snippet will be extracted. It is important to check the entire context to know whether elements that the function interacts with will also need to be moved;
- (b) Move the code snippet to the new scope in the same file;
- (c) Test.

### 3. Example of JavaScript

In the example in Figure 2.14<sup>10</sup> we can see an example of Internal Move refactoring. In the specific context, the function `onreadystatechange` was in the scope inside the `if` context, after refactoring the function was moved to another scope the `else if`.

We can see another example below in Figure 2.15<sup>11</sup> where the function `defaultNegativeCompare` was in the scope `Expectation.prototype.wrapCompare` but after refactoring this

---

<sup>10</sup><https://github.com/requirejs/requirejs/commit/5463c8f5940c05427289afa106f5748b35542aee>

<sup>11</sup><https://github.com/jasmine/jasmine/commit/533bda5d2400755a1ef49bfd59712af1f620496e>

```

213 -     if (text.createXhr()) {
214 -         text.get = function (url, callback) {
215 -             var xhr = text.createXhr();
216 -             xhr.open('GET', url, true);
217 -             xhr.onreadystatechange = function (evt) {
218 -                 //Do not explicitly handle errors, those should be
219 -                 //visible via console output in the browser.
220 -                 if (xhr.readyState === 4) {
221 -                     callback(xhr.responseText);
222 -                 }
223 -             };
224 -             xhr.send(null);
225 -         };
226 -     } else if (typeof process !== "undefined" &&
213 +     if (typeof process !== "undefined" &&
214 +         process.versions &&
215 +         !!process.versions.node) {
216 +             //Using special require.nodeRequire, something added by r.js.
@@ -237,6 +224,19 @@
237 224         }
238 225         callback(file);
239 226     };
227 +     } else if (text.createXhr()) {
228 +         text.get = function (url, callback) {
229 +             var xhr = text.createXhr();
230 +             xhr.open('GET', url, true);
231 +             xhr.onreadystatechange = function (evt) {
232 +                 //Do not explicitly handle errors, those should be
233 +                 //visible via console output in the browser.
234 +                 if (xhr.readyState === 4) {
235 +                     callback(xhr.responseText);
236 +                 }
237 +             };
238 +             xhr.send(null);
239 +         };
240 240     } else if (typeof Packages !== 'undefined') {
241 241         //why Java, why is this so awkward?
242 242         text.get = function (url, callback) {

```

Figure 2.14: Internal Move function refactoring example

function was in the `Expectation.prototype.instantiateMatcher`, these modification was performed in the same file.

## INTERNAL MOVE RENAME FUNCTION

### 1. Motivation

This refactoring consists of a match of Internal Move refactoring and Rename refactoring. It begins with the desire to change a function within a specific scope and also change its name.

### 2. Step-by-step process of applying refactoring

The application of this refactoring consists of the following steps:



```

2598 - Expectation.prototype.wrapCompare = function(name, matcherFactory) {
2598 + function wrapCompare(name, matcherFactory) {
2599 2599     return function() {
2600 2600         var args = Array.prototype.slice.call(arguments, 0),
2601 -         expected = args.slice(0),
2602 -         message = '';
2601 +         expected = args.slice(0);
2603 2602
2604 2603         args.unshift(this.actual);
2605 2604
2606 -         var matcher = matcherFactory(this.util, this.customEqualityTesters),
2607 -         matcherCompare = matcher.compare;
2605 +         var matcherCompare = this.instantiateMatcher(matcherFactory);
2606 +         var result = matcherCompare.apply(null, args);
2607 +         this.processResult(result, name, expected, args);
2608 +     };
2609 + }
2608 2610
2609 - function defaultNegativeCompare() {
2610 -     var result = matcher.compare.apply(null, args);
2611 -     result.pass = !result.pass;
2612 -     return result;
2613 - }
2611 + Expectation.prototype.instantiateMatcher = function(matcherFactory) {
2612 +     var matcher = matcherFactory(this.util, this.customEqualityTesters);
2614 2613
2615 -     if (this.isNot) {
2616 -         matcherCompare = matcher.negativeCompare || defaultNegativeCompare;
2617 -     }
2614 +     function defaultNegativeCompare() {
2615 +         var result = matcher.compare.apply(null, arguments);
2616 +         result.pass = !result.pass;
2617 +         return result;
2618 +     }

```

Figure 2.15: Another Internal Move function refactoring example

- (a) Implementation of refactoring Internal Move;
  - (b) Implementation of refactoring Rename;
  - (c) Test.
- ### 3. Example of JavaScript

As depicted in Figure 2.16<sup>12</sup>, we observe the change of scope for the Call function, and its name is now MixinCall.

## REFACTORINGS RELATED TO FILES

A .js file is a text file that contains a set of lines that will JavaScript code. The entire file will fit into a context within the code and can be part of different layers of the software, such as models, controllers, etc. Given their location in the code, these files can contain sets of functions, attributes, classes, interfaces, among others. Moving a file represents moving the entire set that made this file. Just like moving a file, renaming is to adapt this file to

<sup>12</sup><https://github.com/less/less.js/commit/16746e9b1eca8e5cbf0b2fb9f8e412a5ad26e95a>

```

  235  lib/less/tree/mixin-call.js
  ...  ...  @@ -1,153 +1,154 @@
  1  - module.exports = function (tree) {
      1  + var Node = require("./node.js"),
      2  +   Selector = require("./selector.js"),
      3  +   MixinDefinition = require("./mixin-definition.js"),
      4  +   defaultFunc = require("../functions/default.js");
  2  5
  3  - var Call = function (elements, args, index, currentFileInfo, important) {
      4  -   this.selector = new(tree.Selector)(elements);
  6  + var MixinCall = function (elements, args, index, currentFileInfo, important) {
      7  +   this.selector = new(Selector)(elements);
  5  8   this.arguments = (args && args.length) ? args : null;
  6  9   this.index = index;
  7  10  this.currentFileInfo = currentFileInfo;
  8  11  this.important = important;
  9  12  };

```

Figure 2.16: Internal Move Rename function refactoring example

a new context or even to the context itself in which the name initially given is no longer representative. The next subsections will discuss MOVE and RENAME refactorings to files and MOVE RENAME composite refactoring. These types of refactorings will be discussed in our study based on the classifications discussed in the Fowler study [10].

## MOVE FILE

### 1. Motivation

When deciding to move a file, the programmer intends to add information to a new context, so the programmer needs to identify if this entire file set will be necessary and is coherent to be in the new context.

### 2. Step-by-step process of applying refactoring

We can follow the same logic for JavaScript, highlighted in the following points by Fowler [10]:

- (a) Identification of the regions that use this file;
- (b) Move the file to the new context;
- (c) Performs all necessary adaptations to the new location of the files. This step consists of checking all imports that were directed to the old file location and adapting them to the new file location;

(d) Test.

### 3. Example of JavaScript

In the example in Figure 2.17<sup>13</sup>, we can see an example of the MOVE refactoring file that the tool detected by moving the entire file from the `src/moveToAngularCom/Model.js` directory to the destination directory `src/delete/Model.js`.



Figure 2.17: Move file refactoring example

## RENAME FILE

### 1. Motivation

When renaming a file, the programmer wants to adapt this file that belonged to a context that changed its intention, so it needs to identify if the new file name will be representative of the context.

### 2. Step-by-step process of applying refactoring

We can follow the same logic for JavaScript, highlighted in the following points by Fowler [10]:

- (a) Identification of the regions that use this file;
- (b) Renaming the file name;
- (c) This step consists of checking all imports that were directed to the old file location and adapting them to the new file location;
- (d) Test;

<sup>13</sup><https://github.com/angular/angular.js/commit/11a6431f8926c557f3c58408dacc98466e76cde1>

### 3. Example of JavaScript

In the example in Figure 2.18<sup>14</sup> below we can see an example of file renaming, changing the name `createError.spec.js` to `AxiosError.spec.js` in Javascript code from the Axios repository.

```
test/specs/core/createError.spec.js
1 - var createError = require('../../lib/core/createError');
2 -
3 - describe('core::createError', function() {
4 -   it('should create an Error with message, config, code, request, response and isAxiosError', function() {
5 -     var request = { path: '/foo' };
6 -     var response = { status: 200, data: { foo: 'bar' } };

test/specs/core/AxiosError.spec.js
1 + var AxiosError = require('../../lib/core/AxiosError');
2 +
3 + describe('core::AxiosError', function() {
4 +   it('should create an Error with message, config, code, request, response and isAxiosError', function() {
5 +     var request = { path: '/foo' };
6 +     var response = { status: 200, data: { foo: 'bar' } };
7 +     var error = new AxiosError('Boom!', 'ESOMETHING', { foo: 'bar' }, request, response);
```

Figure 2.18: Rename file refactoring example

## MOVE RENAME FILE

### 1. Motivation

This refactoring consists of executing two refactorings together, MOVE FILE and RENAME FILE. When changing a context file, there may be a need to also change its name, resulting in this composite refactoring.

### 2. Step-by-step process of applying refactoring

We can follow the same logic for JavaScript, highlighted in the following points by Fowler [10]:

- (a) Application of the MOVE FILE refactoring steps;
- (b) Application of the RENAME FILE refactoring steps;
- (c) Test;

<sup>14</sup><https://github.com/axios/axios/commit/7f1236652adb813ff884be008fe73ddf0590c664>

### 3. Example of JavaScript

In the example in Figure 2.19<sup>15</sup> we can see the application of a compound refactoring called MOVE RENAME on a file. You can see that the file initially called ReactDOM-FrameScheduling.js now has a new name ReactScheduler.js and a new directory.

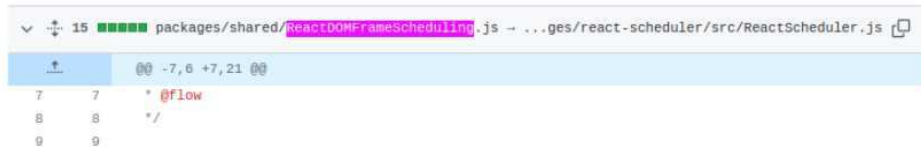


Figure 2.19: Move Rename file refactoring example

## REFACTORINGS RELATED TO CLASSES

The next refactorings that were identified by the tool are related to the JavaScript versions that the software was developed. Classes before the ES6 (JavaScript 5) version could be abstracted and represented by creating functions, and these represented the same functionality. Starting with ES6, it was possible to create a *class* instance to better represent an object in JavaScript. The motivation for refactoring MOVE and RENAME CLASS is the adaptation of classes to a context that has evolved until the name initially chosen is no longer so representative.

### MOVE CLASS

#### 1. Motivation

Adaptation of an already developed class to a new context, to promote better code modularity.

#### 2. Step-by-step process of applying refactoring

- (a) Identification of regions that use this class;
- (b) Transporting the class to the new context;
- (c) Carry out all necessary adaptations to the new classroom location.

<sup>15</sup><https://github.com/refdiff-study/react/commit/999b656ed1c94b00fcfd043f54e18ade7553dee0>

- (d) Identify source contexts that reference the location of the old class and that will reference the new location of the moved class;
- (e) Test.

### 3. Example of JavaScript

The figure 2.20<sup>16</sup> represents an example of refactoring move class, that moves `NaturalModuleIdsPlugin` from `lib` to `lib/ids` source.

```

43 lib/NaturalModuleIdsPlugin.js
26 - class NaturalModuleIdsPlugin {
27 -   /**
28 -    * @param {Compiler} compiler the compiler instance
29 -    * @returns {void}
30 -    */
31 -   apply(compiler) {
32 -     compiler.hooks.compilation.tap("NaturalModuleIdsPlugin", compilation => {
33 -       compilation.hooks.optimizeModuleOrder.tap(
34 -         "NaturalModuleIdsPlugin",
35 -         modules => {
36 -           modules.sort(byIndexOrIdentifier);
37 -         }
38 -       );
39 -     });
40 -   }
41 - }

31 lib/ids/NaturalModuleIdsPlugin.js
14 + class NaturalModuleIdsPlugin {
15 +   /**
16 +    * @param {Compiler} compiler the compiler instance
17 +    * @returns {void}
18 +    */
19 +   apply(compiler) {
20 +     compiler.hooks.compilation.tap("NaturalModuleIdsPlugin", compilation => {
21 +       compilation.hooks.moduleIds.tap("NaturalModuleIdsPlugin", modules => {
22 +         const modulesInNaturalOrder = Array.from(modules).sort(
23 +           compareModulesByIndexOrIdentifier(compilation.moduleGraph)
24 +         );
25 +         assignAscendingIds(modulesInNaturalOrder, compilation);
26 +       });
27 +     });
28 +   }
29 + }

```

Figure 2.20: Move class refactoring example

## RENAME CLASS

### 1. Motivation

Adaptation of the name of the developed code class to an evolving context, to promote a better understanding of the code.

<sup>16</sup><https://github.com/refdiff-study/react/commit/999b656ed1c94b00fcfd043f54e18ade7553dee0>

## 2. Step-by-step process of applying refactoring

The application of this refactoring follows these next steps:

- (a) Identification of regions that use this class;
- (b) Renaming the class name;
- (c) This step consists of checking all imports that were directed to the old class and adapting them to the new name class;
- (d) Test.

## 3. Example of JavaScript

The figure 2.21<sup>17</sup> represents the application of refactoring RENAME classes in JavaScript code, renaming the class that was previously called `TreeSitterHighlightIterator` to `LanguageLayer`.



```
src/tree-sitter-language-mode.js
559 - class TreeSitterHighlightIterator {
560 -   constructor (languageMode, treeCursor) {
386 + class LanguageLayer {
387 +   constructor (languageMode, grammar) {
561 388     this.languageMode = languageMode
389 +     this.grammar = grammar
390 +     this.tree = null
391 +     this.currentParsePromise = null
392 +     this.patchSinceCurrentParseStarted = null
393 +   }
394 +
```

Figure 2.21: Rename class refactoring example

## 2.3 Refactorings versus Merge Conflicts

The study [17] was the first to analyze the relationship between refactorings and merge conflicts. This work aimed to investigate the extent to which these two variables are related, discussing whether conflicts involving refactorings are more difficult to solve and which types of refactorings are more prone to errors. The methodology of the study selected Java code repositories and, for each repository, identified the conflict regions and the previous modifications that led to the conflict, known as evolutionary commits. After identifying the

<sup>17</sup><https://github.com/atom/atom/commit/e60f0f9b6084e220b2b54cf4218fdf31f9733bd9>

evolutionary commits, the study focused on searching for refactorings in those commits and relating them to the conflict regions to determine whether there is a relationship between the research variables, refactorings, and merge conflicts. The study found that about 22% of refactoring actions were involved in merge conflicts and also obtained results on which types of refactorings are more related to conflicts.

Similar to Mahmoudi et al. study [17], Oliveira et al. [20] was the first to analyze the presence of refactoring actions in conflicting code for JavaScript. The study shows that approximately 7% of the analyzed scenarios involved refactoring actions in conflicting files, with 4% of them exhibiting refactoring at the conflict region level. Moreover, a moderate and positive correlation was found between the quantity of refactoring types and the number of conflicts, suggesting a potential insight into the limit of refactorings to be performed in a single commit. Move and Internal Move refactorings were the most commonly associated with conflicting files and conflict regions, explaining a discussion about Internal Move, a refactoring type related to scope and more connected to languages with specific structural features, such as the ability to develop nested functions. This study serves as an initial exploration of the relationship between refactorings and merge conflicts, prompting discussions on the need for advancements in methodologies applied to other languages and the development of better tools for JavaScript. This study stands out from previous research by addressing this relationship in a popular language that had not been extensively explored, showing types of refactorings most associated with merge conflicts. Additionally, it contributes to a data set containing floss and pure refactoring in JavaScript code commits.



# Chapter 3

## Exploring the relationship between Refactorings and Merge Conflicts

To analyze the relationship between refactoring actions and merge conflicts in JavaScript code, we performed a study to analyze the presence of refactorings in merge scenarios that involved conflicts. This verification was performed at the file and conflict region levels. This chapter aims to discuss the entire methodological process adopted, results, and implications for the first and second research questions.

Initially, we discuss the methodology of our study to address the first two research questions. Two studies were performed, which are presented, with the input and output variables, in Section 3.1. Subsequently, in Section 3.2, we discuss the preparation of our environment and what was developed to collect the variables for our research. With the collection of this data, Section 3.3 covers everything from the descriptive analysis of the data to examples of the variables, culminating in the section dedicated to addressing the research questions.

Presenting the data, Section 3.4 explores the implications of the values found through statistical analyses, pointing out the points of contribution from our study. To conclude, we discuss the threats to the validity of our methodological process in Section 3.5.

The studies conducted in this chapter aim to answer the following Research Questions:

- **RQ1:** Is there a relationship between refactoring and merge conflicts in JavaScript programs?
- **RQ2:** What refactoring patterns relate most to merge conflicts in JavaScript programs?

The methodology for the first research question involves obtaining the evolutionary commits that contributed to the conflict regions and extracting any refactorings present in these commits, if they exist. This allows us to examine the history of changes leading up to the conflicts and analyze the role of refactorings in their occurrence. The methodology for the second research question aims to identify patterns in the relationships between refactorings and merge conflicts. This involves analyzing the data gathered from the first research question to uncover any recurring patterns or trends. This analysis helps us gain a deeper understanding of the potential relationship between specific types of refactorings and the likelihood of conflicts during code integration.

Next, each stage of the methodology for collecting and analyzing evolutionary commits is detailed.

## 3.1 Methodology

In this section, we present the methodology for our first two research questions. In addition to the overall figure illustrating each step, we provide tables detailing the inputs and outputs of each stage.

### 3.1.1 Methodological study for RQ1: Analyzing the Relationship between Refactorings and Merge Conflict

We used the methodology based on the study of Mahmoudi et al. [17] to collect the presence of refactorings in conflicting files and their respective conflict regions. It is based on the analysis of evolutionary commits. These commits represent the evolution of code present in the merge commits parents. Extracting evolutionary commits was executed through the terminal interface itself, using Git commands, while the part of collecting refactorings was extracted using the RefDiff 2.0 tool from the study by Silva et al. [24]. The choice of this tool was made because, up to the data collection moment, it was the only one that collected refactoring actions performed in JavaScript code. To better illustrate the metrics that were being used as inputs and outputs in each activity of the QP1 methodology, Figure 3.1 and Table 3.1 have been developed, with a description of each step provided subsequently.

Table 3.1: Descriptive analysis of metrics repository selection

Activity	Description	Input	Output
1	Mining JavaScript code repositories	Mining metrics	Repository of Javascript code
2	Mining merge commit information	Repository of Javascript code	Conflict commit hash
3	Detecting conflicting regions	Parents of conflicting commits	Conflicting regions of each merge scenario
4	Extracting evolutionary commits that built conflicts	Conflicting regions of each merge scenario	Change regions of evolutionary commits
5	Collecting refactoring actions in evolutionary commits	Hash of commits extracted by evolutionary commits	Refactorings done in this commit
6	Detecting the relationship between refactoring actions and conflict regions	Local of conflict code and local of refactoring change	Validation if there is overlapping between refactoring and region change

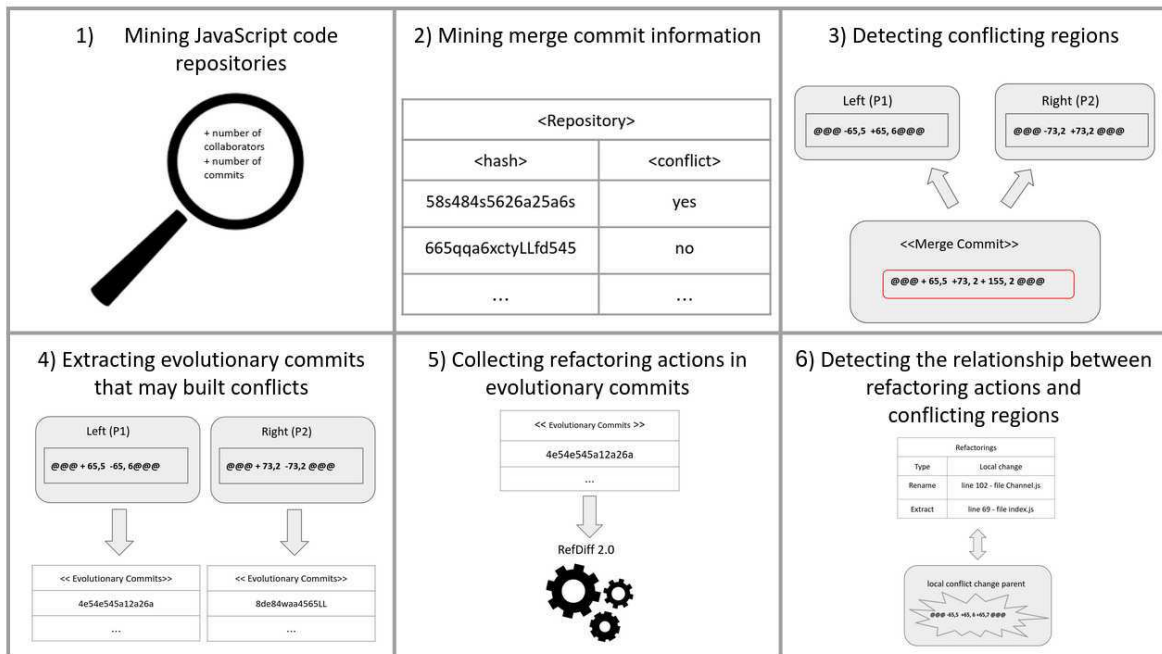


Figure 3.1: Methodology for analyzing the relationship between refactorings and merge conflicts

### **1) Mining JavaScript code repositories**

The initial stage of this work consisted of selecting the repositories that would be used as the subject of the study (first step of Figure 3.1). To do this, we selected related works that had already analyzed JavaScript projects and their characteristics, such as the studies by [24] and [26]. Next, we used two main metrics to select the most representative repositories for our study: the number of developers involved and the total number of commits. Based on these metrics, we selected a sample of 50 repositories and subsequently extracted the merge commits related to the project under analysis.

### **2) Mining merge commit information**

After the process of extracting merge commits from the repository, this stage aimed to refine the sample by selecting only merge scenarios that have at least one conflicting file (second step of Figure 3.1). We developed a script to take the total merge scenarios as input and return only the conflicting scenarios. The main idea of this stage is to perform the merge between P1 and P2, which represents the left and right commits, i.e., the parent commits of the merge commit. Using the commands "git checkout P1", "git merge P2", and "git diff -U0", the interface returns the entire result of the merge process, highlighting which files have at least one conflict region. The "git diff -U0" command returns the difference between the merged commits without adding blank lines.

### **3) Detecting conflicting regions**

Identifying conflicting regions within conflicting files after the merge process is the goal of this stage. The third step in Figure 3.1 demonstrates the process of identifying these regions of conflict. The command "git diff -U0" is used to return these regions. The merge commit (MC) is identified with the symbol "@@...@@". This symbol consists of three pairs, where the first two represent the location of the conflicting code in the respective parent commits, P1 and P2. By identifying the conflict regions in the parent commits of the merge commit, it is possible to determine the evolutionary commits that contributed to the construction of each of these regions. This will be detailed in the next stage of the methodology.

#### **4) Extracting evolutionary commits that built conflicts**

During this stage of the methodology (fourth step of Figure 3.1), the focus was on collecting information about the commits that contributed to the construction of the code in conflict regions, i.e., the commits that introduced changes in the parent commits of each merge commit. Using the identified change regions within each parent commit of the merge commit, we executed the commands "git log -L start(P1), end(P1): file P2..P1" and "git log -L start(P2), end(P2): file P1..P2". These commands take as input parameters the start and end of the change region in the respective parent, as well as the conflicting file path. The output of this command includes all the commits that contributed to the construction of the conflict region. The result of executing this command would be all the commits that are between P1 and its common ancestor and that were part of the evolution of the parent commit. With these commits that contributed to the parent commits, it is possible to run the refactoring tool for each commit and identify if any refactoring was performed within it. Additionally, during this stage of the methodology, we collected the data using scripts that we developed. This data is essential for the study as it represents the number of conflicts and conflict regions of .js files being analyzed. Data such as the total number of conflicts, conflict regions, and analyzed .js files are collected in this stage to be used for the correlation of the research variables.

#### **5) Collecting refactoring actions in evolutionary commits**

When evaluating the history of source code, identifying evolutionary commits, as seen in the previous step, is an essential task. In the fifth step of Figure 3.1, we use RefDiff 2.0 tool [24] to identify refactorings within these commits, as well as indicate in which region of the file the change was made and what refactoring type was applied. By providing accurate information about refactorings performed on the source code, RefDiff 2.0 facilitates the discussion of design problems in the code, as the application of refactoring begins with a bad smell in the code. Therefore, it is possible to better understand how the code has evolved through the action of refactorings.

### 6) Detecting the relationship between refactoring actions and conflict regions

Finally, when collecting refactorings within evolutionary commits, it is necessary to verify the presence of refactorings within the conflicting file and the conflict regions collected in step 3, to identify whether they are related by overlapping lines (sixth step of Figure 3.1). At the end of this step, all relationships between refactorings and merge conflicts at the level of the conflicting file and the level of the conflicting region are collected.

### 3.1.2 Methodological study for RQ2: Analyzing the Relationship between Refactorings Types and Merge Conflicts

In this section, we describe the methodology used to analyze the relationship between refactoring types and merge conflicts demonstrated in Figure 3.2. Table 3.2 illustrates the metrics that were used as inputs and outputs.

Table 3.2: Descriptive analysis of metrics repository selection

Activity	Description	Input	Output
1	Summary of refactorings types in Merge commits with conflicts	Data with refactorings types founded in conflicts	Summary of refactorings types founded
2	Statistical analysis of types of refactorings in the conflicting files and conflict regions	Variables: Refactoring types and conflicting regions	Statistical correlation tests

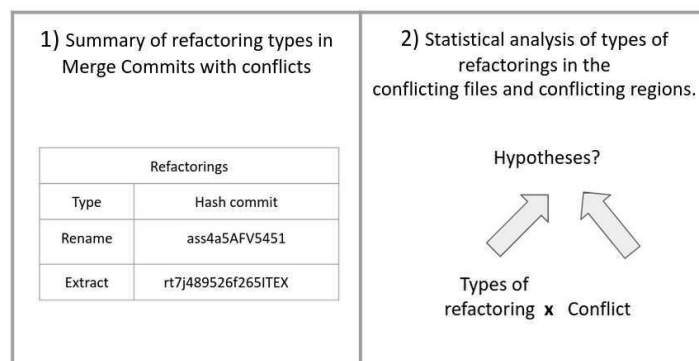


Figure 3.2: Methodology for analyzing the relationship between refactoring types and merge conflicts

### **1) Summary of refactorings types in Merge Commits with conflicts**

The objective of this stage is to collect information about the most common types of refactoring that are involved in conflicting files and conflict regions. Identifying the most common types of refactoring that are most often involved in merge conflicts can be very useful in ensuring the integrity of the code in a repository. By collecting this information, it is possible to better understand which refactorings can lead to conflicts, allowing tag preventive measures to be taken to avoid them.

### **2) Statistical analysis of types of refactorings in the conflicting files and conflict regions.**

To obtain quantitative analyses, statistical tests of correlation will also be performed in this stage between the study variables, which are: the occurrence of refactoring types and merge conflicts, to identify possible relationships between them at a statistical level.

## **3.2 Study Setup**

This section describes the preparation of our environment for conducting the study for RQ1 and RQ2, including the selection of initial metrics and the information on the local where the experiment was conducted.

### **3.2.1 Selection of JavaScript repositories**

We selected the most repositories from a list of repositories analyzed in previous studies, such as [24] and [26]. These studies focused on the analysis of refactorings and conflict analysis, respectively, and had evaluation datasets with lists of repositories analyzed. To select also new repositories, we conducted a search on GitHub for JavaScript repositories, choosing them based on the initial metrics we identified. To select the repositories for our study, we used metrics that emphasize the importance of selecting good inputs for empirical studies. Specifically, we used the number of contributors and the number of commits as our selection criteria. We believe repositories with high values for these metrics are more likely to have merge conflicts.

### 3.2.2 Creation of scripts to collect text data information

To gather the necessary data for the study, we developed five Python scripts that collected information on merge scenarios and important evolutionary commits, such as the location of conflict regions. These scripts were developed using version 3.0 and included five main functions:

- Collection of conflict scenarios ;
- Collection of information on conflicting commits;
- Collection of the relationship between refactorings and conflicting files and conflict regions;
- Collection of conflict information by merge scenario;
- Collection of information on types of refactorings.

A manual inspection was performed through manual tests to validate the obtained data. All the material developed and extracted from the scripts is available in the repository <sup>1</sup>, complete with step-by-step instructions for their execution.

### 3.2.3 Initialization and configuration of the RefDiff 2.0 refactoring tool

To collect information related to refactorings, we used the Ref Diff 2.0 tool from the study [24]. This tool is multilingual, meaning it can detect these actions for many programming languages. In our study, we used the plugin developed by the study team for JavaScript codes. The tool was executed in Eclipse and configured through Maven artifacts.

### 3.2.4 Execution environment

To perform data analysis, we used a computer with an i5 processor with 4 cores at 2.30GHz and 12 gigabytes of memory. The average internet connection was 130 Mbps.

---

<sup>1</sup>[https://github.com/joseglauberbo/data\\_mestrado\\_dissertacao](https://github.com/joseglauberbo/data_mestrado_dissertacao)



## 3.3 Results

In this chapter, we explore the methodological aspects addressed in our study for RQ1 and RQ2. In Section 3.3.1, we present a descriptive analysis of the data, emphasizing measures of central tendency. The purpose is to provide a more detailed description of the variables in our study. Our goal is to present, from the outset, the selection of initial metrics to the key variables, such as refactorings, merge conflicts, and their relationship. We further explore the discussion of the chosen variables, interpreting the values found in the previous section and providing concrete examples of our data found. In Sections 3.3.2 and 3.3.3 we answered our research questions based on the findings obtained during the study. In Section 3.4, we discuss our findings, assess the hypotheses, and draw conclusions. Concluding the discussion, in Section 3.5, we present threats to the validation of our study, based on the methodology applied in the research questions.

Initially, we selected 100 JavaScript code repositories, from which 76 have at least one merge conflict in the Javascript file (files with `.js`). These 76 repositories were made by 31,329 contributors and all of these have 547,421 commits. A total of 81,856 merge scenarios were examined. Among them, 6,356 were found to have at least one merge conflict. Only conflicts occurring in `.js` files were considered, while conflicts in configuration files (such as build files and readme files) and test files were disregarded. As a result, a total of 4,206 valid merge conflicts were included in the analysis. Within these valid merge conflicts, a total of 7,821 conflict regions were identified. From the evolutionary commits, the RefDiff tool collected 2,961 refactorings applied within files involved in conflicts. Also, out of these refactorings, 1,236 were specifically applied within the conflict regions themselves. This tool captured various types of refactorings, including Rename, Move, Extract, Inline, Internal Move, Move Rename and Extract Move.

### 3.3.1 Descriptive analysis

Table 3.3 and Table 3.4 show some measures of central tendency and dispersion of the data, such as standard deviation, the minimum and maximum value of the set, and quartiles by the variables of our study.

The variables analyzed in the study encompass repository selection metrics as well as

Table 3.3: Descriptive analysis of metrics repository selection

Variable	Mean	SD	Min	Max	Q1(25%)	Q2(50%)	Q3(75%)
N° of contributors	412.22	702.78	12	4704	104.25	214.5	372
N° of commits	7202.9	10313.9	392	41503	1883.75	3024.5	7459.25
N° of merge scenarios	1077.05	1817.13	35	9181	201.75	413	738.5
N° of merge scenarios with conflict	83.63	140.73	3	710	14.75	25.5	71

Table 3.4: Descriptive analysis of variables refactorings and conflicts

Variable	Mean	SD	Min	Max	Q1(25%)	Q2(50%)	Q3(75%)
N° conflicts .js	55.34	98.1	1	649	9	18.5	62.2
N° conflicting regions	102.9	189.2	2	1263	12.7	34.5	105.7
N° refactorings (conflicting files)	38.9	62.7	0	306	2	9	49.7
N° refactorings (conflicting regions)	16.2	32.1	0	194	0	2	13.2
N° relationship (refactorings and conflicting files)	227.1	838.5	0	6846	4	13.5	117.5
N° relationship (refactorings and conflicting regions)	24.8	61.5	0	387	0	3	18.25

variables representing merge scenarios, conflicts, and conflict regions. The selection of repositories with a wide range of data allows for the examination of whether the study's findings apply to both large and small repositories, with a large or small number of merge scenarios. The inclusion of outliers in the analysis further highlights the dispersion of the data and provides additional insights into the variations observed in the variables. By considering repositories with diverse characteristics, the study aims to enhance the generalizability and robustness of its results. Table 3.4 provides a comprehensive visualization of the two main research variables in the study, which are conflicts and refactorings. These variables were obtained through the examination of their evolutionary commits. Figure 3.3 shows dispersion metrics for selection repositories and the next section analyzes these variables.

To select repositories with a high number of merge and conflict scenarios, we chose to use the metrics "number of contributors" and "number of commits" as initial criteria. When analyzing the data presented in Table 3.3, we observed that our repositories had, on aver-

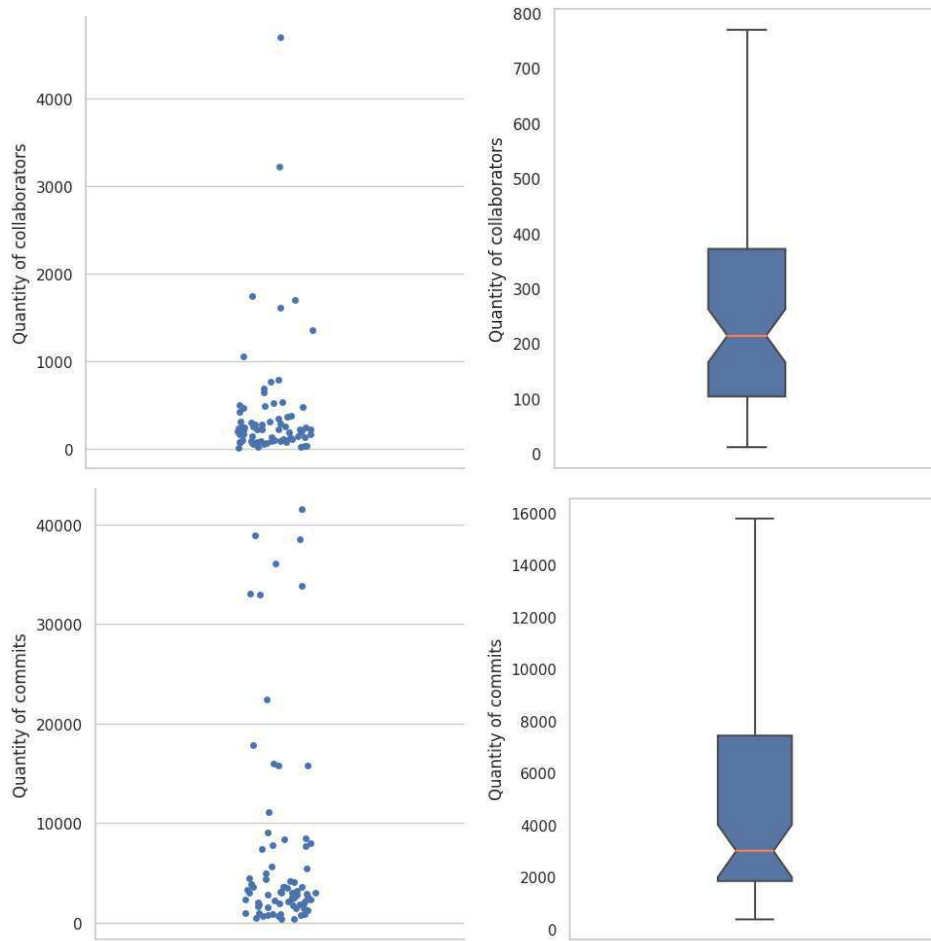


Figure 3.3: Initial metrics of selection repositories Javascript

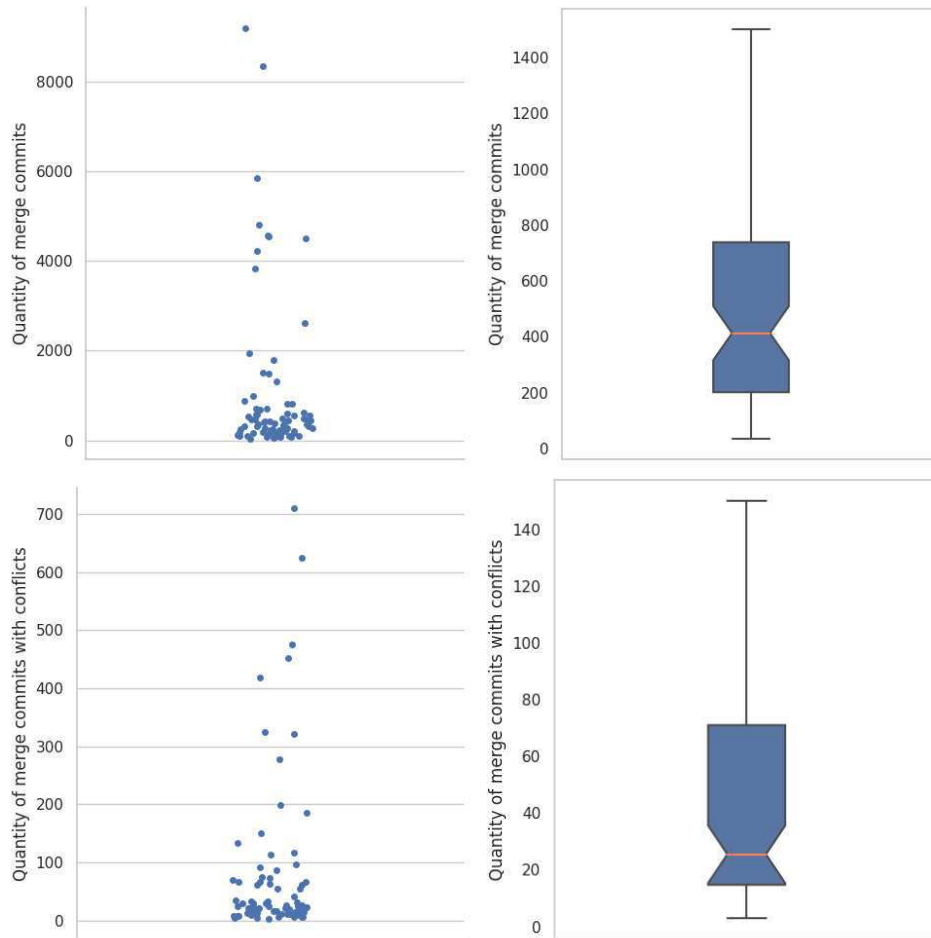


Figure 3.4: Dispersion metrics about merge commits and merge commits with conflicts

age, 412 contributors, ranging from a minimum of 12 to a maximum of 4,704. The notable dispersion of this data is positive, as it provides a comprehensive and meaningful representation for our analysis. To the number of commits metric, we found that the repositories analyzed had an average of 7,202 commits, with a wide dispersion of data, ranging from 392 to 41,503. This diversity suggests that our selection encompasses repositories of different sizes and stages of development, covering large, small, and medium projects.

Choosing these two initial metrics, each repository had the number of merge commits and merge commits with conflicts collected, starting the fundamental point for our study. The repositories presented around 1,077 merge scenarios, showing a notable dispersion of data. Of these scenarios, it was found that the mean is 83 scenarios with conflicts, with repositories having only 3 conflict scenarios while others with 710 scenarios. It is important to highlight that all variables selected so far have exhibited significant dispersion in the data

collected.

A correlation analysis was carried out between these variables, as illustrated in Figure 3.5. This analysis allows us to evaluate whether the two initial metrics chosen were effective in selecting repositories that contain representative merge scenarios.

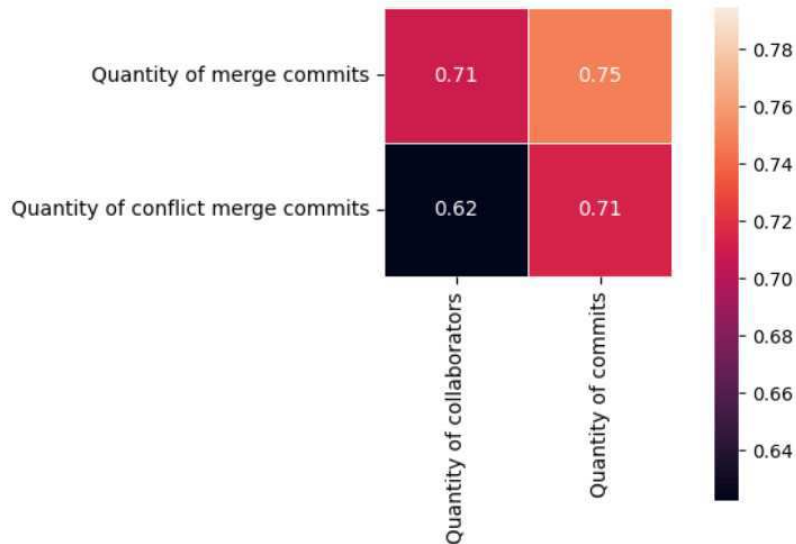


Figure 3.5: Correlation Matrix with metrics selection

A clearer correlation emerges between these variables, suggesting that as the number of contributors and commits in the repository increases, the likelihood of encountering more merge scenarios and conflicts also rises.

Figures 3.6, 3.7, 3.8 show us how our research questions variables are dispersed. Given that our study focuses on the relationship between refactoring action and conflicts that occur in merge scenarios, both at the file level and at the conflict region level, this subsection has the objective to provide insight into what was discovered for the variables "number of conflicts .js", "number of conflict regions", "number of refactorings in conflicting files", "number of refactorings in conflicting regions", "number of the relationship between refactorings and conflicting files" and "number of the relationship between refactorings and conflict regions".

As shown in Table 3.4 and Figure 3.6, we observed significant variability in the repositories, ranging from those with only one conflicting file to those with as many as 649 conflicting files. Additionally, we found repositories that contain from 2 to 1,279 conflict regions.

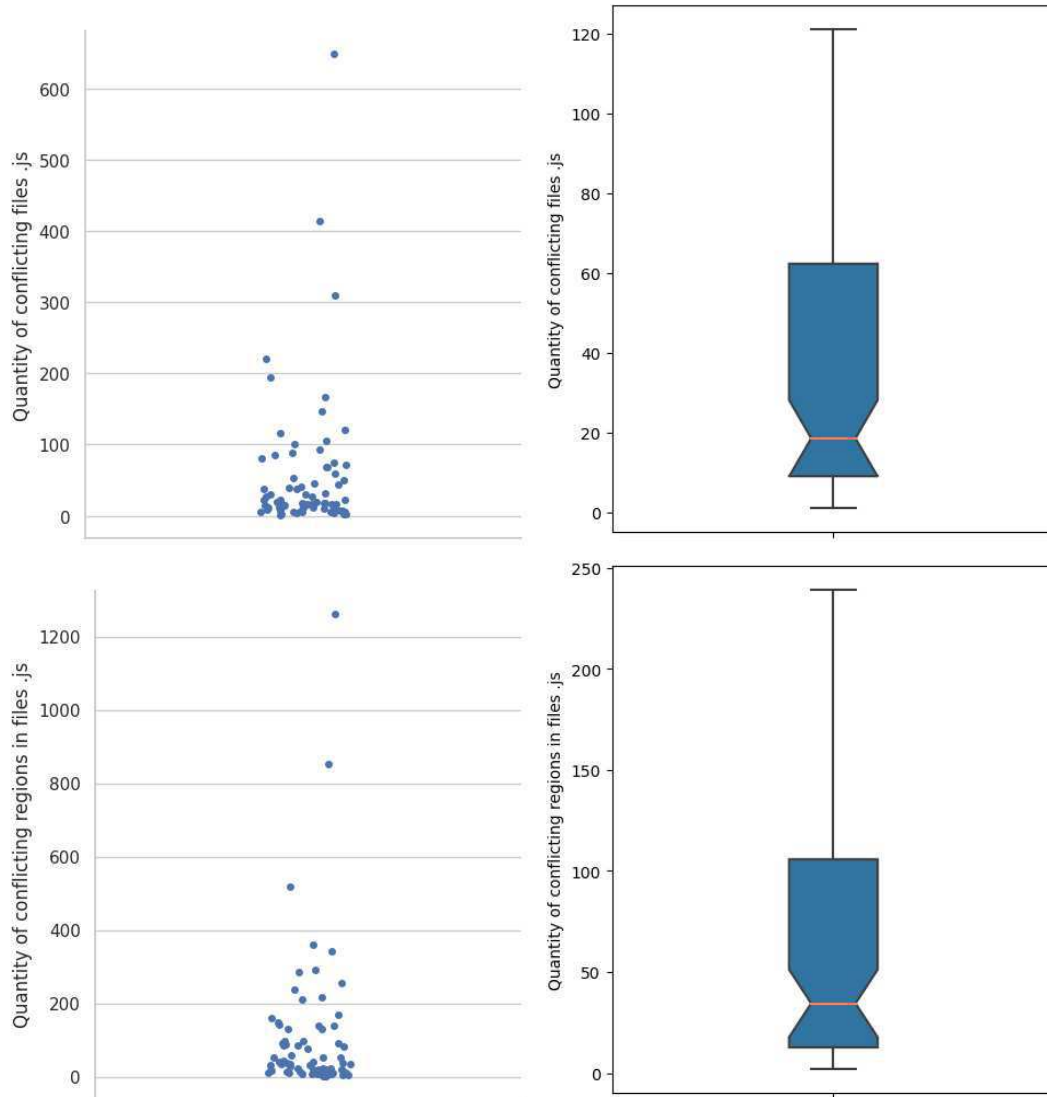


Figure 3.6: Dispersion metrics about conflicting files and conflicting regions

As evidenced in the Figures 3.7, there is a higher presence of relationships at the level of conflicting files compared to the level of conflict regions. In the context of conflicting files, scenarios of merge are identified, reaching up to 7,000 relationships with detected refactorings. Upon analyzing the boxplot, it is observed, through the median, that 50% of the data falls below approximately 13 relationships, while the average number of relationships is around 227. When we check at the conflict region level, we have a smaller quantity of relationships identified. Through our data, a disparity has been noted, which can be attributed to the randomness and diversity inherent in our repositories.

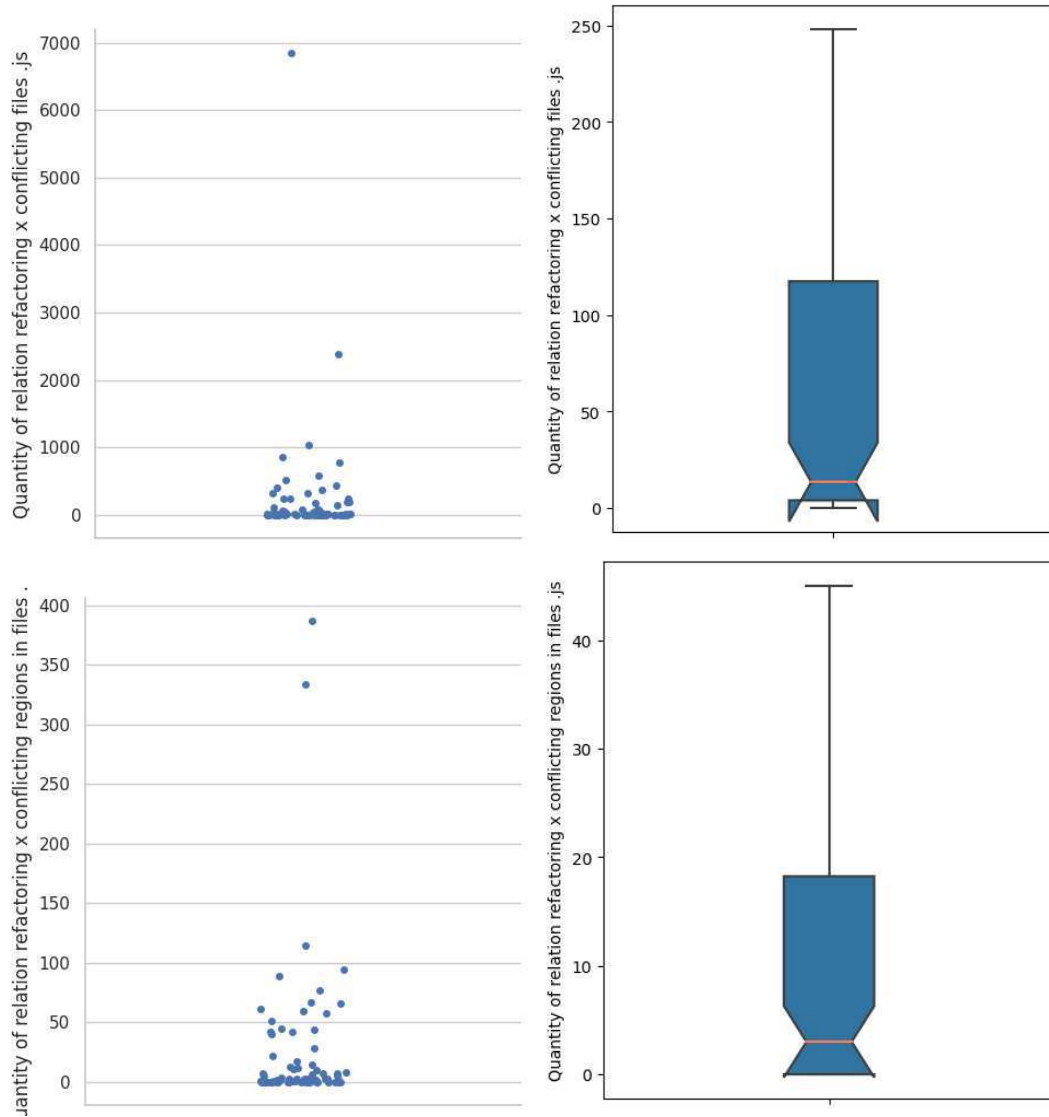


Figure 3.7: Dispersion metrics about relationship research variables for RQ1

Similar to the charts illustrating the number of relationships between refactorings and merge conflicts, graphs were generated to represent the dispersion of the number of refactorings in conflicting files and conflict regions. Concerning the number of relationships, the number of refactorings exhibited lower values, but with a significant dispersion and little presence of outliers, indicating a more balanced distribution at the conflicting file level.

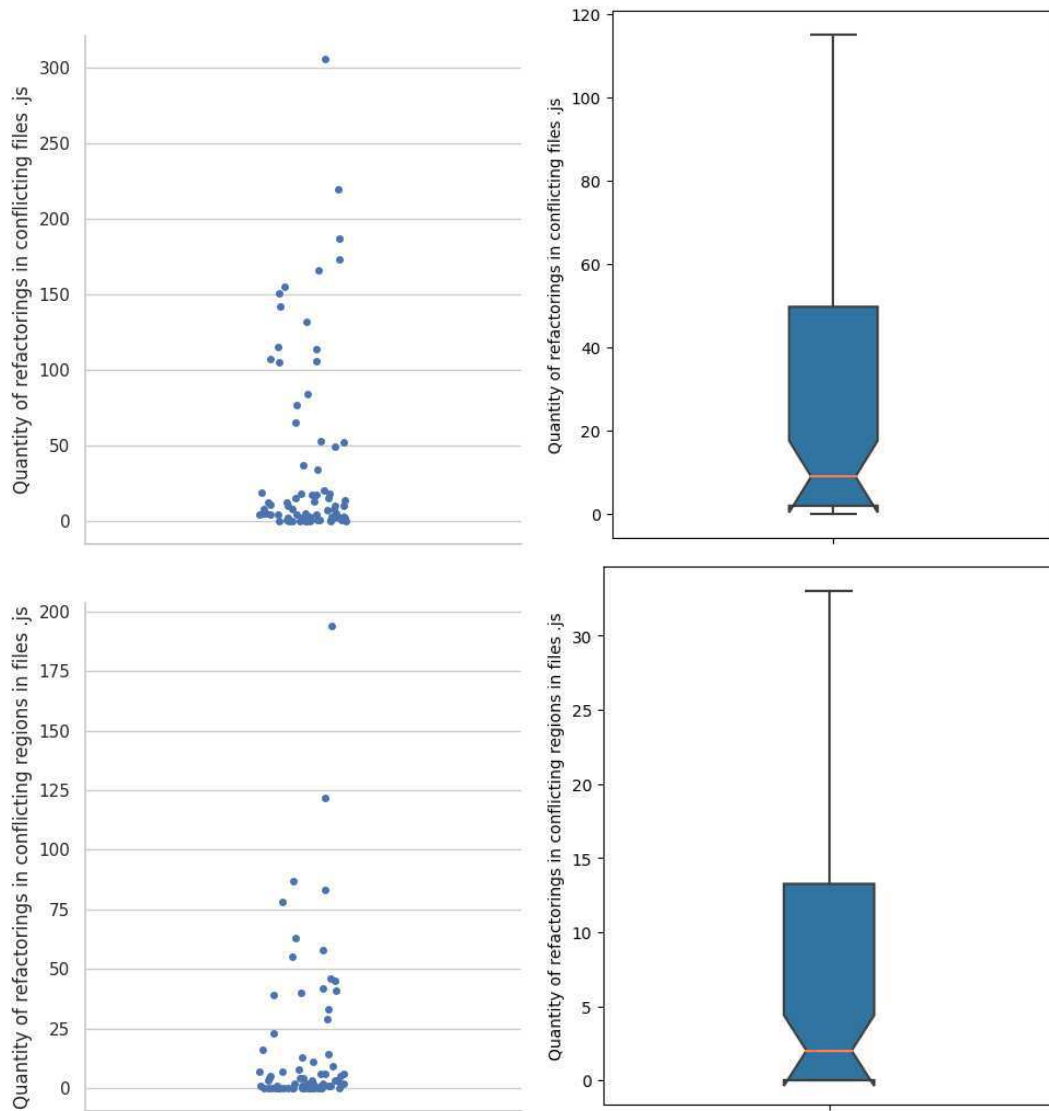


Figure 3.8: Dispersion metrics about the number of refactorings in research variables for RQ1



In the following figures, it is possible to check conflict situations and conflict regions captured by the methodology adopted. In this Figure 3.9, it is possible to observe the moment in which we identify the conflicting files in each merge scenario. In the case of the mentioned example, two files have merge conflicts: the test/test-async.js file and the lib/async.js file. To illustrate the example, we will choose the lib/async.js file, since it is an executable file in .js format. Later, in Figure 3.10, we present an example of a conflict region collected in this specific scenario.

```
joseglauber@joseglauber-Aspire-E5-574:~/Documentos/teste/async$ git checkout c64997e79593f83468f1db2517d40e385098414a
HEAD is now at c64997e Merge pull request #692 from wltsmrz/master
joseglauber@joseglauber-Aspire-E5-574:~/Documentos/teste/async$ git merge 2a13d0857682663e556b1a344d8c33d3a6c289bf
Mesclagem automática de test/test-async.js
CONFLITO (conteúdo): conflito de mesclagem em test/test-async.js
Mesclagem automática de lib/async.js
CONFLITO (conteúdo): conflito de mesclagem em lib/async.js
Automatic merge failed; fix conflicts and then commit the result.
```

Figure 3.9: Metric: number of conflicts

```
@@@ -166,2 -177,38 +200,43 @@@
++<<<<<< HEAD
+   async.eachLimit = function (arr, limit, iterator, callback) {
+     var fn = _eachLimit(limit);
+=====
+   async.forEachOfSeries = function (obj, iterator, callback) {
+     callback = callback || function () {};
+     var keys = _keys(obj);
+     var size = keys.length;
+     if (!size) {
+       return callback();
+     }
+     var completed = 0;
+     var iterate = function () {
+       var sync = true;
+       var key = keys[completed];
+       iterator(obj[key], key, function (err) {
+         if (err) {
+           callback(err);
+           callback = function () {};
+         }
+         else {
+           completed += 1;
+           if (completed >= size) {
+             callback(null);
+           }
+           else {
+             if (sync) {
+               async.nextTick(iterate);
+             }
+             else {
+               iterate();
+             }
+           }
+         }
+       });
+       sync = false;
+     };
+     iterate();
+   };
+
+   async.forEachLimit = function (arr, limit, iterator, callback) {
+     var fn = _forEachLimit(limit);
++>>>>>> 2a13d0857682663e556b1a344d8c33d3a6c289bf
```

Figure 3.10: Example of collected conflict

After collecting this conflict region, we identified the commit that was responsible for introducing this content into the source code, as we can see in Figure 3.11.

```

Commit 1fecb21940135b2ff647a93d1fc83df03b363714
Author: Caolan McMahon <caolan@caolanmcmahon.com>
Date: Sun Feb 10 22:40:20 2013 +0000

    rename forEach functions to each and add aliases for old names

diff --git a/lib/async.js b/lib/async.js
--- a/lib/async.js
+++ b/lib/async.js
@@ -147,2 +149,2 @@
-   async.forEachLimit = function (arr, limit, iterator, callback) {
+   async.eachLimit = function (arr, limit, iterator, callback) {
-     var fn = _forEachLimit(limit);
+     var fn = _eachLimit(limit);

```

Figure 3.11: Example of collected region conflict

The tool captures a RENAME refactoring that occurred in the `forEachLimit` function on line 147, being renamed to `eachLimit` on line 149, as illustrated in Figure 3.12.

```

147 -   async.forEachLimit = function (arr, limit, iterator, callback) {
148 -     var fn = _forEachLimit(limit);
149 +   async.eachLimit = function (arr, limit, iterator, callback) {
150 +     var fn = _eachLimit(limit);

```

Figure 3.12: Example of collected refactoring in conflict file and region conflict

Given the examples above, our script identifies that for this merge scenario, there is a relationship between the RENAME refactoring action and the conflicting file and the conflict region, since the refactoring was introduced in an evolutionary commit that made exactly the location of the region of conflict.

### 3.3.2 Answering RQ1: Is there a relationship between refactoring and merge conflicts in JavaScript programs?

After collecting variables related to the number of conflicts, conflict regions, and refactoring actions, we also collected the variable that represents the relationship between these variables, i.e., if there was at least one overlapping line between the location of the refactoring application and the location of the conflict, there is a relationship between both variables. Given this, out of the 76 repositories and 6,356 conflict scenarios analyzed, 17,271 relationships between refactoring actions and conflicting files were found. By restricting the

application of refactoring at the level of conflict regions, 1,888 relationships were found at the file level, these relationships are in Figure 3.13.

We also collected how many instances of refactorings were found in these conflicting scenarios, both at the file level and at the conflict region level. 2,961 were found instances of refactorings in conflicting files, 1,236 of which are also related to the region of conflict. The results can be seen in Figure 3.14.

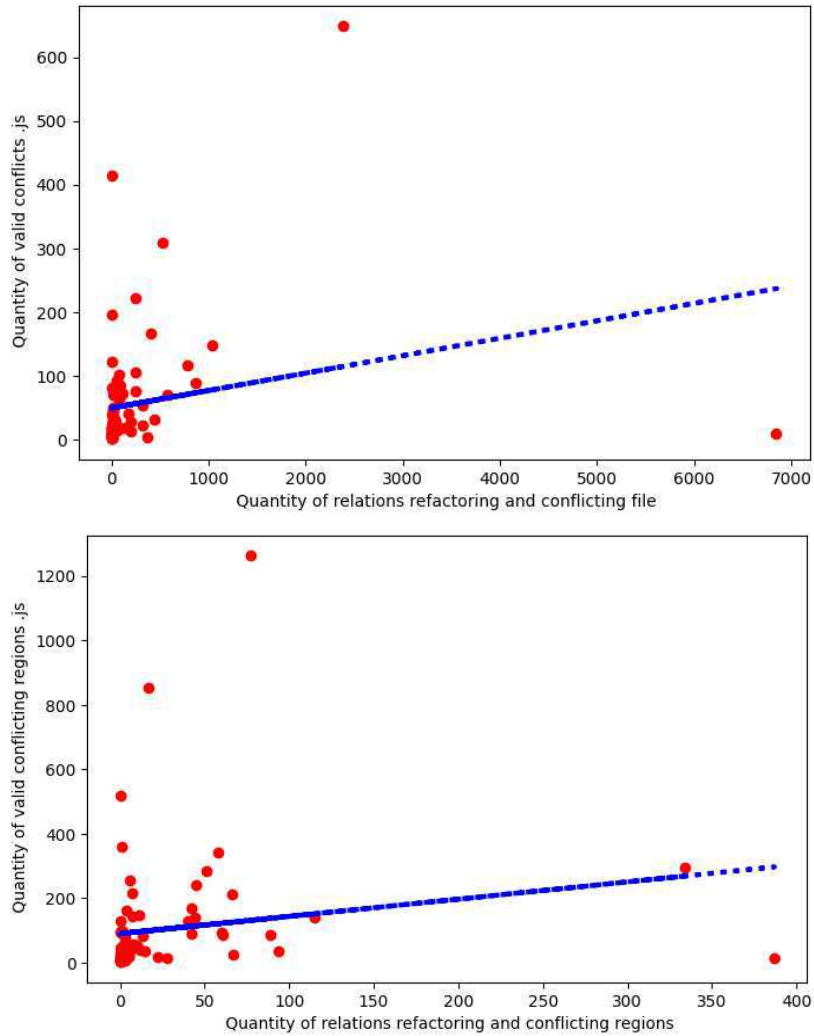


Figure 3.13: Dispersion graph of variables (relationship/conflicts) of QP1

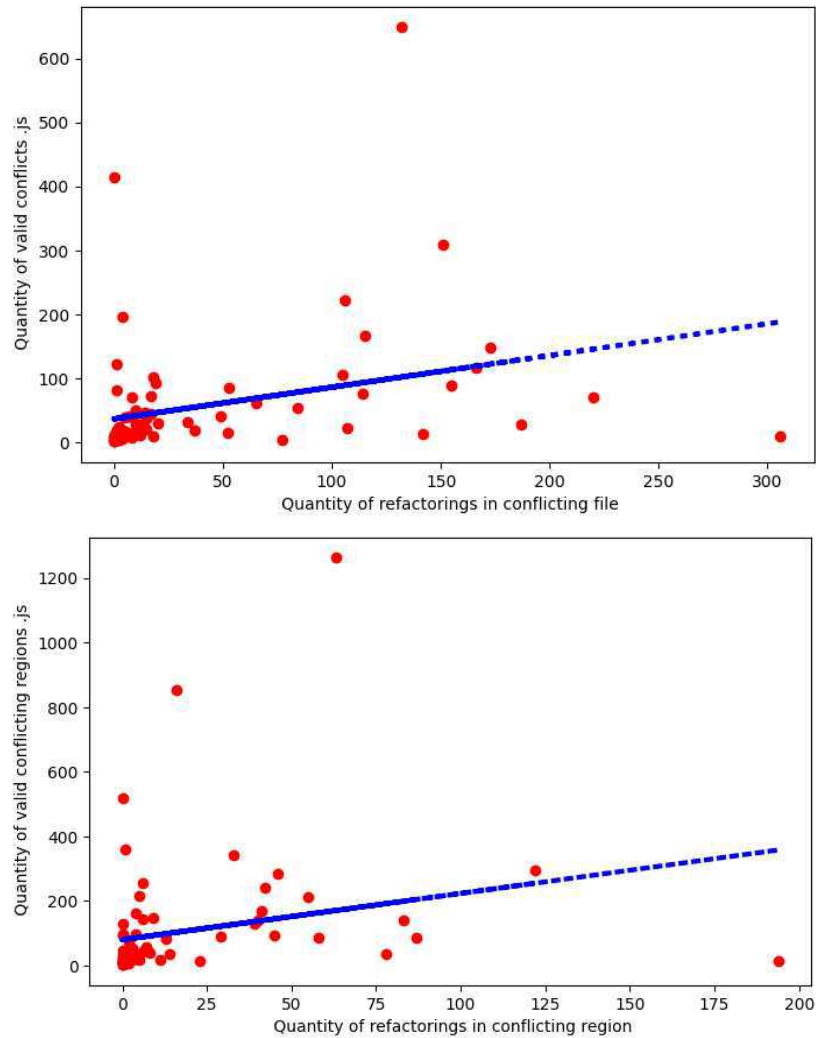


Figure 3.14: Dispersion graph of variables (n° of refactorings/conflicts) of QP1

The study identified 465 merge scenarios that have at least one relationship between the refactoring and the conflicting file, and of these, 253 have at least one relationship at the level of conflict region, representing 7% and 4% of the sample, respectively. Figure 3.15 represents how these data are distributed.

It can be observed that there are a low number of scenarios that have at least one relationship between the refactoring action and the conflicting file and its conflict region. To better analyze the relationship between these main variables of the research, the correlation between the number of conflicting files/regions and the quantity of this relationship was analyzed. Similarly, the correlation between the number of conflicting files/regions and the number of refactorings found in these scenarios was also examined. Table 3.5 shows some

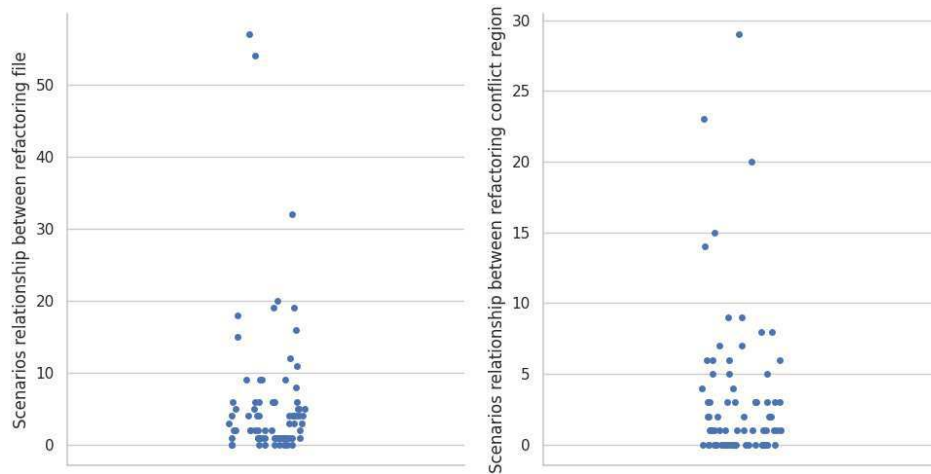


Figure 3.15: Dispersion graph of scenarios involved in merge conflict correlation relationships between them.

Table 3.5: Descriptive analysis of variables of study

Correlation	N° of conflicts .js	N° of region conflicts .js
N° of relationship between refactorings and conflicting files	0.55	0.61
N° of relationship between refactorings and conflicting regions	0.5	0.6
N° of refactorings in conflicting file	0.56	0.60
N° of refactorings in conflicting region	0.54	0.58

As observed, both relationships show a moderate positive correlation. In addition to the correlation analysis, a linear regression model was developed between both analyses. Our study developed linear regression models, where the dependent variable (Y) was defined as the "number of conflicting files/conflicting regions", and the independent variable (X) was defined as the "number of relationships between refactorings and conflicting file/number of relationships between refactorings and conflicting regions". These results can be analyzed in Table 3.6. When we now consider the linear regression model with the independent variable being the "number of refactorings", we observe different results in Table 3.7.

Table 3.6: Number of Relationship x Conflict

Relationship x Conflict	Conflicting file	Conflicting region
regression-model	$y = 49.1 + 0.027X$	$y = 89.6 + 0.53X$
p-value	0.042	0.13
r-squared	0.055	0.03

Table 3.7: Number of refactorings x Conflict

Number of refactorings x Conflict	Conflicting file	Conflicting region
regression-model	$y = 36 + 0.49X$	$79.6 + 1.43X$
p-value	0.005	0.03
r-squared	0.10	0.05

Both discussions about these linear regressions are in Section 3.4.

### 3.3.3 Answering RQ2: What refactoring patterns relate most to merge conflicts in JavaScript programs?

For this research question, information was collected on the types of refactorings present in scenarios that involve conflicts. A total of 2,961 instances of refactorings were found through the RefDiff 2.0 tool that was performed on files involved in conflicts, of which 1,236 were found within conflict regions. All eight types of refactorings that were analyzed in this study were found at the file level of conflicting files and region conflicts.

For each relationship found between a conflicting file/conflict region, there is an associated refactoring. Table 3.8 shows the frequency with which each type of refactoring was related to each of the conflicts and Figure 3.16 and 3.17 are graphics dispersion about how these data are distributed.

We can observe a notable disparity in the dispersion of data associated with the type of refactoring in the repositories. Although Internal Move was the most associated refactoring in conflicting areas, the scatterplots above indicate that Move refactoring is the most widely distributed among repositories, also presenting a smaller presence of outliers. Internal move

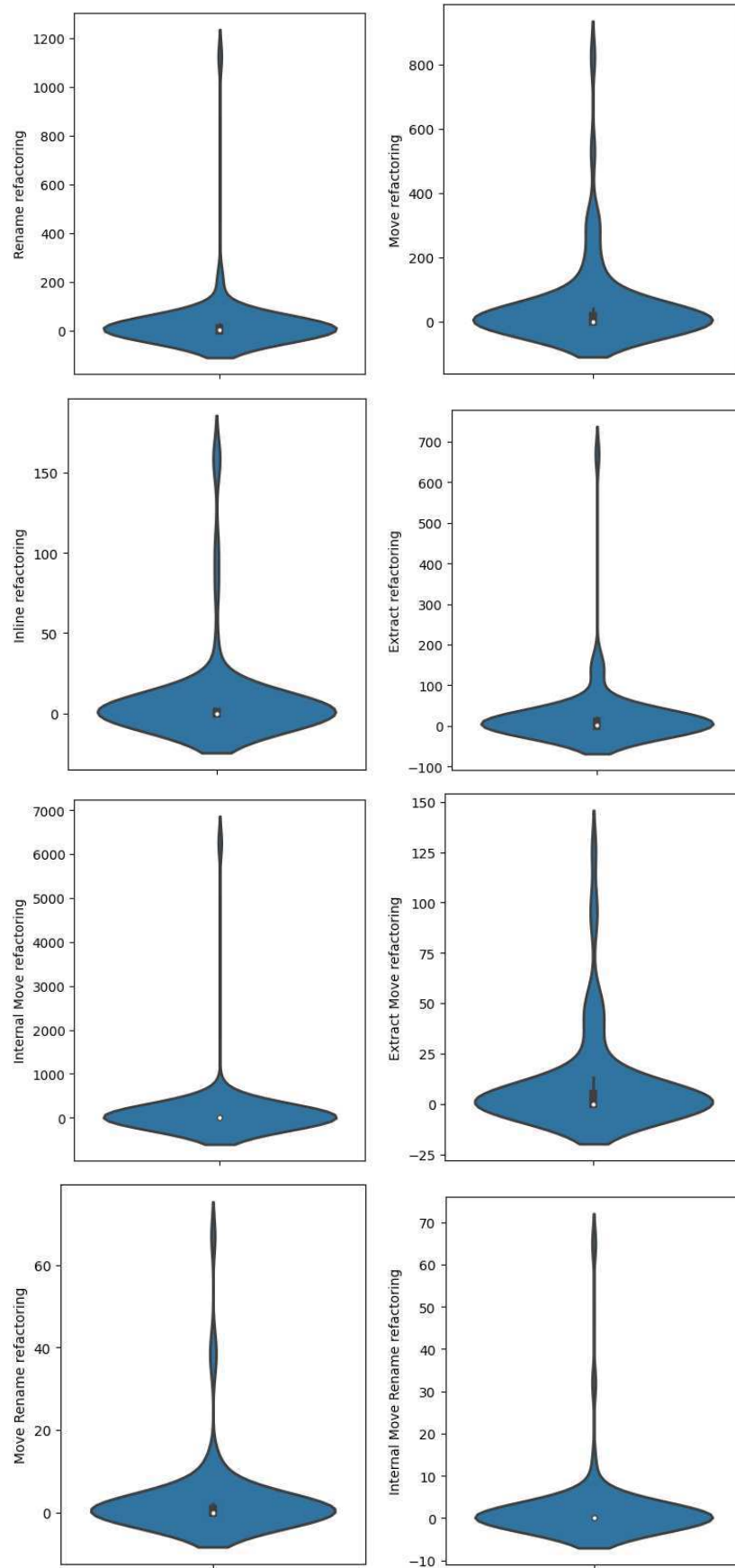


Figure 3.16: Violinplot to types of refactoring involved in conflicting file

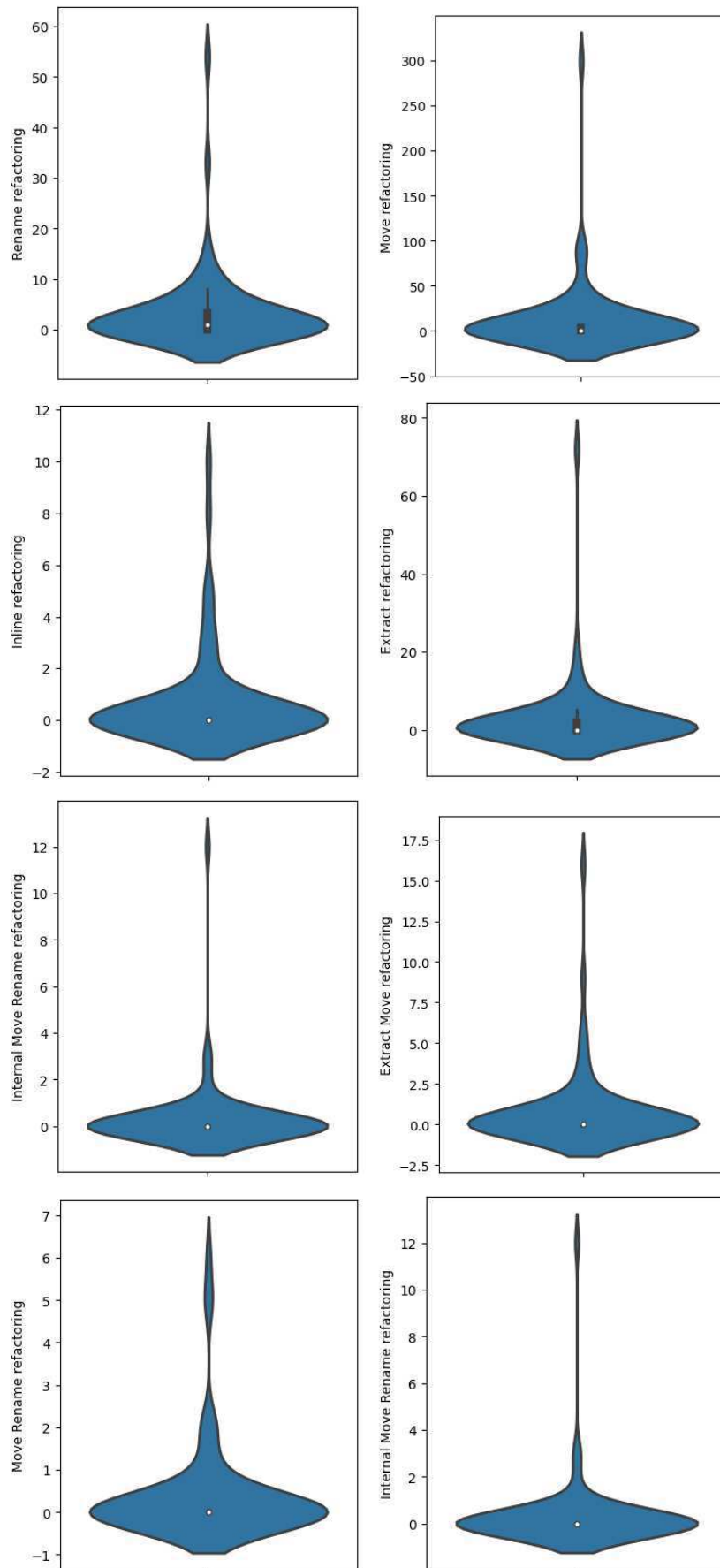


Figure 3.17: Violinplot to types of refactoring involved in conflicting regions



Table 3.8: Descriptive refactorings relationship founded in conflicting file and conflicting regions

Refactoring Type	Relationship Number in conflicting files	Relationship Number in region conflicts
<b>Rename</b>	2098	243
<b>Move</b>	3760	913
<b>Inline</b>	626	50
<b>Extract</b>	1713	203
<b>Internal Move</b>	8366	454
<b>Extract Move</b>	734	57
<b>Move Rename</b>	225	29
<b>Internal Move Re-name</b>	139	23

and move conflicts were the most commonly found when analyzing the conflicting file and conflict region levels. Below, statistical information about the data, as well as correlations, will be presented.

Similarly to what was done for RQ1, it is desired to verify the correlation between variables related to conflicts and the variable "number of refactoring types involved in the conflict". This verification was conducted at the level of conflicting files and also at the level of conflicting regions and is presented in Figure 3.19.

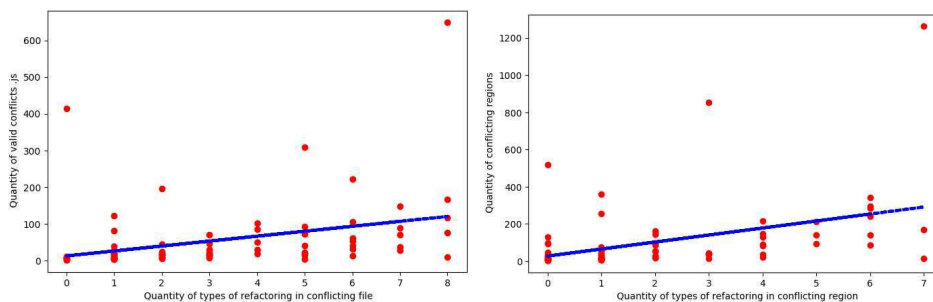


Figure 3.18: Dispersion graph of the relationship between the type of refactorings and conflicting variables

A moderately positive correlation is observed between the variables analyzed. In the

same way, as we addressed in the first research question, we conducted a linear regression between the analyzed variables. the results can be verified in Table 3.9. First, in Figure 3.18 are dispersion graphics that show the correlation between variables.

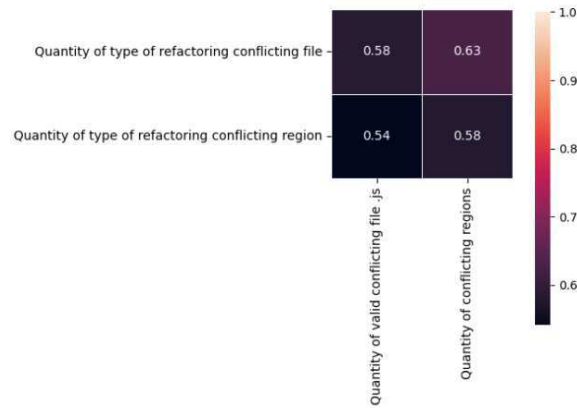


Figure 3.19: Correlation between quantity type of refactorings and conflicting file/region

Table 3.9: Number of Type of refactorings x Conflict

N° Type Refactorings x Conflict	Conflicting file	Conflicting region
<b>regression-model</b>	$y = 12.12 + 13.4X$	$y = 26.534 + 37.691X$
<b>p-value</b>	0.003	0
<b>r-squared</b>	0.11	0.18

For this regression model, we define the dependent variable (Y) as the "number of conflicts/conflict regions" and the independent variable as the "number of refactoring types in conflicting files/conflicting regions". The results of this regression linear will be analyzed in Section 3.4.

## 3.4 Discussion

In summary, our study found that from 6,356 conflicting scenarios, 465 merge scenarios have at least one relationship between a conflicting file and a refactoring action, around 7% of our total scenarios. Analyzing at the conflict region level, there are 253 conflict scenarios, representing around 4% of our sample.

When analyzing the results in Table 3.6, we observed that, when considering the impact of refactoring relationships in conflicting files, the p-value allows us to reject the null hypothesis, indicating the existence of a significant effect of the variable X over Y. The coefficient of determination (r-squared) reveals that variable X explains 5.5% of the variation in the data for variable Y. However, when examining the relationship at the conflict region level, we did not find sufficient statistical evidence to reject the null hypothesis, suggesting the lack of a significant relationship between the variables.

When we now consider the linear regression model with the independent variable being the "number of refactorings", we observe different results. At the conflicting file level, the p-value is notably low (0.005), indicating a significant relationship between the variables, with the independent variable explaining 10% of the variation in the dependent variable. However, at the conflict region level, we observed a more modest impact, with variable Y influencing up to 5% on variable X.

#### **Summary 3.4.1.** Results for RQ1

Through correlation and linear regression studies, our study demonstrates a bigger correlation between refactoring and conflicts at the level of conflicting files. This finding highlights the relevance of in-depth analysis of specific relationships between variables for a more complete understanding of the results, in addition to suggesting analyses regarding the impact of refactoring on the structure of the entire conflicting file.

Our study also focused on analyzing the number of refactoring instances that were related to conflict. 2,961 instances of refactorings collected by RefDiff 2.0 were found, of which 1,236 were also in conflict regions. Our statistical analyses showed a moderate correlation between the variables, showing a possible influence between the number of refactorings performed in the merge scenario and the number of conflicts that may occur, suggesting a deeper study of the relationship between the variables for better results.

To analyze not only the relationship between refactorings and conflicts, our study also uncovered results related to the types of refactorings performed within these areas. Through the relationships found in RQ2, our study found that Internal move, Move, and Rename are

the types of refactorings most related to merge conflicts, both at the conflicting file level and the conflict region. When conducting the study we found that many conflicts had more than one type of refactoring carried out. All 8 types of refactorings that the tool can collect in conflicting files and conflict regions were found.

When analyzing the results at the level of conflicting file and conflicting region in Table 3.9, we observed the two low values for the p-value, indicating that, in both cases, we can reject the initial null hypothesis that stated the non-existence of a significant relationship between the variables. We can therefore consider that there is a significant variance between them. When examining the  $R^2$  value, we find that, at the conflicting file level, approximately 11% of the variance in variable Y is explained by variable X, while at the conflict region level, this value is around 18%.

**Summary 3.4.2.** Results for RQ2

Through statistical analysis, we found promising results that demonstrate that the number of types of refactorings involved in the process can be directly related to the occurrence of the conflict in a merge scenario, this represented the most substantial relationship found in our study. In addition found that Internal Move, Rename, and Move types are most associated with merge conflicts, both at the file level and the conflict region level.

## 3.5 Threats to Validity

This section will present the threats to validity that were identified during the methodology of our study.

### 3.5.1 Internal Validity

Throughout our investigation, we noted certain inconsistencies in the functionality of the RefDiff 2.0 tool. The main issue challenge is the tool's incapacity to scrutinize refactoring actions within merge commits, coupled with some false positives and false negatives. The RefDiff study reported precision and recall of 91% and 88%, respectively, in identifying refactoring actions in JavaScript code [24]. In the context of evolutionary commits, our find-

ings indicate that the RefDiff tool faced challenges during the refactoring collection phase in certain instances. This was attributed to the distinctive characteristics of these commits, such as certain text formatting within the commit, resulting in parser errors.

In our study, our specific emphasis was placed on the analysis of files directly implicated in conflicts. Since the beginning, we opted for an approach exclusively dedicated to examining refactorings within the conflicted files. This strategic choice was largely shaped by the constrained tooling support accessible for JavaScript. To evaluate the influence of refactorings in non-conflicting files would have demanded substantial resources, both in terms of memory and time. The insufficient tooling support available for these files would have placed a substantial burden on the assessment process.

### **3.5.2 External Validity**

Our study is limited to the size of a selected sample, and therefore, the results presented here cannot be generalized to all JavaScript repositories. The conduct of our study on a limited number of repositories is due to the lack of suitable tools to streamline and automate the process.

Even with the selection of initial metrics, there is no guarantee that we chose the best repositories for evaluation. Extensive repositories may have a reduced number of conflicts, as other characteristics, such as those related to the team, can influence these variables. The process of collecting refactorings is also a relatively time-consuming procedure and requires specific configurations, making its application to larger datasets more challenging.

### **3.5.3 Constructor Validity**

When analyzing some merge scenarios, we observed the presence of untraceable commits, known as dangling commits, which lack references to any branch. These commits pose a threat to our study, as it is not always possible to extract the entire content of the conflict region when it contains dangling commits. In the context of refactorings, it is important to note that there is more than one way to perform the same type of refactoring. This variability can pose a threat to our study, compromising the effectiveness of refactoring detection by the RefDiff 2.0 tool. Due to the methodology of our study, the quantity of untraceable commits

was not collected, providing a potential avenue for future research.

A crucial aspect of our study aims to identify conflicts within the software source code. During code merging in Git, all files involved in conflicts are included, and not all of these files are executable JavaScript files with the `.js` extension. We observed the presence of various files, such as configurations, READMEs, and test files, among others. This diversity poses a threat to the validity of our study, as our focus is on identifying conflicts in executable JavaScript code files, typically developed by programmers.

It is important to show that we also encountered minified files, which is a compression process to enhance speed and save space. Although these files are automatically generated, they have the `.js` extension. To address these challenges, we applied a filter to our data, considering only files with the `.js` extension. Additionally, for minified files with the `.js` extension, we implemented a filter that checks whether the name follows a typical naming pattern for minified files, if there, we discard this file. This strategy ensured that we captured the most representative set of executable `.js` files in our analysis.

Similar to Mahmoudi et al.'s study [17], our research aims to identify refactoring actions in conflicting code. However, it is essential to note that the influence of refactoring on conflicts cannot be conclusively asserted without a more in-depth analysis of the conflict content.

### **3.5.4 Conclusion Validity**

The present study encounters challenges regarding the validity of its conclusions, with a specific emphasis on researcher bias. When selecting variables to address the research questions, there is a possibility of choosing variables that may not provide the best answers for our conclusion. To mitigate this issue, meetings and discussions were conducted to determine which variables to analyze, drawing on variables from other existing studies.

# Chapter 4

## An examination of commit evolutionary: floss or pure refactoring?

The third research question analyzes the content of the commits that created the conflict, so it is possible to verify whether this content is composed only of refactorings or other modifications, thus making it possible to better analyze the contribution of refactoring to the merge conflict.

To begin, in Section 4.1, we discuss the methodology employed in this study, which involves a systematic manual analysis. Figure 4.1 provides an overview of the methodology used to address the third research question, highlighting the steps involved in each process. Following that, in Section 4.2, we present the results used to answer our research question. In Section 4.3, we will present discussions of our previously presented results, and in Section 4.4, we conclude with a study's validity threats.

We answer the following research question:

- **RQ3:** The evolutionary commits that made conflicting code contain only refactorings (pure refactoring) or other modifications (floss refactoring)?

Next, each stage of the methodology for collecting and analyzing evolutionary commits is detailed.

## 4.1 Methodology

In this section, we will present the methodology for our third research question. In addition to the overall figure illustrating each step, we will provide tables detailing the inputs and outputs of each stage.

### 4.1.1 Methodological study for RQ3: Analyzing the content of evolutionary commit (floss and pure refactoring)

Through methodology 1, we identified the evolutionary commits that built the region conflicts. At the moment, our study verifies the occurrence of refactoring actions in regions of conflict by comparing edited lines in the evolutionary commit and the output of RefDiff 2.0 that indicates the location of the refactoring action. To get value for our study we decided to adopt a strategy that analyzes the content of the evolutionary commit code involved in conflict through a manual analysis, to check whether evolutionary commits have only refactorings in their code sent or which are also composed of other types of modifications.

It is essential to discuss that when implementing the methodology for the third research question, we conducted a systematic analysis, without including automatic semantic analysis. The decision to perform this analysis manually was driven by a lack of knowledge regarding tools capable of automating this task, due to limitations in studies within the JavaScript domain. The step-by-step methodology is detailed in Figure 4.1 and the next subsections.

To better illustrate the metrics that be used as inputs and outputs in each activity of the RQ3 methodology, the following Table 4.1 has been developed, with a detailed description of each step provided subsequently.



Table 4.1: Descriptive analysis of metrics repository selection

Activity	Description	Input	Output
1	Select a sample of evolutionary commit	Extracted data evolutionary commits	Sample of evolutionary commits and variables - quantity of files involved evolutionary commits and quantity of files collected by RefDiff 2.0
2	Compare refactorings by RefDiff 2.0 and GitHub interface and looking for others changes	Sample of evolutionary commits and collected variables	Metrics about evolutionary commits
3	Categorizing floss and pure evolutionary commit	Evolutionary commits analyzed	Summary of floss and pure refactoring evolutionary commits

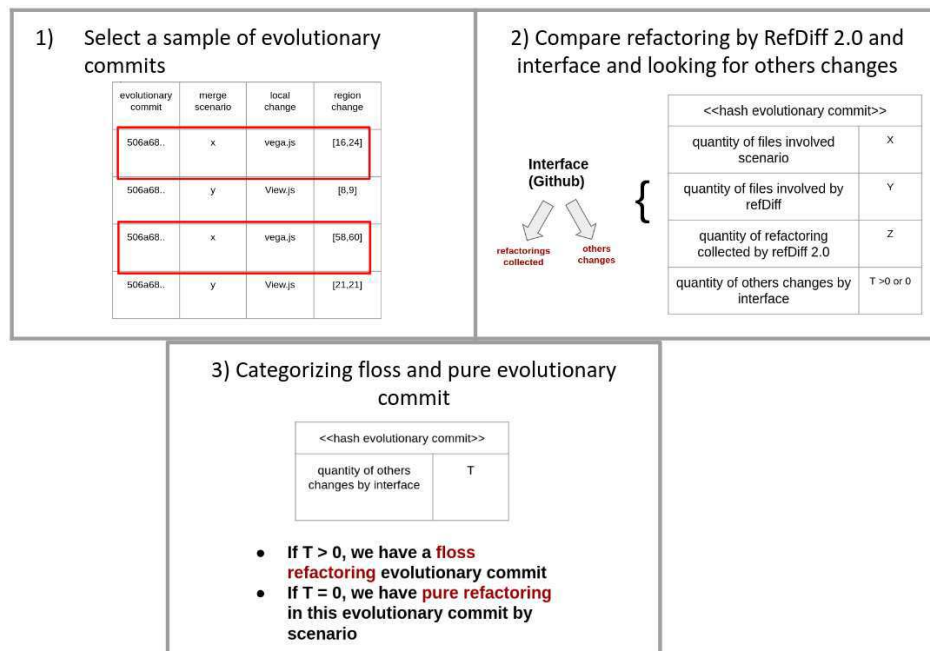


Figure 4.1: Methodology for manual analysis of content by evolutionary commit

### Select a sample of evolutionary commit

In this initial stage of the methodology, a stratification was executed on our data regarding evolutionary commits. As we already know the modifications of each commit evolutionary by merge scenario, we choose a representative sample by sample calculator <sup>1</sup> that considers

<sup>1</sup><https://comentto.com/calculadora-amostal/>

64 repositories from 76 that we have, this value guarantees a significant sample with a significance of 95%. From these 64 repositories we selected a sample of evolutionary commits that have at least one refactoring.

### **Compare refactorings by RefDiff 2.0 and GitHub interface and looking for other changes**

At this stage of our methodology, we extract from these commits in the GitHub interface how many files were involved in the merge and how many files were collected by the RefDiff 2.0 tool with refactoring actions. We look for the refactorings collected by RefDiff 2.0 and any relationships that exist with them. Soon after, we look for other modifications that are not related to refactoring. With this, we collect the variables: the number of files involved in the merge scenario, the number of files involved by RefDiff, the number of refactorings collected by the tool, and the number of other changes identified by the GitHub interface.

### **Categorizing floss and pure evolutionary commit**

For our study, we established the classification of "floss refactoring" for any modification that was not identified as refactoring by RefDiff 2.0 or, even if not found, fit the definitions in Section 2. We considered a modification as "pure refactoring" only when the evaluated commit exclusively contained refactoring operations. We defined that the scope of analysis would be the evolutionary commit since, from the outset, it is used as our primary object for collecting information on the variables. Any addition of functionalities, test files, build files, and minified files will be classified as "floss refactoring" if included in the evolutionary commit.

## **4.2 Results**

The variables collected at this stage of our study helped us identify which changes were involved in the commit that created the conflict. For each repository and each evolutionary commit we collected the metrics: "the number of files identified with refactoring by RefDiff 2.0", "number of modified files identified in the source code in the GitHub interface", "number of refactorings identified by RefDiff", "number of other modifications identified in the

source code in the GitHub interface". With these variables collected through our methodology described previously, we were able to identify which commits in our sample were floss or pure refactoring.

It is worth mentioning that as it was a manual analysis, we were careful to analyze the refactorings and modifications involved in the process, so whenever we identified false spurious errors (refactorings collected by RefDiff) they were disregarded from our study, to obtain good accuracy of our results. In Figure 4.2 below we can identify an example of a pure refactoring commit. We can verify that in this commit only one file was modified with only the Rename refactoring modification.

The screenshot shows a GitHub commit titled "fix parseAssignableListItem function name misspelling". The commit is by user **sebmck** and was committed on Jan 25, 2015. It shows 1 changed file, `acorn.js`, with 3 additions and 3 deletions. The diff view shows the following changes:

```

2414 2414     while (!eat(close)) {
2415 2415         first ? first = false : expect(_comma);
2416 2416         if (tokType === _ellipsis) {
2417 -         elts.push(parseAssingableListItem(parseRest()));
2417 +         elts.push(parseAssignableListItem(parseRest()));
2418 2418         expect(close);
2419 2419         break;
2420 2420     }
2421 2421     var elem;
2422 2422     if (allowEmpty && tokType === _comma) {
2423 2423         elem = null;
2424 2424     } else {
2425 -         elem = parseAssingableListItem(parseMaybeDefault());
2425 +         elem = parseAssignableListItem(parseMaybeDefault());
2426 2426     }
2427 2427     elts.push(elem);
2428 2428 }
2429 2429 return elts;
2430 2430 }
2431 2431
2432 - function parseAssingableListItem(param) {
2432 + function parseAssignableListItem(param) {
2433 2433     if (eat(_question)) {
2434 2434         param.optional = true;
2435 2435     }

```

Figure 4.2: Example of pure evolutionary commit involved in conflict

For the evolutionary commit floss refactoring, according to our definition, we considered everything that was not a `.js` file and was not related to refactoring as an extra modification.

Commits with modifications to HTML files, build, tests, minimized files, CSS, etc. were considered as extra changes, in addition to changes that were not refactorings identified by RefDiff and were not within the scope of refactorings that we defined in the Background section. In Figure 4.3 we can see an example of an evolutionary floss refactoring commit.

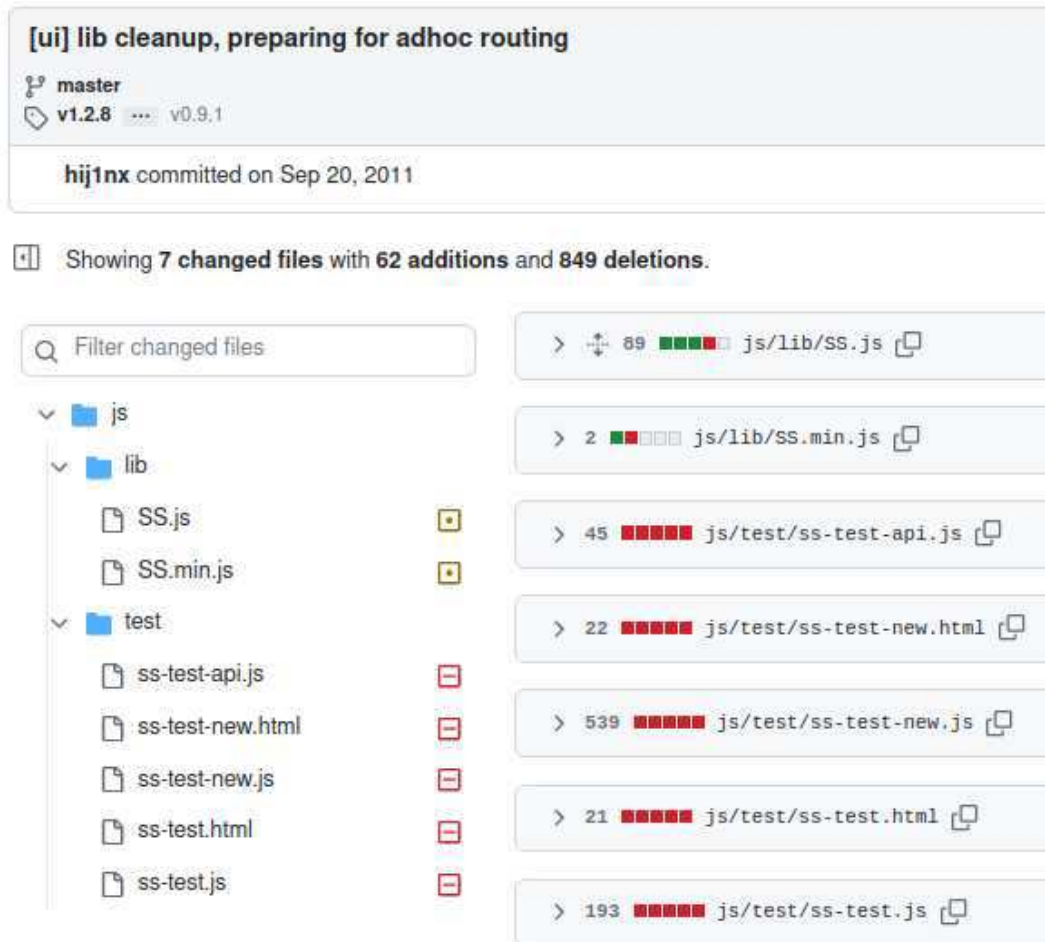


Figure 4.3: Example of floss evolutionary commit involved in conflict

#### 4.2.1 Answering RQ3: The evolutionary commits that made conflicting code contain only refactorings (pure refactoring) or other modifications (floss refactoring)?

Given all of our correlation analysis between the refactoring and merge conflicts variables, our third research question has focused on examining the content of these regions through the

evolutionary commits. The study analyzed a sample of evolutionary commits to determine whether they were generated through "floss refactoring" or "pure refactoring", allowing for better identification of refactoring's responsibility for the conflict. A manual analysis was performed on 64 of the 76 repositories in our sample, covering 535 evolutionary commits. Of these, 448 commits are "floss refactoring" (84%), and 87 are "pure refactoring" (16%). The distribution of data by repository is shown in Figure 4.4. Of the 64 repositories to this question, 33 have all the evolutionary commits involved in the conflict process classified as floss, which represents more than 50% of our sample of repositories.

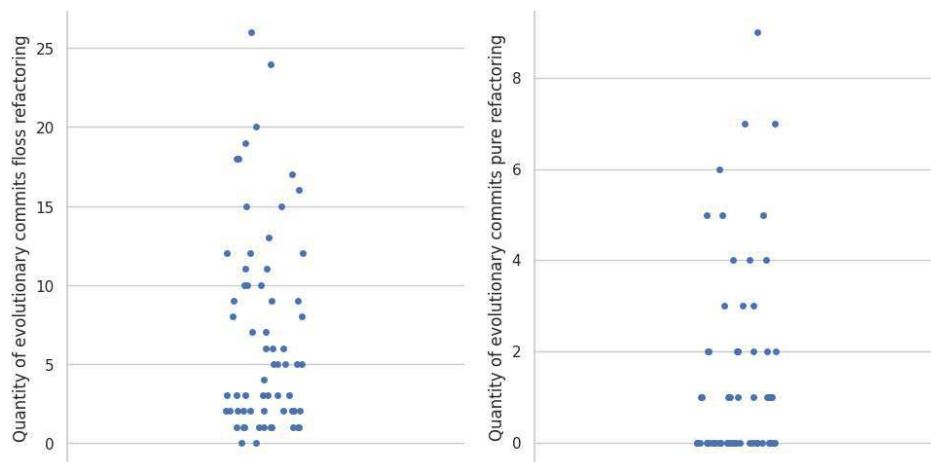


Figure 4.4: Dispersion of evolutionary commits floss/pure of QP3

### 4.3 Discussion

During the manual analysis of our study, we were able to check the instances of refactorings and their applications in the commit on the GitHub interface. It was found that the tool considered some modifications to the build file as refactorings. For our study, we did not take these instances, to have a more accurate classification of floss and pure refactoring. Our study did not evaluate if the program's behavior was preserved after executing the refactoring, this was because we did not have tools for this analysis, in summary, we checked if there was a relationship between the tool's output and the modifications involved in the interface, or if there are any instance followed the refactoring pattern defined in the Background section.

A point highlighting is that many evolutionary commits have other modifications made

along with refactorings, like build and test files being the most predominant. Another very recurring modification seen was configuration files (files that are not .js) indicating future work that analyzes the configuration setup of the JavaScript code and the occurrence of conflicts.

This manual analysis allowed us to identify that many of the scenarios previously investigated in QP1 and QP2 include other modifications in their evolutionary commits, as per the sample used in QP3. It was found that only about 16% of these analyzed commits exclusively contain refactoring operations. This finding provides an intriguing starting point for future research, as it enables the analysis of scenarios with pure refactorings, allowing the identification of whether refactoring was indeed the cause of the conflict. Additionally, our study contributed to the creation of a dataset containing floss and pure refactoring in JavaScript, establishing a valuable foundation for future investigations.

Based on the results found in our study, we can observe the need to explore the true cause-and-effect relationship between these variables. Our manual study identified many floss refactoring commits, so it is crucial to investigate what other modifications these are and whether they could have been the cause of the conflict. As future work, we can suggest the development of discussions on programming best practices and the necessary tools for exploring floss and pure refactoring in JavaScript.

#### **Summary 4.3.1.** Results for RQ3

To analyze the content of analyzed conflicts, our manual analysis found that of the 535 evolutionary commits analyzed, 448 of them were classified as floss refactoring and 87 as pure refactoring, 84%, and 16% respectively. Most of the repositories analyzed (52%) had all evolutionary commits as floss, demonstrating a large load of other modifications that are performed together with refactoring actions.

## **4.4 Threats to Validity**

This section will present the threats to validity that were identified during the methodology of our study.

### **4.4.1 Internal Validity**

During the manual analysis, we examine the refactorings identified by the tool and the presence of other modifications, which may or may not follow the refactoring pattern defined in the Background section. This approach introduces a threat to internal validity as the analyses are based on the subjective knowledge of the researcher, potentially introducing bias. Manual analysis inherently carries risks, as an automated approach based on accurate metrics may bring more consistent results.

### **4.4.2 External Validity**

As the analysis is conducted manually, our study is confined to a specific sample, and the results cannot be generalized due to the limited size of the sample. To mitigate this issue, we applied a calculation to obtain a sample with 95% confidence, providing a more robust foundation for our conclusions.

### **4.4.3 Constructor Validity**

Our study chose to perform an analysis of "floss" and "pure refactoring" through evolutionary commits. If we wanted to examine more rigorously the influence of refactoring on conflicts, we could have adopted an analysis at the level of the conflict, considering that the scope of the evolutionary commit is more comprehensive. This choice impacts construct validity, as the granularity of the analysis can influence the interpretation of results.

### **4.4.4 Conclusion Validity**

The present study faces challenges related to the validity of its conclusions, particularly highlighting the bias associated with the number of analyzed repositories and researcher bias. The drawn conclusions relied on the researcher's expertise and were derived through a manual analysis. To mitigate the inherent bias in manual analysis, the study was systematically conducted, focusing on the identification of refactorings without considering whether they preserved the code's semantics. It is important to acknowledge that the study has limitations due to the absence of JavaScript tools capable of automating these processes, underscoring

the need for future developments in this area.



# Chapter 5

## Related Work

This section presents several studies that provide a solid base for our study. This research has two main variables, refactorings and merge conflicts, so this chapter will focus in to show the contributions of these variables in this area. About refactorings, it is essential to show studies that discuss the beginning of the problem, where the necessity of the refactorings comes up, like bad smells. Furthermore, it presents studies that have techniques to analyze refactoring actions in software programs. On the other side, about merge conflicts, we have studies that discuss techniques to merge code without conflicts, and consequences if a conflict exists. There are a few studies that investigate the relationship between these two variables, and our study comes to an evolution about the investigation and refactoring and merging conflict in Javascript code.

There are a lot of studies to language Java that discuss bad smells like your begin and consequences [29], [25]. Barros and Adachi in their study [4] have a mapping investigation about code smells in Javascript code, verifying if the bad smells were defined and this definition. This study analyzes 8 different works published between 2013 and 2020, identifying 26 different types of bad smells that have been defined for the Javascript language and how these bad smells as evolved. Similarly, the study of Johannes et al. [15] has a large-scale empirical study about code smells. This study focuses on extracting code smells in repositories, resulting in 12 types of code smells in 1807 releases. The main contribution of this study was a better investigation of how the code smell persists in the system. The researchers found that files without code smells have hazard rates of at least 33% than files with code smells, in addition to discussing types of refactorings that are most involved with problems.

---

Both studies are important to our research because they bring the beginning of the discussion of refactorings, since Fowler [10] discusses that the exact moment to apply refactoring is at the start of code smell.

Martin Fowler in his study [10] brings a significant contribution to the definition of refactorings, showing ways to make a design code better. This study is very important because is the first to classify and discuss patterns and step by step to make a better refactoring. There is a large of refactorings that he presents in his study, refactorings like extract, inline, move and rename. These types of refactorings are present in our study, but analyzed in Javascript code. Base of definitions brought to Fowler, Opdyke et al. [21] show more about refactoring actions applied in object-oriented frameworks, showing how to automatically apply these, detailing three of the most complex refactoring and designing constraints needed in a refactoring. Studies like [7], [8], [27] present discussions about the process to apply refactorings automatically, some proposes are tools that implement JavaScript refactorings based on pointer analysis, others have an approach based in a static analysis.

Silva et al. in their study [24] proposed a multi-language refactoring detection tool - RefDiff 2.0. Different from the other tools in academics, RefDiff 2.0 is specific to detect actions refactorings applied by developers in a software evolution. This tool is the first to collect these information about refactoring to Javascript code, and it will be used in our study. They have significant precision and recall to Java, Javascript and C languages, detecting refactorings like move, rename, extract and inline. To Javascript the tool has a precision and recall of 91% and 88% respectively.

To merge processes, there a substantial studies that provide a better explanation of techniques merging. Mens in his study [18] provides a state-of-the-art about software merging, he discusses the technique of two-way and three-way merging, also textual, syntactic, semantic and structural merging. In this work, he shows that 90% that all merge scenarios need unstructured merge (textual) because of they simplicity and only 10% need more complex merge, like semi-structured or structured merge, also discusses that all VSCs uses textual merge because have more efficiency, scalability and accuracy. Tavares et al. [26] analyzing the benefits of using semi-structured merge instead of unstructured in Javascript code, analyzing by the perspective of true positives and false positives. In repositories that he analyzed merge techniques, he found that the semistructured merge tool JSFSTMerge reports fewer

---

spurious conflicts than unstructured merge, but this gain is smaller than semistructured merge tool to Java code, showing that this area of merge tools in Javascript code needs more studies to better results. Ghiotto et al. in their study [13] introduce a search-based approach algorithm to minimize conflicts merge. Additionally, Apel et al. in [3] presented developed tools for Java, C# and Python to reduce the number of conflicts.

Ahmed et al. in [2] presents an empirical examination of the relationship between code smells and merge conflicts. Their objective was to analyze if entities that contain certain types of code smells are more prone to be involved with merge conflicts. Additionally, they investigated if these "smelly" entities are also associated with other types of changes. To achieve this, they mined 143 repositories from GitHub. The results of their study revealed that poor design choices have a significant impact on maintainability, merge operations, and the overall quality of the resulting code. Specifically, they found that two code smells, namely Blob Operation and Internal Duplication, were the most frequently associated with merge conflicts. This research shows the importance of identifying code smells and showing design decisions to mitigate the occurrence of merge conflicts.

Mahmoudi et al. in their study [17] was the first study to verify the relation between refactorings and merge conflicts. In their paper, they perform an empirical study in almost 3000 well-engineered open-source Java software repositories and collect 15 popular refactoring types. The findings revealed that a significant portion, specifically 22%, of the observed merge conflicts involved refactoring operations. Furthermore, the study identified the Extract Method as a particularly problematic refactoring type involving merge conflicts. This shows that caution must be exercised when applying this specific refactoring technique to avoid potential conflicts during the merging process. Both studies cited were developed by Java because of the specified characteristics of language and its popularity, a portion of our study follows the methodology of Mahmoudi et al. study [17], involving the collection of evolutionary commits by identifying conflict regions. However, our focus is focused on the JavaScript language.

# Chapter 6

## Conclusions

Throughout the software evolution process, a series of activities are constantly performed to enhance its quality. One example of these activities is refactoring, which seeks to improve the quality of the internal code structure. It is crucial to perform these preventive actions throughout the software's entire lifecycle. Due to the need for code integration during this lifecycle, code merges come into play. These merges are facilitated by Version Control Systems, enabling the incorporation of local code into a global context, thus promoting software evolution. However, these merges are not always successful, leading to well-known merge conflicts. In-depth studies have been conducted to analyze aspects related to code design, the identification of code smells, as well as the execution of refactoring actions in source code, and how these elements can impact the occurrence of merge conflicts.

In this work, we present an empirical study that provides initial insights into the occurrence of refactorings in JavaScript code conflicts. The first part of this study aims to analyze the presence of refactoring actions in conflicting files and conflict regions, identifying which types are more closely related to the conflicting area through the analysis of evolutionary commits, and examining the commits that contributed to the conflict region. In the second stage of our investigation, we focus our analysis on the content of evolutionary commits, adopting a perspective that distinguishes between "floss refactoring" and "pure refactoring." This multifaceted approach offers a deeper understanding of the dynamics of refactoring amid code conflicts in JavaScript projects.

To examine the occurrence of refactorings in conflicts, we developed a quantitative approach that traced the origin and destination of refactorings, checking the edited lines during

---

conflicts. We identified the presence of refactorings in conflicts when there was a match between these lines. In addition to assessing this alignment between lines, our study also quantified the number of instances of refactorings in these conflicting areas. The collection of this data was facilitated through scripts developed for extraction in the selected JavaScript repositories. These scripts were responsible for extracting metrics that addressed the research questions outlined in Chapter 3.2.2.

For the first part of the study, we identified that approximately 7% of merge scenarios involve at least one refactoring action in conflicting files. Of these, 4% exhibit this refactoring at the level of the conflict region. Our statistical analyses revealed a moderate and positive correlation between refactoring and conflicts, at the file and conflict region levels, approximately 0.6 in a Spearman correlation. When applying simple linear regressions, we established the null hypothesis of no relationship between the analyzed variables, meaning that the presence of refactoring would not influence merge conflicts. However, upon examining the number of relationships between refactoring and conflicts at the level of conflicting files, we could reject the null hypothesis, indicating that about 5% of the data in the dependent variable Y (number of conflicts) is explained by the independent variable X (number of relationships). When analyzing the conflict region level, we found no evidence of a significant correlation.

In the context of the quantities of instances of refactorings within the conflicting file and conflict region, we could reject the null hypothesis in both cases, suggesting a significant relationship between the variables. The R-squared value obtained indicates that approximately 10% of the data in the dependent variable X (number of conflicts) is explained by the influence of the independent variable (number of instances of refactorings), while at the conflict region level, we find a scenario where 5% of the data is explained by the independent variable. These results suggest an initial relationship between conflicts and refactorings when analyzed from the perspective of line overlap. Despite the variability of the data regarding the influence of the independent variable on the dependent variable not being substantial, these results indicate an influence that serves as a starting point for future studies on this relationship in JavaScript code.

Still, within our first study, we conducted a quantitative analysis to quantify the types of refactorings found in the relationships established in Research Question 1 (RQ1). The types

---

of refactorings "Internal Move," "Move," and "Rename" stood out as the most associated with the identified conflicts. The distinctive aspect of our study lies in the realization that "Internal Move" is the type of refactoring most related to conflicts. This type of refactoring was frequently observed in JavaScript code due to the specific characteristics of the language, which allow for the nesting of functions. In addition to discovering the types most associated with conflicts, similar to the static analyses of RQ1, we found a significant correlation between the number of types of refactorings and the occurrence of conflicts, both at the level of conflicting files and at the level of conflict regions. For the dependent variable Y (number of conflicts), it was found that approximately 11% of the data is explained by the independent variable X (number of types of refactorings). Analyzing the independent variable Y as the number of conflict regions, we found that about 18% of the data is explained by the independent variable X. These results indicate a stronger association between the study variables, representing a possible relationship between the number of different types of refactorings and the occurrence of conflict. These findings serve as a starting point for discussions on the threshold of types of refactorings in a commit.

Given the statistical evidence revealed in the first part of our study, the second phase focuses on a qualitative analysis, where we manually examine a portion of our sample of evolutionary commits. The objective is to identify which of these commits consist exclusively of refactorings and which also include other modifications. By analyzing the content of these evolutionary commits, we gain a more in-depth perspective to determine whether the conflict may have been caused by refactoring or other external factors. The analysis covered 535 evolutionary commits, and approximately 84% of them were identified as "floss refactoring," meaning they include elements beyond refactorings, such as bug fixes, configuration commits, addition of tests, among others. Our study regarding the collection of "floss" and "pure refactoring" was an initial experiment, given the significant lack of tools that delve more deeply into the influence of refactoring in JavaScript code. Therefore, our study does not evaluate whether refactoring preserves or alters the code's behavior; instead, it was conducted in a way that involves manual analysis of the presence of refactorings or other modifications. Despite being an experimental study, conducted manually and susceptible to human errors, we identified that a significant portion of our commits (84%) involved in conflicts is replete with other modifications. It is crucial to assess the commits identified

as "pure refactoring" and, from that, determine whether they were indeed the causes of the conflicts. We emphasize that a relevant discovery of our study was the presence of many build files and configurations involved in conflicts in JavaScript projects, and many of them are present in commits classified as "floss refactoring." This highlights the importance of studies investigating the relationship between other modifications performed in conjunction with refactorings and the identified conflicts.

## 6.1 Future work

This work represents a significant starting point in the discussion of the relationship between merge conflicts and refactoring actions in JavaScript code, showing a positive correlation between these variables. It is the first study to analyze this interaction with Javascript code, contributing to the initial understanding of this dynamic. It is crucial to acknowledge that the results found in this research cannot be generalized due to the nature of the statistical analysis conducted on a specific sample. Therefore, we emphasize the need for future studies to refine this initial discussion. Subsequent research can use the methodology of this study as a foundation and expand to more comprehensive samples.

During the data collection phase regarding refactorings in conflicts, we identified a challenge related to the lack of specific tools and methodological discussions for the JavaScript language. This underscores the importance of studies that refine techniques already employed in other languages and propose new approaches for tools supporting these analyses. A valuable direction for future research would be the development of more advanced refactoring collection tools capable of detecting a broader range of refactoring types, leading to more robust results.

Furthermore, a pertinent aspect for future investigations would be to extend the discussion on merge conflicts beyond the traditional three-way merge, analyzing the perspective of merge strategies and the number of conflicts generated.

For future work, it is crucial to have dedicated automated tools for the analysis of "floss" and "pure refactoring" in JavaScript code. This advancement in tools would provide a more comprehensive insight into the true influence of refactoring on merge conflicts, contributing to the knowledge base in this research area.

# Bibliography

- [1] Paola Accioly, Paulo Borba, and Guilherme Cavalcanti. Understanding semi-structured merge conflict characteristics in open-source java projects. *Empirical Software Engineering*, 23:2051–2085, 2018.
- [2] Iftekhar Ahmed, Caius Brindescu, Umme Ayda Mannan, Carlos Jensen, and Anita Sarma. An empirical examination of the relationship between code smells and merge conflicts. In *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 58–67. IEEE, 2017.
- [3] Sven Apel, Jörg Liebig, Benjamin Brandl, Christian Lengauer, and Christian Kästner. Semistructured merge: rethinking merge in revision control systems. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 190–200, 2011.
- [4] Aryclenio Xavier Barros and Eiji Adachi. Bad smells in javascript-a mapping study. In *Anais do IX Workshop de Visualização, Evolução e Manutenção de Software*, pages 1–5. SBC, 2021.
- [5] Valerio Cosentino, Javier L Cánovas Izquierdo, and Jordi Cabot. A systematic mapping study of software development with github. *Ieee access*, 5:7173–7192, 2017.
- [6] Rafael de Souza Santos and Leonardo Gresta Paulino Murta. Evaluating the branch merging effort in version control systems. In *2012 26th Brazilian Symposium on Software Engineering*, pages 151–160. IEEE, 2012.
- [7] Asger Feldthaus, Todd Millstein, Anders Møller, Max Schäfer, and Frank Tip. Tool-supported refactoring for javascript. In *Proceedings of the 2011 ACM international con-*



- ference on Object oriented programming systems languages and applications*, pages 119–138, 2011.
- [8] Asger Feldthaus and Anders Møller. Semi-automatic rename refactoring for javascript. *ACM SIGPLAN Notices*, 48(10):323–338, 2013.
- [9] David Flanagan and Gregor M Novak. *Java-script: The definitive guide*, 1998.
- [10] Martin Fowler and Kent Beck. Refactoring: Improving the design of existing code. In *11th European Conference. Jyväskylä, Finland, 1997*.
- [11] Robert Fuhrer, Frank Tip, Adam Kiežun, Julian Dolby, and Markus Keller. Efficiently refactoring java applications to use generic libraries. In *ECOOP 2005-Object-Oriented Programming: 19th European Conference, Glasgow, UK, July 25-29, 2005. Proceedings 19*, pages 71–96. Springer, 2005.
- [12] Alejandra Garrido and José Meseguer. Formal specification and verification of java refactorings. In *2006 Sixth IEEE International Workshop on Source Code Analysis and Manipulation*, pages 165–174. IEEE, 2006.
- [13] Gleiph Ghiotto, Leonardo Murta, and Marcio Barros. A caminho de uma abordagem baseada em buscas para minimização de conflitos de merge. In *IV Workshop em Engenharia de Software baseada em Buscas*.
- [14] Gleiph Ghiotto, Leonardo Murta, Márcio Barros, and Andre Van Der Hoek. On the nature of merge conflicts: a study of 2,731 open source java projects hosted by github. *IEEE Transactions on Software Engineering*, 46(8):892–915, 2018.
- [15] David Johannes, Foutse Khomh, and Giuliano Antoniol. A large-scale empirical study of code smells in javascript projects. *Software Quality Journal*, 27:1271–1314, 2019.
- [16] Meir M Lehman, Juan F Ramil, Paul D Wernick, Dewayne E Perry, and Wladyslaw M Turski. Metrics and laws of software evolution-the nineties view. In *Proceedings Fourth International Software Metrics Symposium*, pages 20–32. IEEE, 1997.
- [17] Mehran Mahmoudi, Sarah Nadi, and Nikolaos Tsantalis. Are refactorings to blame? an empirical study of refactorings in merge conflicts. In *2019 IEEE 26th International*

- Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 151–162. IEEE, 2019.
- [18] Tom Mens. A state-of-the-art survey on software merging. *IEEE transactions on software engineering*, 28(5):449–462, 2002.
- [19] Michael Mohan and Des Greer. A survey of search-based refactoring for software maintenance. *Journal of Software Engineering Research and Development*, 6(1):1–52, 2018.
- [20] José Glauber Oliveira, Melina Mongiovi, and Sabrina Souto. An empirical study of the relationship between refactorings and merge conflicts in javascript code. In *Proceedings of the XXXVII Brazilian Symposium on Software Engineering*, pages 89–98, 2023.
- [21] William F Opdyke. *Refactoring object-oriented frameworks*. University of Illinois at Urbana-Champaign, 1992.
- [22] Achilleas Pipinellis. *GitHub essentials*, volume 2. Packt Publishing, 2015.
- [23] Max Schäfer and Oege De Moor. Specifying and implementing refactorings. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, pages 286–301, 2010.
- [24] Danilo Silva, Joao Paulo da Silva, Gustavo Santos, Ricardo Terra, and Marco Tulio Valente. Refdiff 2.0: A multi-language refactoring detection tool. *IEEE Transactions on Software Engineering*, 47(12):2786–2802, 2020.
- [25] Leonardo Sousa, Willian Oizumi, Alessandro Garcia, Anderson Oliveira, Diego Cedrim, and Carlos Lucena. When are smells indicators of architectural refactoring opportunities: A study of 50 software projects. In *Proceedings of the 28th International Conference on Program Comprehension*, pages 354–365, 2020.
- [26] Alberto Trindade Tavares, Paulo Borba, Guilherme Cavalcanti, and Sérgio Soares. Semistructured merge in javascript systems. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1014–1025. IEEE, 2019.

- 
- [27] Kristín Fjóra Tómasdóttir, Mauricio Aniche, and Arie Van Deursen. The adoption of javascript linters in practice: A case study on eslint. *IEEE Transactions on Software Engineering*, 46(8):863–891, 2018.
- [28] Gustavo Vale, Angelika Schmid, Alcemir Rodrigues Santos, Eduardo Santana De Almeida, and Sven Apel. On the relation between github communication activity and merge conflicts. *Empirical Software Engineering*, 25:402–433, 2020.
- [29] Bartosz Walter, Francesca Arcelli Fontana, and Vincenzo Ferme. Code smells and their collocations: A large-scale experiment on open-source systems. *Journal of Systems and Software*, 144:1–21, 2018.

# Appendix A

## Appendix of study

All the data, scripts, setup, and execution details pertaining to the three research questions addressed in this study are accessible in the following repository:

- Study Setup in GitHub: <[https://github.com/joseglauberbo/data\\_mestrado\\_dissertacao](https://github.com/joseglauberbo/data_mestrado_dissertacao)>