

Universidade Federal
de Campina Grande

UNIVERSIDADE FEDERAL DE CAMPINA GRANDE

CENTRO DE ENGENHARIA ELETRICA E INFORMÁTICA

UNIDADE ACADÊMICA DE SISTEMAS E COMPUTAÇÃO

COORDENAÇÃO DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

LUCIANA DE QUEIROZ LEAL GOMES

THE HIDDEN CHALLENGES OF MERGING: A Large-Scale Empirical Study Across Merge Tools

CAMPINA GRANDE – PB

2026

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

The Hidden Challenges of Merging: A Large-Scale Empirical Study Across Merge Tools

Luciana de Queiroz Leal Gomes

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

Área de Concentração: Ciência da Computação
Linha de Pesquisa: Evolução e Manutenção de Software

Melina Mongiovi e Sabrina Souto
(Orientadoras)

Campina Grande, Paraíba, Brasil
©Luciana de Queiroz Leal Gomes, 12/05/2026

Universidade Federal de Campina Grande - UFCG
Sistema de Bibliotecas - SISTEMOTECA
Catalogação de Publicação na Fonte. UFCG - Biblioteca Central

G633h Gomes, Luciana de Queiroz Leal.
The hidden challenges of merging: a large-scale empirical study across merge tools / Luciana de Queiroz Leal Gomes. – 2026.
109 f. : il. color.

Tese (doutorado em Ciência da Computação) – Universidade Federal de Campina Grande, Centro de Engenharia Elétrica e Informática, 2026.
"Orientação: Profa. Dra. Melina Mongiovi, Profa. Dra. Sabrina Souto".
Referências.

1. Merge Conflicts. 2. Merge Tools. 3. Refactoring. 4. Refactoring Tactic. 5. Mining Software Repositorie. I. Mongiovi, Melina. II. Souto, Sabrina. III. Título.

UFCG/BC CDU 004.41(043.2)

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

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: secpg@computacao.ufcg.edu.br

FOLHA DE ASSINATURA PARA TESES E DISSERTAÇÕES

LUCIANA DE QUEIROZ LEAL GOMES

THE HIDDEN CHALLENGES OF MERGING: A LARGE-SCALE EMPIRICAL STUDY ACROSS MERGE TOOLS

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

Aprovada em: 12/05/2026.

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. **Tiago Lima Massoni,** UFCG, Examinador Interno

Prof. Dr. **Ivan do Carmo Machado**, UFBA, Examinador Externo

Profa. Dra. **Paola Rodrigues de Godoy Accioly**, UFPE, Examinadora Externa

Documento assinado eletronicamente por **Ivan do Carmo Machado**, **Usuário Externo**, em 14/05/2026, às 09:12, conforme horário oficial de Brasília, com fundamento no art. 8º, caput, da Portaria SEI nº 002, de 25 de outubro de 2018.

Documento assinado eletronicamente por **MELINA MONGIOVI CUNHA LIMA SABINO**, **PROFESSOR(A) DO MAGISTERIO SUPERIOR**, em 14/05/2026, às 10:21, 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 **Paola Rodrigues de Godoy Accioly**, **Usuário Externo**, em 14/05/2026, às 13:17, 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 **TIAGO LIMA MASSONI**, **PROFESSOR(A) DO MAGISTERIO SUPERIOR**, em 14/05/2026, às 16:09, 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 15/05/2026, às 10:46, 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 15/05/2026, às 15:40, 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 **6512751** e o código CRC **56A3660A**.

Referência: Processo nº 23096.032452/2026-81 SEI nº 6512751

Resumo

O desenvolvimento de software evoluiu de abordagens centralizadas para colaborativas, com Sistemas de Controle de Versão (VCSs), como o Git, facilitando a integração por meio de operações de mesclagem. No entanto, conflitos de mesclagem que surgem durante a integração de código podem atrasar projetos e reduzir a produtividade. Este estudo investiga fatores que influenciam tais conflitos analisando os resultados de três ferramentas de mesclagem: Git, jFSTMerge e IntelliMerge. O Git é uma ferramenta baseada em texto amplamente utilizada, enquanto o jFSTMerge e o IntelliMerge são ferramentas que mesclam arquivos escritos em linguagem Java, projetadas para lidar com cenários mais complexos. Analisamos mais de 507.000 cenários de mesclagem de 530 projetos Java hospedados no GitHub, considerando cinco fatores: o número de commits evolutivos (os commits com alterações que levam a um conflito), o número de desenvolvedores, o número de refatorações, o número de regiões de refatoração e o número de refatorações nos ramos (pais) esquerdo e direito da mesclagem. Embora a refatoração vise melhorar a qualidade do código, ela também pode intensificar esses conflitos. Encontramos evidências de que cerca de 43% dos cenários com conflito no Git, 32% no jFSTMerge e 37% no IntelliMerge podem estar relacionados a refatorações. Além disso, os cenários com conflito no Git que envolviam refatorações também estavam associados a tempos de integração mais longos. As refatorações apresentaram uma forte associação com conflitos de mesclagem em todas as ferramentas. Após controlar variáveis de magnitude de mudança, como *code churn* e o número de arquivos modificados, as refatorações permanecem fortemente associadas a conflitos de mesclagem, aumentando a probabilidade de conflito para cerca de 58% em nosso conjunto de dados do Git. Entre os 62 tipos de refatoração identificados, um subconjunto consistente (33,8%), incluindo renomeação, movimentação de classes e extração de métodos, aparece frequentemente em conflitos em todas as ferramentas. Também classificamos as refatorações por tática, como puras ou floss, constatando que 65% dos cenários com conflito envolvem refatorações do tipo floss e que conflitos envolvendo essa tática de refatoração tendem a exigir tempos de resolução ligeiramente maiores. De modo geral, nossos resultados sugerem que a refatoração não é apenas uma prática voltada para a melhoria da qualidade do código, mas também um fator associado à dinâmica do processo de mesclagem. Essas descobertas destacam a importância de

planejar e coordenar cuidadosamente as refatorações. Desenvolvedores podem utilizar essas observações para definir diretrizes de mesclagem, enquanto pesquisadores podem basear-se nelas para projetar ou aprimorar ferramentas de mesclagem mais bem equipadas para lidar com alterações relacionadas à refatoração.

Palavras-chave: Conflitos de Mesclagem; Ferramentas de Mesclagem; Refactoring; Tática de Refactoring; Mineração de Repositórios de Software.

Abstract

Software development has shifted from centralized to collaborative approaches, with Version Control Systems (VCSs), such as Git, facilitating integration through merge operations. However, merge conflicts that arise during code integration may delay projects and reduce productivity. This study investigates factors influencing merge conflicts by analyzing merge outcomes from three tools: Git, jFSTMerge, and IntelliMerge. Git is a widely used text-based tool, whereas jFSTMerge and IntelliMerge are state-of-the-art Java merge tools designed to handle more complex scenarios. We analyzed over 507,000 merge scenarios from 530 Java projects on GitHub, considering five factors: the number of evolutionary commits (the commits with changes that lead to a merge conflict), the number of developers, the number of refactorings, the number of refactoring regions, and the number of refactorings in the left and right merge parents. While refactoring aims to improve code quality, it may also intensify these conflicts. We found evidence that nearly 43% of Git, 32% of jFSTMerge, and 37% of IntelliMerge conflicting scenarios may be related to refactorings. Additionally, Git's conflicting scenarios involving refactorings were also associated with longer resolution times. Refactorings show a strong association with merge conflicts across all tools. After controlling for change-magnitude variables, such as code churn and the number of modified files, refactorings remain strongly associated with merge conflicts. The presence of refactorings also increased the likelihood of a conflict to about 58% in our Git dataset. Among the 62 refactoring types identified, a consistent subset (33.8%), including renaming, moving classes, and extracting methods, frequently appears in conflicts across all tools. We also classify refactorings by tactic as pure or floss, finding that 65% of conflicting scenarios involve floss refactorings. Additionally, conflicts involving floss refactorings tend to require slightly longer resolution times. Overall, our results suggest that refactoring is not only a practice aimed at improving code quality, but also a factor associated with the dynamics of the merging process. These findings highlight the importance of carefully planning and coordinating refactorings. Developers may use these observations to define merging guidelines, while researchers may build on them to design or improve merge tools better equipped to handle refactoring-related changes.

Keywords: Merge Conflicts; Merge Tools; Refactoring; Refactoring Tactic; Mining Software Repositories.

Agradecimentos

Agradeço a Deus pelas conquistas alcançadas e pela oportunidade de chegar até este momento.

Agradeço às minhas orientadoras, cuja orientação, dedicação e rigor científico foram fundamentais para o desenvolvimento desta pesquisa. É ao subir nos ombros de gigantes que se torna possível ampliar horizontes.

Agradeço à minha família , meus pais e irmãs, pelo apoio incondicional ao longo de toda a minha trajetória.

Agradeço aos meus filhos e ao meu esposo, pelo suporte constante, pela compreensão e pela paciência diante das minhas ausências.

Agradeço à banca avaliadora pelas valiosas contribuições e considerações, que enriqueceram significativamente este trabalho.

Agradeço à UEPB, instituição de ensino que faço parte, pela concessão do meu afastamento parcial, possibilitando a realização desta pesquisa.

Por fim, agradeço a todos que, direta ou indiretamente, contribuíram para a concretização deste trabalho.

Contents

1 Introduction **1**

1.1 Problem 2

1.2 Motivating Example 4

1.3 An Empirical Study of Refactorings and Merge Conflicts Across Merge Tools 7

1.4 Summary of Contributions 10

1.5 Implications of this Research 11

1.5.1 For Developers 11

1.5.2 For Tool Builders 12

1.5.3 For Researchers 13

1.6 Structure 14

2 Background **15**

2.1 Code Merging 15

2.1.1 Merge Conflicts and Evolutionary Commits 17

2.1.2 Merge Approaches 18

2.1.3 Merge Tools 21

2.1.4 Research in Software Merging 25

2.2 Refactoring 26

2.2.1 Pure and Floss Refactoring 27

2.2.2 Refactoring Miner Tool 29

2.2.3 Refactorings and Merge Tools 32

3 Investigating Factors Related to Merge Conflicts: A Tool-Supported Study **35**

3.1 Research Goals and Questions 35

3.2 Study Setup . . . 37
3.2.1 Subjects Description and Characterization . . . 37
3.2.2 Collecting Historical Data on Subjects . . . 38
3.2.3 Merge Tools Selection . . . 38
3.3 Adopted Approach . . . 39
3.3.1 Step 1 – Collecting Data . . . 40
3.3.2 Step 2 – Merge Tools Execution and Environment . . . 40
3.3.3 Step 3 – Filtering Collected Data . . . 41
3.3.4 Step 4 – Automated Analysis . . . 42
3.4 Approach for Data Analysis . . . 42
3.4.1 Selected Variables . . . 42
3.4.2 Statistical Analysis . . . 43
3.4.3 Time to Merge Conflicts with Refactoring Involved . . . 45
3.5 Results . . . 45
3.5.1 **RQ.1** - What factors are associated with the occurrence of merge conflicts? . . . 47
3.5.2 **RQ.2** - Which types of refactoring are most associated with merge conflicts? . . . 54
3.6 Discussion . . . 55
3.7 Threats to Validity . . . 58
3.7.1 Construct Validity . . . 59
3.7.2 Internal Validity . . . 60
3.7.3 External Validity . . . 60
3.7.4 Statistical Conclusion Validity . . . 60
3.8 Concluding Remarks . . . 61

4 Towards a Deeper Understanding of Refactorings and Merge Conflicts 62
4.1 Study Goal and Research Questions . . . 63
4.2 Approach to Analyze Refactoring Impacts on Merge Conflicts . . . 64
4.2.1 Variables . . . 65
4.2.2 Statistical Analysis . . . 66

4.2.3 Analyzing Pure/Floss Refactoring in Merge Conflicts 68
4.2.4 Metrics Related to Change Magnitude 71
4.2.5 Metric Related to the Purity of Refactorings in a Conflicting Merge Scenario 72
4.3 Results 72
4.3.1 **RQ.3** - Is refactoring independently associated with merge conflicts? 73
4.3.2 **RQ.4** - Does the association between refactoring and merge conflicts vary across merge tools? 76
4.3.3 **RQ.5** - Which refactoring tactic, pure or floss, is more associated with merge conflicts? 77
4.3.4 **RQ.6** - How are merge conflicts distributed across refactoring tactics (pure vs. floss), considering the types of refactorings involved? . . . 81
4.3.5 **RQ.7** - Do merge scenarios involving only pure refactorings differ in resolution time from those with floss involvement? 83
4.4 Discussion 84
4.5 Threats to Validity 87
4.5.1 Construct Validity 87
4.5.2 Internal Validity 88
4.5.3 External Validity 88
4.6 Concluding Remarks 89

5 Related Work **90**
5.1 Factors and Indicators Related to Merge Conflicts 90
5.2 Comparison and Evaluation of Merge Tools 91
5.3 Impact of Refactorings on Merge Conflicts 93

6 Conclusions **95**
6.1 Future Work 97
6.1.1 AI Application in Repositories 98

List of Figures

1.1 Merge Conflict in an Academic Example 5
1.2 Possible manual resolution of the merge conflict 6

2.1 Commits in a three-way merge scenario 17
2.2 A Conflicting Scenario Example 18
2.3 A jFSTMerge conflicting merge scenario 22
2.4 A IntelliMerge conflicting merge scenario 24
2.5 Example of Refactoring Miner Operation 30
2.6 Purity Checker execution simplified 32

3.1 Adopted Approach 39
3.2 Detailing Step 1 40
3.3 Detailing Step 2 41
3.4 Detailing Step 3 41
3.5 Merge conflict reported by all three tools 57
3.6 Time to merge conflicting scenarios with and without refactorings involved (median values) 58

4.1 Steps to analyze refactorings 65
4.2 Steps applied in manual analysis 69
4.3 Analyzed elements in a merge scenario 69
4.4 Floss Ratio Results 78
4.5 Extract Method from an Evolutionary Commit on Catroid project 80
4.6 Push Down Method from an Evolutionary Commit on DroidPlanner project 80
4.7 Extract Method from an Evolutionary Commit on cassandra project 81

4.8 Occurrences of Refactoring Types and the Tactic Applied 83

List of Tables

3.1 Variables collected in this study 36
3.2 Projects data 38
3.3 Summary of the merge outcomes from the tools 46
3.4 Distribution of Conflicts per project 46
3.5 Variables results for each tool 47
3.6 Odds Ratios from regression considering conflict factors for each tool (***p<0.001; **p<0.01; *p<0.05) 48
3.7 Logistic regression model specifications. 49
3.8 Model VI Detailed 50
3.9 Model VII Detailed 51
3.10 Model VIII Detailed 52
3.11 Top ten refactoring types involved in merge conflicts by tool 55

4.1 Variables applied in this study 66
4.2 Contingency table 74
4.3 Distribution of LOC churn and number of files changed by merge conflict occurrence 75
4.4 Contingency table for jFSTMerge results 76
4.5 Contingency table for IntelliMerge results 76
4.6 Pure and Floss Refactorings by Refactoring Type Involved in Conflicts . . . 82

Chapter 1

Introduction

Collaborative software development has become increasingly common over the years, allowing multiple developers to work in parallel on the same project [13], [14], [68]. This practice enables teams to combine different skills and expertise, even when contributors are geographically distributed. As a result, software projects can benefit from improved coordination, faster development, and more effective achievement of project objectives.

Branch development and collaborative development environments offers advantages as they allows collaborators to share knowledge, comments, issues, and solutions, reflecting in knowledge distribution over the team [72]. However, during code synchronization, Version Control Systems (VCSs) may face merging conflicts when developers try to integrate different versions of the same code, and the detected divergences may prevent successful completion [52], [67], [49]. In general, merge conflicts are easy to resolve, but specific conflicts may introduce significant challenges to software development and be difficult to address [17], [39] . These challenging conflicts may cause delays in software development as developers attempt to understand and fix them, potentially affecting code quality [5], [16], [51]. When multiple commits are involved, and many conflicts arise, developers may find it challenging to fully comprehend the situation [14], [50], [51]. In such cases, productivity can be affected because of the increased time required to resolve issues [69], [76], [82].

Merge conflicts account for 10% to 56.5% of all merges according to previous research [5], [19], [24], [31], [39], [44]. Several studies have analyzed this topic, aiming to minimize conflicts by developing merge tools [7], [8], [24], [33], [73], identifying their causes [17], [24], [49], [51], [59], [82], investigating contributing factors [3], [31], [47], [59], [65], and

comparing existing solutions [70], [73], [74].

Merge conflicts have increasingly been recognized as a problem, with different possible solutions depending on the scenario being merged [74]. In practice, developers often rely on unstructured merge strategies, which use a textual approach to detect line changes [52]. Even though it handles different file types quickly, this agnostic approach may lead to more merge conflicts than expected [2], [8], [23]. In such cases, merge results may include spurious conflicts, false positives incorrectly detected due to formatting or ordering changes, which could be avoided with more refined techniques [2], [26].

Tool builders and researchers have developed alternatives to textual merging to mitigate these issues. Syntactic merge tools, which leverage parse trees to detect syntactically incorrect code fragments, or semantic merge tools, which employ graph-based data structures to analyze semantic changes, are the two bigger categories of merging tools [2], [8], [21], [23], [24], [73]. Understanding the circumstances that lead to merge conflicts, even when applying these improved tools, is essential to developing more efficient solutions.

1.1 Problem

Considering developers' activities, previous research has emphasized the search for minimizing merge conflicts and reinforced the difficulty in resolving a portion of them, suggesting that dvelopers tend to avoid them as much as possible [11], [39], [73], [81], [82]. In this light, some studies have analyzed the factors behind merge conflicts, observing challenges in resolving them, how technical and organizational factors affect merge conflicts, and the programming language constructs involved in conflicts [31], [39], [81], [82]. Although previous studies have investigated factors such as project size, number of developers, and evolutionary commits, the underlying causes of merge conflicts remain incompletely understood [3], [31], [50], [82]. In particular, it remains unclear how different types of changes affect the likelihood of conflicts during code integration.

As common as code merging is in software development, refactorings are a particular type of modification in code that involves reviews of the same code. These internal restructuring operations are intended to preserve the software's external behavior [37], [64]. Developers frequently apply refactorings to reduce complexity, enhance code quality, and

facilitate software evolution [1], [10], [30], [37], [53], [64], [75].

While refactorings are generally seen as beneficial to the software [9], they may also increase the likelihood of merge conflicts [32], [33], [46], [49], [50], [82]. This side effect may occur because refactorings often follow structural patterns that do not align well during merges [63], [66]. Additionally, refactorings may contribute to conflict complexity [32], [50], [73], along with factors such as the number and complexity of conflicting lines [17], [50], [51], [59], [82], the expertise in the conflict's technical area [51], and the number of involved developers [50], [82].

Refactorings are challenging for merge tools to handle [70]. An important finding in this field is that refactorings account for at least 22% of merge conflicts in Git [50]. However, the specific conditions under which these conflicts arise, as well as a more comprehensive analysis of refactorings and other historical aspects related to conflicts, remain underexplored. Additionally, it is unclear whether findings based on Git from various studies can be generalized to other merge tools.

Moreover, refactoring activities can still vary depending on how developers perform them. Programmers may need to fix bugs or add features while refactoring code. When refactoring activities interleave code improvements, it is referred to as floss refactoring [55], [56]. In contrast, when developers perform only refactoring transformations, it is known as pure (or root-canal) refactoring. Among these two tactics, floss refactoring is the most commonly applied [4], [56], [75].

Furthermore, the role of different refactoring tactics, such as consistent, behavior-preserving structural changes (pure refactorings) versus those involving overlapping edits (floss refactorings) [55], [56], in the emergence of conflicts has not yet been investigated. Analyzing pure and floss refactorings in this context may reveal the nature of conflicting changes and support the development of more targeted resolution strategies. To address these open questions, we conduct two complementary studies. The first study presented in Chapter 3 examines factors related to merge conflicts, and the second study in Chapter 4 focuses on refactorings as a predominant factor influencing their occurrence.

1.2 Motivating Example

In Figure 1.1, we present a didactic example in which developers attempt to integrate their contributions on a Git-based Version Control System. The example models a simplified payroll system for calculating employees' salaries within a company. In the Base branch, the `Employee` class computes taxes and prints salary information. Two developers create the Left and Right branches to improve the `Employee` class.

In the Left branch, the developer implemented the salary calculation by introducing a conditional expression to compute taxes, followed by a renaming transformation. The method `print` is renamed to `getEmployeeInfo`, and the tax computation logic is included in this method with a conditional rule that applies different tax rates to employees with lower salaries. In the Right branch, the developer introduces structural changes by extracting the tax computation into a new class named `TaxCalculator`, renaming the method `print` to `toString`, and delegating both salary and tax calculations to this new class.

When these contributions are merged, a conflict arises from edits within the same methods and differing rename operations. The resulting merged file contains conflicts in renamed methods and inconsistencies caused by additional edits on both sides. Since the modifications diverge across the three versions, the conflict cannot be automatically resolved by a textual merge tool such as Git.

In this scenario, the developers' decisions lead to a merge conflict. In the Left branch, the developer renamed a method to improve readability, whereas in the Right branch, the developer implemented a new class to include the tax computation logic. These concurrent changes result in a semantic conflict. Additionally, dependencies among the modified components can further increase both the impact and the complexity of conflict resolution.

Even syntactic or semantic merge tools may be unable to integrate the changes in this example. Multiple edits occur in parallel, making it difficult to determine which version is more appropriate. Each branch applies an incompatible rename operation to the same method while also modifying its internal logic. As a result, a merge tool cannot infer the most suitable version without potentially violating the contributions of at least one developer. Conflicts of this nature emphasize the need for tools that alert developers when their



Base

```java
public class Employee {

    private float discounts;

    // getters and setters definition

    public float calcTaxes() {
        return discounts;
    }

     public void print() {
          System.out.println("Name: " + getName());
          System.out.println("Salary: " + (getSalary() - calcTaxes()));
     }

}
```

Left

```java
public class Employee {

    private float salary;

    // getters ans setters definition

    public void getEmployeeInfo() {
        float taxes = (salary <= 1000) ? 0 : 100;
        System.out.println("Name: " + getName());
        System.out.println("Salary: " + (getSalary() - taxes));
    }

}
```

Right

```java
public class Employee {

    private TaxCalculator calculator = new TaxCalculator();
    private float salary;

    // getters and setters definition

    public float calcSal() {
        return getSalary() - calculator.calcTaxes();
    }

    @Override
    public String toString() {
        return "Name: " + getName() + "\nSalary: " + calcSal();
    }

}
```

```java
public class TaxCalculator {

    private float discounts;

    public float calcTaxes() {
        return discounts;
    }
}
```

Merged

```java
public class Employee {

<<<<<<< HEAD
    private float salary;

    // getters ans setters definition

    public void getEmployeeInfo() {
        float taxes = (salary <= 1000) ? 0 : 100;
        System.out.println("Name: " + getName());
        System.out.println("Salary: " + (getSalary() - taxes));
    }
=======
    private TaxCalculator calculator = new TaxCalculator();
    private float salary;

    // getters ans setters definition

    public float calcSal() {
        return getSalary() - calculator.calcTaxes();
    }

    @Override
    public String toString() {
        return "Name: " + getName() + "\nSalary: " + calcSal();
    }
>>>>>>> right

}
```

Figure 1.1: Merge Conflict in an Academic Example

overlapping activities may lead to merge conflicts. Avoiding this type of conflict is not trivial. Merge tools with more specialized algorithms might attempt to detect change patterns in refactorings, but in this case, they would still struggle to determine the most appropriate resolution and would ultimately report a merge conflict.

A possible manual resolution to this conflict is to adopt the structural change and separation of concerns from the Right branch, while incorporating the business rule for tax calculation from the Left branch. This provides a cleaner and more maintainable solution. Figure 1.2 illustrates this resolution, highlighting the Left branch contributions in yellow. Although not the only possible approach, it illustrates that resolving such conflicts typically

requires manual intervention beyond the capabilities of merge tools.

```java
public class Employee {

    private TaxCalculator calculator = new TaxCalculator();
    private float salary;

    // getters and setters definition

    public float calcSal() {
        return getSalary() - calculator.calcTaxes(getSalary());
    }

    @Override
    public String toString() {
        return "Name: " + getName() + "\nSalary: " + calcSal();
    }

}
```

```java
public class TaxCalculator {

    private float discounts;

    public float calcTaxes(float salary) {
        discounts = (salary <= 1000) ? 0 : 100;
        return discounts;
    }
}
```

Figure 1.2: Possible manual resolution of the merge conflict

Prior work has investigated factors related to merge conflicts, ranging from localized code changes to broader technical and organizational aspects. For example, Accioly et al. [3] analyzed conflict predictors based on localized code changes, such as edits to the same method and to dependent methods, showing that these patterns are frequently associated with conflicts. Complementing this perspective, Dias, Borba, and Barreto [31] investigated technical and organizational factors, including contribution size, modularity, and timing, showing that larger and less modular contributions are more likely to be associated with merge conflicts. While these studies provide important insights into factors related to conflict occurrence, they primarily focus on structural properties of changes and characteristics of contributions. However, they do not explicitly isolate the role of refactorings from other change-related factors.

In this study, we investigate merge conflicts by analyzing merge scenarios extracted from GitHub repositories among different merge tools. During this analysis, refactorings were identified as a particularly relevant factor, motivating an in-depth examination of their role in conflict occurrence. Therefore, we analyze how refactorings are associated with merge conflicts while we apply variables related to change magnitude as control. By isolating the effect of refactorings from other factors, we aim to better understand their contribution to merge conflicts and provide evidence to support both conflict prevention and improvements in merge tools.

1.3 An Empirical Study of Refactorings and Merge Conflicts Across Merge Tools

When analyzing the background presented, several factors impact the occurrence of conflicts [3], [31], [74], [82]. In this study, we analyzed merge conflicts to better understand the role of these factors in conflict occurrence. During this investigation, we observed that refactorings frequently appeared in commits associated with conflicts. This pattern suggested that refactorings might play a distinctive role in how conflicts arise. Building on these findings, we explored the association between merge conflicts and refactoring to gain a better understanding of their relationship. To do so, we analyzed merge scenarios obtained using the three-way merge technique aiming to answer the following research questions:

- **RQ.1** What factors are associated with the occurrence of merge conflicts?
- **RQ.2** Which types of refactoring are most associated with merge conflicts?

As a first step, we examined five factors potentially associated with merge conflicts: the *Number of Evolutionary Commits*, the *Number of Developers*, the *Number of Refactorings*, the *Number of Refactoring Regions* observed in evolutionary commits, and *Refactorings in Parents* of a merge. These factors were selected because they capture aspects of code changes may be related to the occurrence of merge conflicts [50]. Specifically, the number of evolutionary commits and the number of developers reflect the intensity of parallel modifications in the codebase, which increases the likelihood of overlapping changes [31], [50], [81]. Refactorings, both in total number and across distinct regions, change the code's structure and can propagate changes that interact unexpectedly with other contributions. Additionally, refactorings in parent commits may introduce structural dependencies that affect the outcome of subsequent merges. Collectively, these metrics serve as measurable proxies for understanding how historical changes and structural modifications contribute to the occurrence of merge conflicts.

Different merge tools may report different conflict outcomes for the same integration scenario due to their strategies for merging execution. We conducted a preliminary manual inspection of five repositories which revealed disagreements among Git, jFSTMerge, and IntelliMerge regarding conflict detection and resolution. These observations suggested

that conclusions derived from a single merge tool might not generalize across alternative merge approaches, motivating the use of multiple tools in this study. We selected Git as the state-of-practice tool, which treats artifacts as plain text and performs textual merges. We also included jFSTMerge, a state-of-the-art semistructured merge tool, and IntelliMerge, a refactoring-aware semistructured merge tool. Together, these tools represent textual, semistructured, and refactoring-aware merging approaches, allowing us to assess our findings across different merge strategies.

To answer our research questions, we conducted automated analyses of merge approaches and refactoring intents. Considering the merge approaches analyses, we examined 169,980 commits from 600 GitHub Java projects, applying three merge tools to perform over 507,000 distinct merge actions. This provided a broad view of merge conflict scenarios. For each merged case, we collected data on conflicts, collaborators, commits, timestamps, and refactorings in the projects' histories. We then applied descriptive statistics and logistic regression models for each tool to evaluate the influence of these variables on merge conflicts.

The results of this first analysis indicate that the variables *Number of Evolutionary Commits* and *Number of Developers* were not statistically significant when analyzing merge conflicts detected by Git tool. However, these two variables showed statistically significant results for the jFSTMerge and IntelliMerge tools, particularly for the *Number of Developers*, which was identified as a relevant factor in the occurrence of merge conflicts across the two tools. We also identified that the granularity of changes associated with refactorings, as reflected in variables such as the *Number of Refactoring Regions* and the *Number of Refactorings in Evolutionary Commits*, is related to conflict occurrence.

Moreover, the *Number of Refactorings in Parents* is significantly related to merge conflicts. Refactorings in parent commits are linked to increased odds of merge conflicts occurrence. In this context, odds represent the ratio between the probability of a conflict occurring and the probability of it not occurring [78]. The estimated odds ratios for merge tool are of 9.89 (Git), 11.94 (jFSTMerge), and 8.84 (IntelliMerge).

Additionally, we found evidence that nearly 43% of Git, 32% of jFSTMerge, and 37% of IntelliMerge conflicting scenarios were related to refactorings. Despite identifying 62 different refactoring types in evolutionary commits, only 21 were involved in merge conflicts. Ten of them consistently appear in results considering Git, jFSTMerge, and IntelliMerge

tools: Rename Parameter, Rename Method, Rename Variable, Move Class, Extract Method, Move Method, Rename Attribute, Move Attribute, Pull Up Method, and Extract Variable.

Conflicts involving refactorings took longer to merge than those without refactorings, despite occurring less frequently. Specifically, scenarios merged using Git that involved refactoring operations took a median of 16.7 minutes to merge, compared to 12.2 minutes for those without refactorings. A Mann-Whitney U test confirmed that the difference is statistically significant, with a medium effect size, suggesting that the presence of refactorings in an evolutionary commit is associated with the time to merge conflicting contributions ($U - statistic = 8,6494,520.0$, $p - value < 0.01$, $effect - size = 0.35$).

Based on our initial findings from answering RQ.1, RQ.2, and RQ.3, which collectively indicate that refactorings are consistently involved in merge conflicts, we deepen this investigation in two parts. First, since the initial results cannot definitively distinguish whether conflicts are driven by refactoring transformations themselves or by the large scope of changes often associated with them, we introduce control variables related to the magnitude of changes and examine the interaction of this association across the three merge tools investigated. Second, we performed a large-scale classification of refactorings entangled in conflicts as pure or floss, and analyzed the distribution and time-to-merge related to each tactic.

Considering this, the following additional questions aim to explore the influence of refactorings on merge conflicts. We deepen this analysis and perform an extensive automated classification of refactorings' tactics, using evolutionary commits as a data source, and we refine RQ.3, resulting in RQ.7. We posed the following research questions to investigate this nuance:

- **RQ.3** Is refactoring independently associated with merge conflicts?
- **RQ.4** Does the association between refactoring and merge conflicts vary across merge tools?
- **RQ.5** Which refactoring tactic, pure or floss, is more associated with merge conflicts?
- **RQ.6** How are merge conflicts distributed across refactoring tactics (pure vs. floss), considering the types of refactorings involved?

- **RQ.7** Do pure and floss refactorings differ in merge conflict resolution time?

By accounting for confounding variables in the regression model, the results indicated that the strong association between refactorings and merge conflicts persists. According to the interaction model results, this association varies across tools, but consistently confirms the strong association observed in our previous analysis (OR = 24.13, 95% CI [22.96, 25.35]). Git results for the presence of refactorings (OR=24.10, 95% CI [22,91, 25,29]) are followed by jFSTMerge (OR=15.5, 95% CI [13.8, 17.5) and IntelliMerge (OR=9.5, 95% CI [8.3, 10.9]), with the control variables showing a small effect size.

Regarding the relationship between refactoring tactics and merge conflicts, we filtered our dataset to include only evolutionary commits containing refactorings. We automatically classified commits by refactoring tactic, and then the merge conflict containing that refactoring was classified by pure or floss. According to our results, 65% of merge-conflicting scenarios involving refactoring contained floss refactorings.

Finally, we investigated the time to merge commits involving pure and floss refactorings. The median merge time for contributions involving pure refactorings was approximately 62 minutes, compared to 63 minutes for those involving floss refactorings, indicating no statistically significant difference, as determined by the Mann-Whitney test ($U = 1,005,558.5, p_value < 0.22$).

1.4 Summary of Contributions

In summary, the main contributions of our work are [41]:

- This study is among the first to investigate factors associated with merge conflicts using multiple merge tools with different merging techniques. To support this investigation, we constructed a comprehensive dataset comprising 169,980 merge commits per tool across 530 projects. We also collected the corresponding merge parents, historical commits, and the results produced by the three analyzed merge tools. The complete dataset has been made publicly available for replication and further research [1];

[1] https://github.com/refactoringresearch/saner_2025

- Approximately one-third of merge conflicts are associated with refactorings. We quantified the odds increase when refactorings are involved, which is approximately 10 times higher in Git, 12 times in jFSTMerge, and 9 times in IntelliMerge compared to cases without refactorings. These results suggest an association of the structural effects of code changes on the effort required for conflict merging. In addition, we identified 21 refactoring types common across the three tools, including 10 particularly frequent ones associated with merge conflicts across results from the three merge tools, out of a total of 62 refactoring types detected in the commits;

- Refactoring is an independent risk factor, even when confounding factors related to the magnitude of change are controlled for. Git alone presented odds approximately 24 times higher (OR = 24.10) of merge conflicts when refactorings are present.

- The effect of refactoring is tool-independent. The interaction model results showed that the association varies across the three tools but consistently indicates a strong relationship between refactorings and merge conflicts, even after controlling for size changes.

- To the best of our knowledge, this is the first study investigating the influence of pure and floss refactoring tactics in Java projects on merge conflicts. We found that 65% of merge-conflicting scenarios involved floss refactorings. However, we observed no statistically significant difference between pure and floss refactorings in terms of time to merge.

1.5 Implications of this Research

1.5.1 For Developers

Our findings suggest that the number of developers and the presence of refactorings in merge parents may be significant factors contributing to merge conflicts. Developing guidelines that take into account developers' coordination activities, particularly when refactorings are involved, may help minimize these conflicts.

Additionally, developers should pay close attention in branches with several successive

commits. By reducing the number of accumulated changes before a merge and communicating planned refactorings early, teams can decrease merge complexity and prevent error-prone scenarios.

Developers should also be aware that pure refactorings may increase the cognitive effort required to integrate conflicting code. Carefully planning refactorings, isolating them in dedicated commits, or coordinating their integration with teammates can help reduce conflicts in these cases.

Furthermore, the number of merge conflicts associated with floss refactorings is higher, indicating that this tactic may contribute to merge-related issues. Strategies to avoid conflicts should therefore prioritize planning and visibility around refactorings to reduce the likelihood of floss refactoring. While this tactic may align with agile development practices, teams should remain vigilant to ensure that code quality and clarity are not compromised.

1.5.2 For Tool Builders

Merge tools could leverage our findings to alert development teams when multiple contributors are working simultaneously, especially in the presence of refactorings, by recommending additional caution or coordination to prevent potential conflicts. These tools could also refine their conflict detection algorithms to account for the higher risk of conflicts in projects with increased developer involvement. In such scenarios, they might prioritize conflict identification and offer more intelligent merge strategies that better track individual changes. If integrated into developers' workflows, these insights could make merge tools more effective in managing complex merges involving multiple contributors, ultimately supporting a more refined merge process.

Our study also highlights opportunities to enhance conflict resolution strategies. Since the analyzed tools (Git, jFSTMerge, and IntelliMerge) struggle to varying degrees with conflicts involving refactorings, improving refactoring-aware heuristics could lead to more accurate conflict detection and resolution. Tools could also benefit from incorporating findings related to evolutionary commits, particularly those that include structural or functional changes. Merge tools and version control systems could issue warnings when a branch accumulates many changes prior to integration. Tool developers might also implement merge previews or simulations that visualize potential risks based on commit history and change

types, helping developers anticipate and avoid problematic merges. Additionally, tracking the history of refactorings and the number of involved developers could provide a more precise understanding of collaboration patterns and potential sources of conflict.

Furthermore, version control systems and merge tools could incorporate heuristics to detect pure refactorings and indicate the potential for complex conflict resolution in such cases. Automatically classifying refactorings by tactic, such as pure or floss, would enable proactive conflict management, allowing tools to adapt their merge strategies to the complexity and nature of the development context.

1.5.3 For Researchers

Our findings offer several opportunities for future research. From a quantitative perspective, the consistent association between refactorings and merge conflicts across different merge strategies suggests that these factors should be further investigated in other programming languages and development environments. The dataset produced in this study may also support future research on conflict prediction models and related techniques.

From a qualitative perspective, different data collection approaches could be employed, such as interviews with developers and tool builders or the analysis of pull request discussions. These approaches may help researchers understand how developers perceive refactoring-related conflicts and how they mitigate merge conflicts in collaborative development settings. Social and organizational factors explored through this type of research may reveal information that cannot be captured through repository mining alone.

Regarding tool development, our findings suggest that merge conflicts should be viewed as a phenomenon influenced by both structural and temporal dimensions. The strong relationship between refactorings and conflicts indicates that future merge tools should better account for concurrent modifications, structural code changes, and the ways developers perform these changes. This observation also opens opportunities for future studies investigating the interaction between software evolution, developer coordination, and conflict emergence.

The factors identified in this study may support the construction of benchmark datasets and evaluation frameworks for merge tools, conflict prediction techniques, and refactoring-aware approaches, improving comparability, reproducibility, and external validity across future research.

We also emphasize the relevance of distinguishing between pure and floss refactorings, as they impact conflicts differently. This distinction can support cognitive models of developer behavior, conflict prediction strategies, and the creation of labeled benchmarks for tool evaluation and reproducibility.

Finally, to reduce conflicts related to refactorings, future work should explore refactoring-aware algorithms, commit history monitoring, and the role of refactoring tactics in conflict occurrence. These understandings can drive the evolution of more effective merge tools and practices for collaborative development.

1.6 Structure

This dissertation is organized as follows: this Chapter presents an introduction to our work. In Chapter 2, we provide background for the study explaining merge conflict definition, merge approaches and tools, and refactoring-related concepts. Chapter 3 presents results related to Git, jFSTMerge, and IntelliMerge tool exploration, analyzing conflicts originated from these three tools; we described factors explored in the context of merge conflicts, and analyzed the results statistically. In Chapter 4, we examined the refactoring in more depth by fitting models controlling for magnitude changes, analyzed the interaction of merge tools considering refactorings in a merge scenario, and classified pure and floss tactics and their association with merge conflicts. Finally, Chapter 5 presents the works related to this research, and Chapter 6 summarizes the results and contributions of this work and suggests future works.

Chapter 2

Background

In this Chapter, we explore the foundational concepts necessary for understanding the topics discussed in this work. We begin by defining software merging and the approaches of merge tools in three-way merge scenarios in Section 2.1. Next, we discuss evolutionary commits in Section 2.1.1, which provide a broader context for analyzing merge conflicts. We also present the concept of refactorings in evolutionary commits and how to locate them in code. We presented how research in software merging evolved in Section 2.1.4 .

In Section 2.2, we present how refactoring relates to merge conflicts (2.2.3), the concepts of pure and floss refactoring (2.2.1), and finally, the PurityChecker API developed to detect purity on commits in a software project (2.2.2).

2.1 Code Merging

Parallel software development is a practice that enables programmers to collaboratively work on large-scale projects [67]. In such environments, version control tools play a pivotal role in supporting coordination activities, particularly during software merging. In this process, consecutive changes may occur across different branches. For instance, two or more developers might edit the same line of a file in separate branches, or one branch may add or remove a file while another modifies that same file. In these scenarios involving simultaneous and conflicting changes, version conflicts may arise when the merge tool is unable to automatically determine how to combine the files [52], [67]. In this context, merge conflicts began to surface in the field of studies in software configuration management as highlighted

by Mens' seminal work [52], which provides details of software merging.

An issue in collaborative software development is the resolution of merge conflicts, which may arise independently of the merge technique adopted by the merge tool [52]. A merge conflict is characterized by developers attempting to integrate different versions of the same code [52]. The merging process only completes after conflicts are resolved, which may be an overlooked activity during software development. Such situations can make the version control process time-consuming and error-prone [67], consequently leading to unnecessary effort during integration activities [17], [24]. Researchers need to develop approaches that take software development practices into account to effectively resolve merge conflicts. In this context, and considering the techniques related to the number of versions involved in the merging process, we can cite two-way and three-way merge strategies.

Two-way merging tries to merge two versions of the same code by directly comparing the differences between them [52]. Due to this limitation, it is not possible to determine whether the differences stem from modifications made in one or both versions simultaneously [52].

Conversely, three-way merging compares the two versions to be merged with a common ancestor from which both versions originally derive [52]. This technique is more robust in detecting conflicts, as it considers the historical changes made in the common ancestor [52]. As a result, it becomes possible to perform merge operations based solely on the specific modifications introduced in each version. This additional information improves the ability to incorporate code changes, additions, and deletions more accurately.

Figure 2.1 illustrates a typical three-way merging scenario involving four program commits: a common ancestor or base version (b), a left version (l), a right version (r), and the merged version (m), which integrates the left and right contributions by using the common ancestor as a reference for decision-making during the merge process [74], [83]. In tools that implement three-way merging using a textual strategy, such as Git, conflicts are resolved based on textual similarity [8], [52]. Hereafter, any reference to software merging will refer to the three-way merging approach.

When conflicts occur, the version control system emphasizes specific sections within the file where conflicting changes occurred. These sections are often referred to as conflicting chunks or conflicting regions. Git uses special markers " <<<<<<< ", "=======", and " >>>>>>> " to surround conflicting chunks in the file, functioning as boundaries of

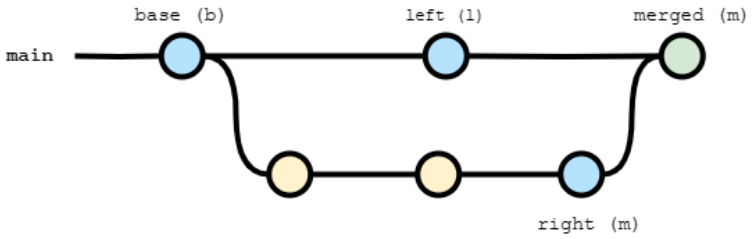


Figure 2.1: Commits in a three-way merge scenario

conflicting changes detected. Figure 2.2 presents these marks in the merged code.

2.1.1 Merge Conflicts and Evolutionary Commits

It is worthwhile to mention the concept of evolutionary commits in the context of software merging to broaden the analysis of a merge conflict. Evolutionary commits are merging scenarios that participate in the code's evolution but are not the four typical commits in a three-way merging [50]. They represent the history of a merge commit and consider the evolution of a conflicting region in merged files. Moreover, they may appear in branches left, right, or both.

Figure 2.2 depicts a typical three-way merging scenario that involves four program commits. A common ancestor or base version, a left version contribution, a right version contribution, and a merged version based on the left and right versions integration when we consider the common ancestor as a comparison base to take decisions during the merge process [74], [83]. In tools that implement three-way merging and apply a textual strategy, conflicts are resolved based on textual similarity [8], [52]. In this scenario, we consider the common ancestor as a comparison base to make decisions during the merge process [74], [83].

In this context, evolutionary commits are intermediate commits that contribute to the code's evolution but are not among the four typical commits in a three-way merge [50]. Considering Figure 2.2, evolutionary commits may be in both branches, represented as yellow circles, observing if they have updates in the same files and lines we found in a merge conflict.

Detailing Figure 2.2, we observe the evolution of Example.java class until a conflict arises. This illustration serves to introduce and clarify other key concepts that will be used

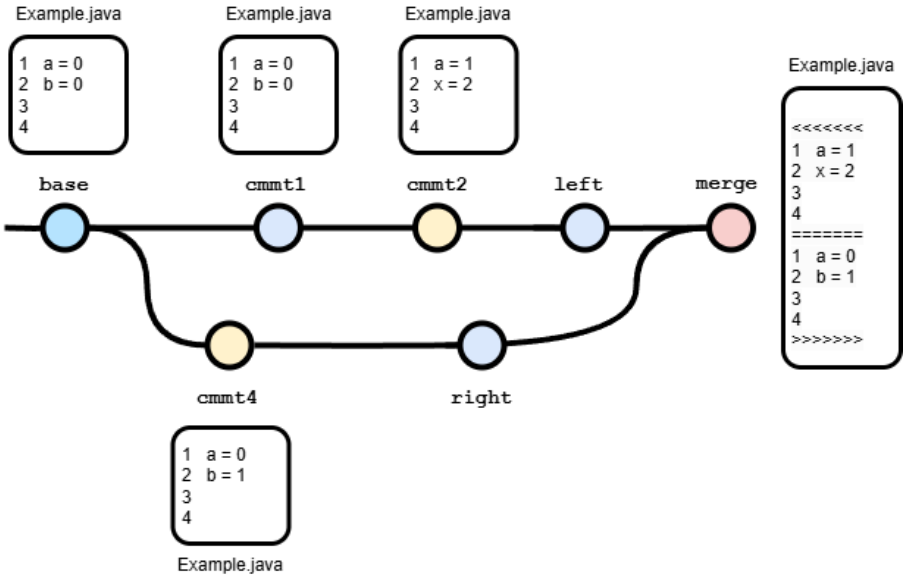


Figure 2.2: A Conflicting Scenario Example

throughout this paper:

1. The conflicting region and the conflicting file: lines 1, line 2, and Example.java file.
2. Evolutionary commits: cmmt2 and cmmt4.
3. Refactoring involved in the evolutionary commits: a renamed variable.

In the cmmt2, we observe the exclusion and addition of a statement changing variable b, or a Rename Variable refactoring, and its value, which change the code behavior and characterize a floss refactoring (detailed in Section 2.2.1). When observing the cmmt4 contributions, just the value of b was changed. Merge conflicts can be particularly challenging to understand when subtle changes are involved, especially when refactorings occur. Even though the changes may seem small in isolation, they can become much more challenging to manage if more lines or files are affected. This underscores the importance of developers proactively understanding all changes in the merge scenario, not just those in the conflicting class, to address merge conflicts effectively.

2.1.2 Merge Approaches

As a current practice, three-way and textual line-based merge tools, also named unstructured merge tools, represent the prevailing method for merging source code artifacts [52], [8], [24].

Unstructured merge operates with plain text, one possible way of software representation in merge tools, which represents an advantage because they are language independent [52]. Moreover, unstructured merge is the preferred approach due to its fast results, despite being susceptible to errors [7], [8], [24].

Unstructured merge tools are prone to errors in conflict detection due to their lack of syntactic and semantic interpretation of the code [26]. These tools operate at the text level, which can result in inaccurate interpretations and failure to identify semantic conflicts. This fact can lead to a lack of detection of non-obvious conflicts and difficulties dealing with significant changes or refactorings in the code [32], [50]. Furthermore, the lack of consideration for the semantic context and the generic application of these tools can compromise the accuracy of conflict detection, especially in specific programming language situations. As complementary approaches, merge tools can also be syntactic, semantic, or structural, depending on how they represent artifacts [52].

General syntactic merge approaches can be categorized based on their underlying data structure: ones that use parse trees and those that use graphs as data structure [52]. Semistructured and structured merge approaches are alternatives to manage the lack of precision in unstructured merge. Semistructured merge approaches partially explore the syntactic structure and static semantics of artifacts to find a balance between unstructured and structured merge [2], [8], [24], [26]. On the other hand, structured merge approaches go beyond simple textual analysis and explore the underlying syntactic structure and static semantics when integrating programs [8], [21], [26], [76], [83].

Unstructured merge performed by Git tool is based on the diff3 algorithm [52]. While merging files, unstructured merge tools often compare two files modified from their common ancestor, the version they were initially derived from, line by line. The unstructured merge tool compares the files using the three-way merge process, which detects different sets of lines, or chunks, within the files. For each block of code, these merge tools verify whether the three revisions have a similar text area that separates the contents of the blocks. The tool reports a conflict if it does not find such an area.

Structured merge tools compare the abstract syntax trees (ASTs) of the code examined to perform merging operations [26]. They also utilize syntactic structure and static semantics. When different developers have modified the same AST node, conflicts may occur [26]. In

collaborative software development, structured merge tools have advantages and disadvantages compared to other merging strategies, such as unstructured or semistructured merge tools. Structured merge tools may reduce the number of reported conflicts, increase the correctness of the merged code, and support more complex and fine-grained code changes [65]. While these tools offer greater precision compared to unstructured ones, they often come with the trade-off of being slower and less effective at preserving the original code structure [7]. Additionally, their dependence on specific programming languages makes it challenging to apply them universally across all languages.

Semistructured merge tools explore both the syntax and semantics of a programming language [2], [8], [24]. These tools define certain program parts as trees and include information about how nodes of specific types, such as methods and classes, should be merged. These trees also incorporate some syntactic structural information. For example, in Java, classes, methods, and fields appear as nodes in the tree, while statements and expressions in method or constructor declarations appear as plain text on the tree leaves [8]. Unstructured or textual merge is applied to merge the leaves. Representing some programming language elements as nodes in the structured merge may significantly impact performance. However, this approach represents a happy medium when compared to the characteristics of structured and unstructured ones.

Although structured merge resolves many conflicts automatically, it falls short when the semantic rather than the syntactic level is involved [76]. Structured and semistructured are less flexible than unstructured merge tools, which motivated the development of agnostic merge tools to avoid the dependence on specific programming languages. However, these tools still need more validation.

Although structured merge resolves many conflicts automatically, it falls short when the semantic rather than the syntactic level is involved [76]. Furthermore, structured and semistructured merge approaches are less flexible than unstructured ones, which has motivated the development of language-agnostic merge tools that avoid dependence on specific programming languages. However, these tools still require further validation [34].

Exploring ways to resolve semantic conflicts is necessary [32], [52]. One of the challenges in software merging toward conflict resolution is understanding the semantics of code changes and improving the accuracy and quality of merged code by leveraging the semantic

meaning of code changes [22]. Considering this, a topic that needs further studies is the analysis of the merged code's semantics and its impact on conflicts and their resolution.

2.1.3 Merge Tools

This subsection provides a more detailed description of the two merge tools, jFSTMerge and IntelliMerge, considered in this study.

jFSTMerge

jFSTMerge [24] performs a semistructured merge tool as an improvement of the FSTMerge Tool [8]. FSTMerge partially leverages the syntactic structure and semantics of artifacts, performing a structured merge by treating high-level tokens as nodes [8]. It applies a FeatureHouse framework as a basis, which utilizes a general model of the structure of software artifacts, specifically the Feature Structure Tree (FST).

FSTs may represent any artifact with a hierarchical structure of nodes. Additionally, the concept of FST superimposition is employed to merge corresponding nodes that are matched by their names, types, and relative positions, starting from the root and proceeding recursively. However, FSTMerge integrates the terminal nodes, such as method bodies, by applying textual merging, similar to the Git merge tool [8]. In the event of an automatic merge failure, the tool displays a message indicating a merge conflict. FSTMerge leverages FeatureHouse framework, becoming a generic merge engine, tested with Java, C#, and Python plugins [8].

jFSTMerge is an specialized FSTMerge tool for Java language. Consequently, its merging process is similar to a three-way merge. Each version of the code is transformed into a syntactic tree, which contains the structural elements of the code, such as classes and methods. The tool compares these syntactic trees, attempting to integrate the differences while preserving the code structure. This improved tool reduces the number of merge conflicts when compared to FSTMerge tool.

jFSTMerge addresses conflicts in a fine-grained manner, distinguishing them by applying specific handlers. The tool also applies these conflict handlers, which are algorithms that analyze code after the semistructured merge and take actions to fit the results according to the

programming language definitions. Conflict handlers in jFSTMerge work to avoid semantic conflicts and ordering conflicts. jFSTMerge can also detect whitespaces and comments, avoiding formatting conflicts.

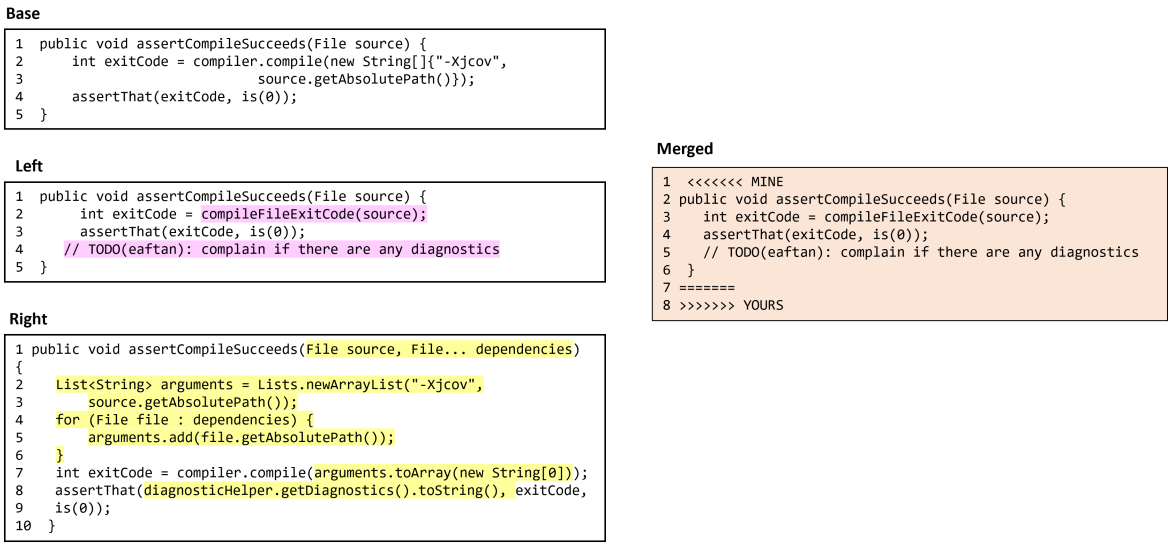


Figure 2.3: A jFSTMerge conflicting merge scenario

Figure 2.3 illustrates a typical example of merging using this tool. In the base version (top left), the `assertCompileSucceeds` method has a simple implementation. In one of the derived versions (bottom left), a developer adds a diagnostic message (TODO) and replaces the direct compiler call with an auxiliary method (`compileFileExitCode`). In the other version (bottom), another developer extends the method signature to receive additional dependencies and modifies the internal logic to process them.

In this example, both developers independently modified the same method. A traditional line-based merge reports a conflict because the edits overlap textually. JFSTMerge analyzes the program structure and identifies that concurrent changes were performed on the same syntactic element. Rather than relying solely on textual overlap, the tool represents the conflict at the level of the program structure, allowing developers to reason about the conflicting modifications more precisely.

Due to its improvements, jFSTMerge may avoid reporting spurious conflicts and, when such conflicts occur, present them in a way that is easier to analyze and resolve [24]. It may also reduce false negatives compared to structured merge and avoid ordering and formatting conflicts. However, it can still encounter issues with certain types of changes, such as rear-

ranging import statements or matching initialization blocks [26]. In addition, jFSTMerge can unintentionally spot some conflicts related to refactoring [24]. Moreover, it detects conflicts related to changes in the same line in the declaration body, which represent a false positive caused by refactoring regarding unstructured merges.

Rearranging import statements that involve types with the same identifier, not matching initialization blocks, and unstructured merge accidentally conflict detection are the three primary sources of the additional false negatives of semistructured merge [24]. Conversely, renamings may cause semistructured merge tools to detect conflicts even when no actual conflict exists [24].

IntelliMerge

IntelliMerge is a refactoring-aware merge tool that extends semistructured merging by representing source code as Program Element Graphs (PEGs) and matching refactored entities before performing the merge. The tool was proposed to address limitations of existing merge approaches, which often fail to correctly recognize refactored program elements and consequently report unnecessary conflicts or produce inaccurate merge results in scenarios involving refactoring [33], [75]. To overcome these limitations, IntelliMerge implements a refactoring-aware merging algorithm for Java programs [73].

IntelliMerge represents source codes as graphs instead of the traditional representation in semistructured tools which applies abstract syntax trees. The graph leaves correspond to methods and constructors and the merge of these leaves is performed using textual strategy. IntelliMerge tool also presents a refactoring-aware process which is its main characteristic. The tool does not apply refactoring detection techniques, such as RefactoringMiner, to explicitly identify refactoring operations. Instead, IntelliMerge infers correspondences between program elements by analyzing contextual, lexical, and structural similarity metrics by applying heuristics and confidence thresholds to infer whether a refactoring occurred. This implementation enables IntelliMerge to recognize and merge changes involving refactoring operations such as renaming, move, extraction, and inlining modifications.

Figure 2.4 presents a conflicting merge scenario from IntelliMerge. In the example shown, one contribution alters the creation of the description object using `DescriptionBuilder.forMethod(...)`, while the other modifies how the descrip-

tion is stored in `methodDescriptions`. Instead of treating these changes merely as concurrent text edits, IntelliMerge identifies the affected program elements and the operations performed on them, as well as the Rename Field refactoring, highlighted in the example. The tool then attempts to automatically combine compatible changes, reducing spurious conflicts and improving the integration of changes involving refactoring or structural modifications to the code. However, when the changes are semantically incompatible, as in the merge scenario presented, manual resolution may still be necessary.

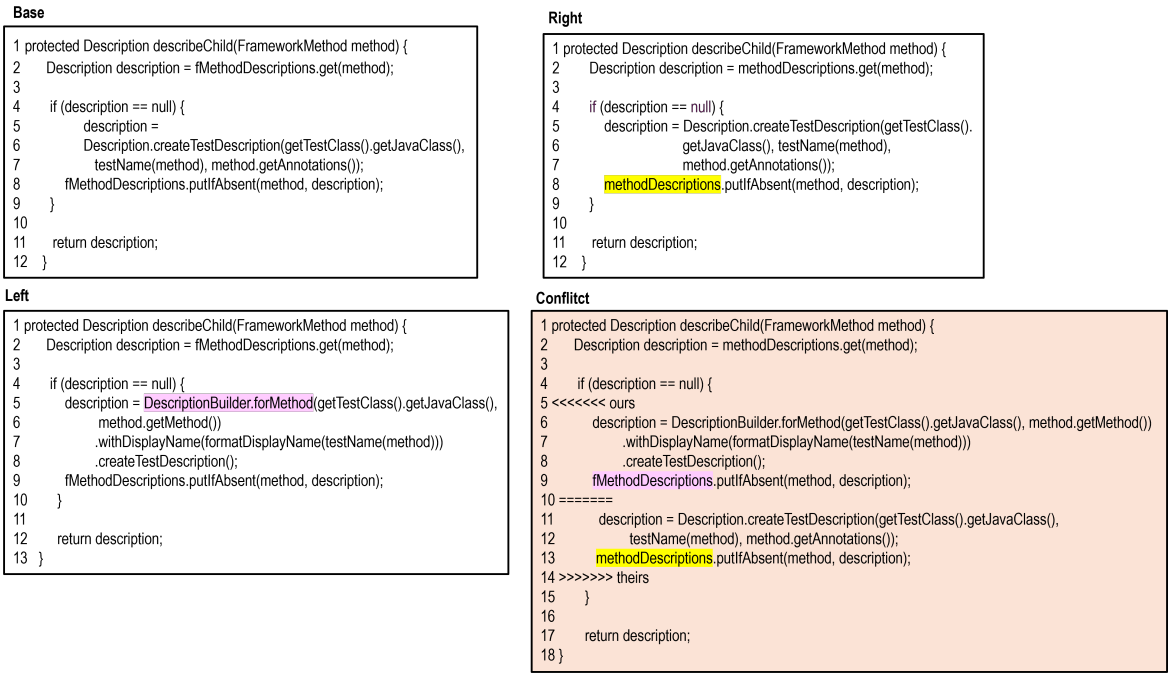


Figure 2.4: A IntelliMerge conflicting merge scenario

The tool analyzes the context and structural relationships among program elements represented in the PEGs. A PEG represents source code as a graph whose vertices correspond to program entities, such as classes, methods, fields, and interfaces, while edges represent structural and behavioral relationships among them, including method invocations, field accesses, inheritance, and object instantiations. The tool applies four main steps in merging process:

- *Code-to-graph step*: this step transforms the source files involved in the left, right, and base scenarios into three PEGs;
- *Matching step*: this step compares the PEGs vertices and identifies the matching pro-

gram elements. Also, it matches the unmatched vertices based on context, name, and body similarities, calculates the confidence threshold for possible matchings, and matches refactored program elements;

- *Merging step*: this step constructs a merged PEG after the matching process finishes;
- *Graph-to-code step*: this step converts the merged code into Java source files. If unsolved conflicts occur during the merging phase, the conflict blocks will be signaled using marks similar to those used by Git merge tool.

IntelliMerge presents some limitations. Its current implementation supports only Java projects, may exhibit lower performance in large-scale merge scenarios due to graph construction and matching operations, and does not guarantee the identification of all merge conflicts.

2.1.4 Research in Software Merging

In the broader context of software merging, studies are divided into two major areas. The first area is related to the development of merge tools [8], [24], [73], [83], and the evolution of unstructured merging [28]. Beyond textual, structured, and semistructured merge approaches, researchers have proposed alternative techniques to improve merge accuracy. Recent work exploits language-specific syntactic separators to emulate structural information while still relying on diff3-based merging [25]. Other approaches focus on semantic conflict detection through static analysis or automated test generation [29]. The second area encompasses research aimed at understanding the correlation between developed code and merge conflicts, comprehending the nature of conflicts, conflicts prediction and seeking resolution strategies based on the characteristics of the conflicts themselves [2], [39], [51], [59], [74], [82].

Significant challenges for merge tool developers are that they must be language-independent to handle any document and intelligent enough to produce meaningful results [83]. In other words, the requirements for the ideal merge tool, such as language independence, syntax, and semantic awareness, are conflicting [83]. Balancing accuracy and performance is a pressing challenge in constructing merge tools [8]. Among these challenges, it is essential to recognize that the primary issue in the context of merge tool development is the inability to detect and resolve all conflicts present during merging [7].

Regarding the presented context, exclusive automatic conflict resolution must be applied with caution [8], leading to the belief that it is not the best solution for the problem [82]. In addition, minimizing the amount of manual intervention is desirable, and maintaining information about conflict resolution is beneficial to help developers resolve conflicts [8]. These requirements emphasize the demand for tools that can be used to inform developers about conflicts better and provide options for resolving them [74], [82].

On the other hand, resolving merge conflicts requires developers to use and find information about the changes they are about to merge [15]. Often, this information can be found in the conflict region itself [39], [74]. However, merge tools may not utilize this information or other details about the conflict, such as dependencies on code external to that class or method, which is not considered when resolving the conflict.

Considering the conflict characteristics and how they affect its resolution, some authors [2], [39], [74], [82] uncovered patterns that can be explored during conflict resolution. Besides, there are characteristics to observe in a merge conflict that may address its resolution. Some are direct, such as LOC (Lines of Code), number of conflicts, conflicting files, and developers involved [73], [82]. Others are indirect but equally important, for example, dependencies among conflicting and non-conflicting files [17], [39], [82]. Merge conflicts can also be complex and cumbersome. As a result, difficulty understanding and resolving a conflict is also a characteristic related to its complexity, an aspect previous studies analyzed [17], [50], [82]. Finally, some authors suggest that there is an ideal order to resolve merge conflicts, considering that some operations should be applied first and the understanding of the role of refactorings in the course of the merge process should be explored as a future issue [8], [39], [82]. The section 2.2 brings up more details about the refactoring subject and how these two concepts can be related.

2.2 Refactoring

Refactoring is a commonly practiced technique in software development, which involves systematically applying behavior-preserving transformations to improve the design of an existing code base [37], [64]. Considered an essential activity for improving the quality, maintainability, and scalability of software, refactoring traces its origins to the early practices

of object-oriented programming and agile methodologies [11], [64].

The concept of refactoring was formally introduced in 1992 [64]. This work established a foundation for identifying and categorizing various refactoring types, along with the conditions required to apply them safely. Later, Martin Fowler popularized the concept in the software engineering community, expanded the types of refactoring previously defined, and devised a catalog of refactorings along practical guidelines for applying them during software evolution [37].

Over time, developers have applied refactoring as a systematic approach to enhancing code quality without altering its external behavior. This process is driven by the goal of making the code base more adaptable to future modifications and reducing the risk of introducing errors during development. Therefore, refactoring is an activity associated with software evolutionary maintenance.

Refactoring evolved over decades as a fundamental practice in software development, reflecting the diverse research traditions that permeate Software Engineering [53]. From the initial approaches proposed by Martin Fowler to modern refinements, the literature highlights distinct research traditions. Initially focused on code improvements and maintenance, these traditions broadened to include performance, security, and continued integration research [53]. These traditions also emphasize the code restructuring and the conceptual evolution of the refactoring role in the software life cycle [1], [53].

Originally, refactoring techniques were developed to deal with bad smells [37]. Bad smells are specific structures in the code that indicate the potential need for refactoring [12]. Duplicate code, feature envy, and large classes are examples of practices that refactorings can improve. Researchers, however, discovered that refactorings are more applied to fix bugs and add functionalities [66], [75].

2.2.1 Pure and Floss Refactoring

Beyond the motivations driving refactoring, the manner in which programmers approach these activities introduces additional layers of complexity. In this context, refactoring tactics are specific choices used during the process of refactoring code that assists developers in improving their code's design, structure, and readability [37]. Two important tactics in refactoring are pure refactoring and floss refactoring [56]. Pure refactoring, formerly known

as root-canal refactoring, involves changing the code structure exclusively without altering behavior [56], representing a complex activity of looking for refactoring opportunities throughout the large modern software applications [42], [48]. In contrast, floss refactoring involves making small changes or performing small-scale refactoring [42], [48] to prepare code for adding features or fixing bugs [56].

Considering that empirical evidence presents floss refactoring as the most applied tactic [4], [42], [48], [56] we must comprehend its definition. When a programmer chooses to interleave refactoring activities with improvements in code, for example, adding a feature or fixing a bug, they perform floss refactoring [55], [56]. Floss refactoring is also related to the improvement of code's health, which differs from pure refactoring, where programmers only perform activities related to the refactoring itself in order to improve its internal quality [4], [55]. When a programmer makes a semantic change during a refactoring session, it is considered a floss-refactoring tactic application; however, when a programmer refactors without altering the code semantics, it is considered pure refactoring [56].

With the advancement of refactoring practices, developing specific tools has become a crucial part of the process. Various integrated development environments (IDEs), such as Eclipse and IntelliJ, and development environments offer integrated support for a range of refactoring tasks, from renaming variables to extracting complex methods. The main goal of applying these tools is to ensure consistency and accuracy, reducing the potential for human errors throughout the process. Considering that developers prefer refactoring manually and refactoring tools cannot always perform refactoring when intertwined with other pieces of code [56], using these tools might make it a laborious and error-prone process [9], [56].

To thoroughly understand floss refactoring, it is important to understand what constitutes refactoring itself, that is, which transformations are applied during a refactoring operation and what edits may occur beyond those transformations. Refactoring consists of specific actions applied to code to improve its design, structure, and maintainability without altering its external behavior [37]. In some cases, however, additional edits, such as insertions, deletions, or updates, may occur as part of a refactoring session. Because floss refactoring combines refactorings with other code modifications, it tends to involve a larger number of error-prone changes [9], [45], [54], which may increase the likelihood of merge conflicts.

Conversely, some studies argue that pure refactorings deserve greater attention and that

more specialized tools should be developed to handle this type of refactoring tactic [60], particularly in large-scale projects [42]. These studies characterize pure refactorings as more detailed and structurally oriented, requiring a higher level of scrutiny. Given this context, we believe exploring these nuances in refactoring may shed light on the merging process.

2.2.2 Refactoring Miner Tool

In the evolving world of software development, the ability to adapt and refine code is critical. As projects grow and change, so too does the need for robust tools that can intelligently track and analyze these transformations. To bridge this gap, RefactoringMiner has established itself as a revolutionary tool allied in the context of code evolution.

The RefactoringMiner tool recognizes refactoring types in merge commits, tracking how these transformations may be related to how software evolves [80]. It also overcame the lack of precision from previous tools by detecting refactoring candidates using an AST-based statement matching algorithm [80].

Considering the code history of a Java project on GitHub, the RefactoringMiner tool takes as input two revisions, which are typically a commit and its parent, and compares the changed files between them [80]. It returns a list of refactoring operations, ranging from low-level changes, such as Extract Method and Rename Method to higher-level structural ones, such as Move Class and Extract Class. For method-level changes, RefactoringMiner uses two pre-processing techniques named abstraction and argumentization to improve the matching of method bodies, particularly in the presence of Extract Method and Inline Method refactorings [80].

The abstraction technique deals with changes in AST types to abstract statements associated with refactoring operations. This technique is related to the deletion or addition of a return statement, which is the cue for identifying specific refactorings. Argumentization is associated with the parametrization involved in some refactoring operations, substituting parameter names with their corresponding arguments. Such a technique replaces expressions with parameters and vice versa, corresponding to the mechanics of specific refactorings. By applying abstraction and argumentization, the RefactoringMiner tool leverages the changes in code structure to detect refactorings where string similarity algorithms would fail.

We present the Figure 2.5 as an illustration of the operation of RefactoringMiner tool.

Given two commits or consecutive revisions of a source file, the tool compares their AST representations and identifies refactoring patterns. In the example, the method calc() was renamed to calculate(), which is reported by RefactoringMiner as a Rename Method refactoring.

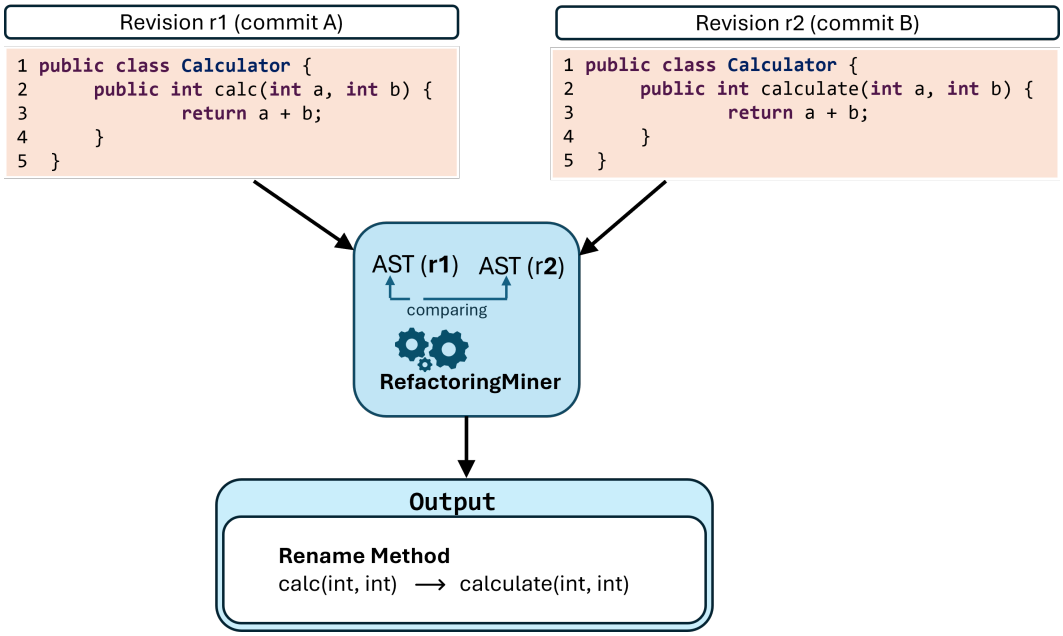


Figure 2.5: Example of Refactoring Miner Operation

Overly applied in previous studies [27], [35], [50], [66], [73], [75], [77], and frequently improved, the RefactoringMiner tool has a precision of 95% and a recall of 92.5%. It supports 40 refactorings from Fowler's book and 52 atomic refactorings related to API changes, including renaming operations, moving methods and attributes, and other structural transformations, such as extract and inline method [79]. The first version of the tool presented an innovative algorithm that does not rely on code similarity to function [80]. In its second version, the authors included support for low-level refactorings inside methods' bodies [79]. In its recent version [6], the Refactoring Miner tool included an API to detect pure and floss refactoring named PurityChecker, which will be presented in the next Section.

The PurityChecker API

PurityChecker [61] is an API integrated into RefactoringMiner 3.0 [6], developed to analyze and classify refactorings with respect to their purity. In this context, purity refers to whether

a refactoring performs only the intended transformation, without being mixed with unrelated changes, in these cases falling into a floss refactorings classification. The PurityChecker evaluates refactoring purity based on the abstraction and argumentization heuristics originally proposed by the RefactoringMiner tool [80]. This approach does not directly verify behavior preservation, but instead assumes that refactorings following typical and localized structural patterns are pure method-level refactorings.

Based on how PurityChecker operates, three main categories of behavior-preserving code changes are considered relevant to refactoring purity [61]:

- Modifications related to the specific mechanics of the refactoring type;
- Adjustments or changes that are not essential for statement mapping;
- Modifications arising from overlapping refactoring activities.

PurityChecker supports purity detection for the following refactoring types: Extract Method, Inline Method, Move Method, Pull Up Method, Push Down Method, Split Method, Extract and Move Method, Move and Inline Method, and Move and Rename Method. The tool analyzes the purity of refactorings in a commit by taking the previous commit in the code evolution as a reference. In this process, it first examines code replacements within mapped statements, followed by an analysis of non-mapped statements. The analysis begins with the application of abstraction and argumentization techniques, proceeds to the identification of tolerable changes, and finally evaluates the effects of overlapping refactorings, assessing whether the code preserves its original behavior after the transformations. Subsequently, the tool evaluates tolerable additions and deletions, including those caused by overlapping refactorings. If all identified substitutions can be justified, the refactoring is considered pure; otherwise, it is classified as floss refactoring.

Figure 2.6 illustrates the Purity Checker execution in a simple manner. The Purity Checker receives the refactoring detected by RefactoringMiner, a Rename Operation as presented in the Figure 2.5. The tool compares the two versions considering the modifications in the analyzed commit. Whether all edits can be explained by refactoring operations, the commit is classified as a pure refactoring. In this example, the presence of additional non-refactoring edits regarding the change in the arithmetic operation in the method return leads to a non-pure classification.

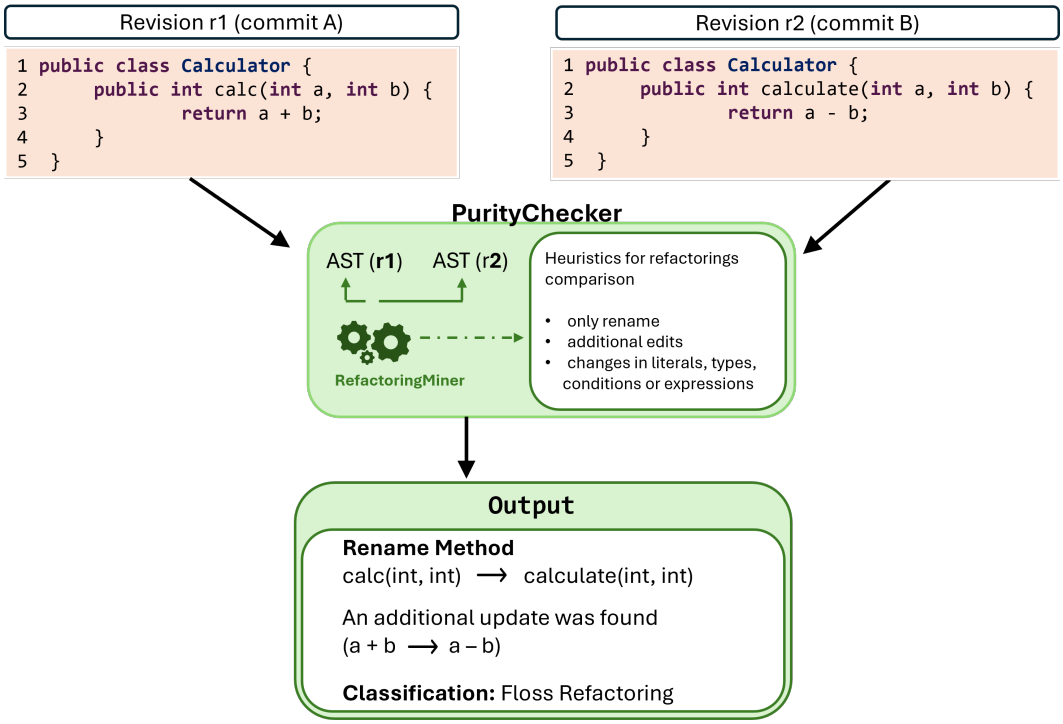


Figure 2.6: Purity Checker execution simplified

In the context of overlapping refactorings, other types are analyzed when they are related to the supported ones. For example, Inline Variable is considered in the context of an Inline Method, and Extract Variable in the context of an Extract Method. Since developers often perform additional refactorings during the application of a primary refactoring, these overlapping changes are not treated as modifications that alter the core mechanics of the main refactoring.

The PurityChecker API results include the refactorings detected in a merge commit, a classification into pure or non-pure refactoring, a brief description of the reasons for the classification, and information about overlapping refactorings. The tool achieved precision and recall rates above 90% [61], indicating strong reliability in identifying pure refactorings.

2.2.3 Refactorings and Merge Tools

In light of the role of refactorings in merge conflicts previously outlined, MolhadoRef is a tool designed to handle refactorings during the merging process [32]. This tool performs three-way merging and employs an operation-based approach, in which a code version is represented as a sequence of operations used to construct it. Merging is then performed at the

API level. MolhadoRef treats a version as composed of the following three operations: API refactorings, API edits, and code edits [33]. When additional edits introduce clear and well-defined semantic modifications in code, MolhadoRef can merge them more reliably and with fewer conflicts. However, if the changes go beyond the mechanics of refactoring and involve complexities that cannot be resolved automatically, some level of manual intervention during the merging process may be necessary.

A large empirical research showed some refactorings, such as the Extract Method, are common in merge conflicts, which are not addressed by MolhadoRef [50]. This fact can be pointed out as a MolhadoRef disadvantage since the Extract Method is the most performed refactoring in projects [75]. An important result of the same study relating refactorings and merge conflicts showed that 22% of merge scenarios had refactorings involved; moreover, 11% of conflicting regions with refactoring involved tend to be larger than the ones without refactorings [50].

Considering the challenges in merging some refactorings, other options to deal with them were implemented. IntelliMerge is also a semistructured merge tool that is designed to handle conflicts caused by refactoring operations [73]. It attempts to address the limitations of MolhadoRef in handling more complex refactorings like Extract Method and Inline Method [73]. It does this by implementing a graph-based refactoring-aware algorithm for object-oriented programs written in Java [73]. For this purpose, using graphs instead of ASTs in matching and merging operations can be more accurate, but it is computationally more complex [7], [52].

As an option in this context, RefMerge was developed to provide improvements and to address limitations of MolhadoRef [35]. Comparing the application of these two tools, RefMerge and IntelliMerge introduce conflicts due to their adopted approaches. As a syntactic tool, IntelliMerge effectively manages conflicts related to ordering and formatting, but it encounters challenges when dealing with class-level refactorings and scenarios involving multiple refactorings. IntelliMerge may introduce additional conflicts, unexpected merges, or omit conflicts because it relies on a similarity score to detect refactorings [35], [80]. In comparison, RefMerge merges the refactorings in a merging scenario, but it introduces conflicts when inverting refactorings associated with movements [35].

As we have discussed semistructured merge tools up to this point, it is important to bring

to light the tool that represents the state of the art in this approach. jFSTMerge [24] performs semistructured three-way-merge tool as an improvement of FSTMerge Tool [8]. jFSTMerge partially exploits the syntactic structure and semantics of artifacts and performs a structured merge by considering high-level tokens as nodes. Within constructors and methods' bodies, jFSTMerge performs textual merge.

By applying its custom handlers, this tool avoids ordering and formatting conflicts, thereby reducing spurious conflicts that commonly arise in unstructured merge results. It also decreases the number of undetected conflicts in structured merge scenarios [26]. However, jFSTMerge is not refactoring-aware. As a result, merge scenarios involving renamings, moves, or other refactoring operations may still lead to additional conflicts or incorrect merge results, even when the changes are not semantically incompatible [24].

In summary, while advancements in merge conflict resolution tools represent significant progress, important limitations remain in handling specific scenarios, particularly those involving complex structural or semantic changes. This limitation, particularly evident in refactorings and nuanced conflict scenarios, highlights a critical area for future research. Investigating improved strategies for conflict detection, as well as the relationship between refactoring operations and merge conflicts in these contexts, presents promising avenues for further study. By exploring these challenges more deeply, researchers can contribute to the refinement of existing tools and methodologies, fostering more robust and adaptive solutions in the evolving software development landscape.

Chapter 3

Investigating Factors Related to Merge Conflicts: A Tool-Supported Study

In software development, merge conflicts pose a significant challenge, particularly when integrating changes across multiple branches. This chapter investigates the conflicts that Git, jFSTMerge, and IntelliMerge detect. It draws attention to the complexities arising from textual differences and structural and semantic changes in the codebase. We consider five variables to analyze merge conflicts, focusing on how code contributions evolve into conflict scenarios. Using logistic regression, we model the factors associated with merge conflicts, enabling a broader evaluation of how different tools handle conflicts. In particular, we emphasize the role of refactorings and examine how their presence relates to conflict occurrence across tools. Through this analysis, we intend to deepen our knowledge of merge conflicts and improve the merging process in collaborative software development environments.

3.1 Research Goals and Questions

In this chapter, we analyze factors associated with the occurrence of merge conflicts. We also examine how Git, jFSTMerge, and IntelliMerge relate to these factors, highlighting differences and potential limitations in how these tools handle conflict scenarios. Our analysis focuses on how historical elements of a merge, such as developer activity and refactoring practices, relate to conflict occurrence. The variables used to address this goal capture the scope of changes applied during code reviews and are summarized in Table 3.1.

Table 3.1: Variables collected in this study

Variable	**Description**
Merge conflict occurrence *(dependent variable)*	A binary variable represents the presence or absence of a merge conflict in a merge scenario, with 1 (one) indicating at least one merge conflict in a merge scenario and 0 (zero) indicating no conflict.
Number of evolutionary commits	The number of evolutionary commits in a conflicting merge scenario.
Number of refactoring regions	Number of regions with refactoring in a conflicting merge scenario.
Number of developers	The number of developers in each conflicting contribution
Number of refactorings in evolutionary commits	The number of refactoring operations performed within evolutionary commits.
Number of refactorings in parents	Number of merge parents with refactorings in a merge scenario.

The variables were derived from previous studies and selected to capture complementary dimensions of merge conflicts. The literature already pointed out refactoring variables related to refactoring as relevant in merge conflicts [33], [35], [50], which represents the structural dimension of a merge conflict; the variable related to evolutionary commits refactoring transformations represents the accumulation of changes over time [50], and the number of developers reflects the concurrent changes by collaboration [59]. While these variables provide broad coverage of relevant dimensions, other factors may also be associated with merge conflicts, such as code ownership, developer experience, semantic dependencies, or the types of changes made. However, these variables were not included due to limitations in data availability or scope, and represent opportunities for future work. To explore each variable in greater detail, we developed three research questions, as outlined below.

- **RQ.1 - What factors influence the occurrence of merge conflicts?**

  Understanding the factors associated with merge conflicts is essential to anticipate and mitigate integration problems in collaborative development. In this question, we investigate how historical characteristics of a merge scenario,such as the accumulation of changes, refactoring activity, and developer participation, relate to the occurrence of conflicts. The variables to answer this question reflect the scope of changes applied

in a code review and are presented in Table 3.1.

- **RQ.2 - Which types of refactoring are most associated with merge conflicts?**

  While prior work has shown that refactoring activity may be related to merge conflicts, less is known about how different types of refactorings contribute to this phenomenon. In this question, we aim to identify specific refactoring patterns that are more frequently associated with conflict scenarios. Using data provided by the Refactoring Analysis tool [50], which relies on Refactoring Miner 2.0 [79], we collected data on various refactoring types, focusing on instances where lines modified by refactoring operations overlapped with those involved in merge conflicts.

These questions and variables may provide a comprehensive view of merge conflicts by examining their attributes, the impact of change history, and the specific role of refactorings.

3.2 Study Setup

In this section, we describe the repositories, merge tools, and variables investigated in our study.

3.2.1 Subjects Description and Characterization

We analyzed a dataset of 530 GitHub projects predominantly written in Java, obtained from an initial sample of 600 projects after applying filtering criteria to exclude non-Java projects. Not all projects presented merge conflicts; however, this is expected, as merge scenarios may or may not result in conflicts. Therefore, our analysis includes both conflicting and non-conflicting merge scenarios, enabling a more comprehensive modeling of conflict occurrence. Table 3.2 provides more detailed information on the selected projects. We observed the number of forks and stars in our selection to offer a clear and concise overview of the dataset's diversity and scale, emphasizing the variability in key metrics across the sampled Java projects [43], [57]. Additionally, the dataset comprises projects from multiple software domains, including Android applications, libraries, frameworks, infrastructure software, and developer tools. This diversity reinforces its heterogeneity in development and

integration practices, increasing confidence that the observed conflicting merge scenarios are not restricted to a single application domain.

Table 3.2: Projects data

Characteristic	Range	Mean	Median
Merge commits	1 - 13,945	347.21	33
Conflicting commits	1 - 4,072	51.92	7
Developers	3 - 143	4.88	2
Forks	7 - 26,333	654.27	192
Stars	18 - 68,473	1,803.54	554.5

3.2.2 Collecting Historical Data on Subjects

In our study, we employed GitHub data to analyze repositories using the Refactoring Analysis Tool [50], which provides detailed information about a project's commit history. The Refactoring Analysis tool examines each project's commit history to identify failed merge scenarios with reported conflicts in Git tool. For each scenario, it locates all commits that modified the conflicting code between the base and parent commits. It then utilizes RefactoringMiner [79], [80] to detect refactoring changes in these commits by comparing them to their previous versions.

3.2.3 Merge Tools Selection

For this study, we selected three merge tools: Git, jFSTMerge, and IntelliMerge. We selected Git because it is the state-of-practice tool, broadly used in software development to manage source code and coordinate collaboration among developers.[1]

Different merge tools may report different conflict outcomes for the same integration scenario due to their merging strategies. During an exploratory manual inspection of five repositories, we observed disagreements among Git, jFSTMerge, and IntelliMerge regarding conflict detection and resolution, as well as different categories of merge issues associated

[1]Survey on version control system preferences among developers, available at: https://survey.stackoverflow.co/2022

with each tool. These observations reinforced the need to investigate merge conflicts using multiple merge strategies rather than relying on a single tool.

Considering the improved conflict-resolution capabilities of semistructured approaches, which may reduce the number of reported conflicts while balancing complexity and simplicity [24], we included jFSTMerge, a state-of-the-art semistructured merge tool that manages abstract syntax trees and applies handlers to address specific merge conflict issues [2].

Since refactoring is a common activity in modern software systems [50], understanding how refactoring operations relate to the merging process is essential. Therefore, we also included IntelliMerge, a refactoring-aware merge tool that enables the analysis of conflicts involving structural code changes. Together, Git, jFSTMerge, and IntelliMerge represent textual, semistructured, and refactoring-aware merge approaches, respectively.

All of our tools' executions were conducted on two computers:

- Intel(R) Core(TM) i7-1065G7, CPU @ 1.30GHz 1.50GHz, 24 GB memory and Windows 11 OS
- AMD Ryzen 7 3700U, CPU 2.30GHz, 24 GB memory, and Windows 10 OS

3.3 Adopted Approach

Our approach to address the questions posed in Section 3.1 comprises four main steps, as depicted in Figure 3.1.

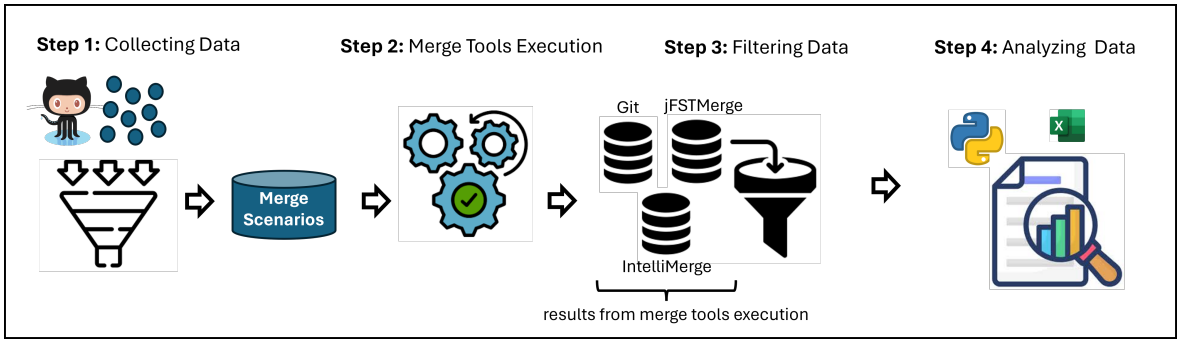


Figure 3.1: Adopted Approach

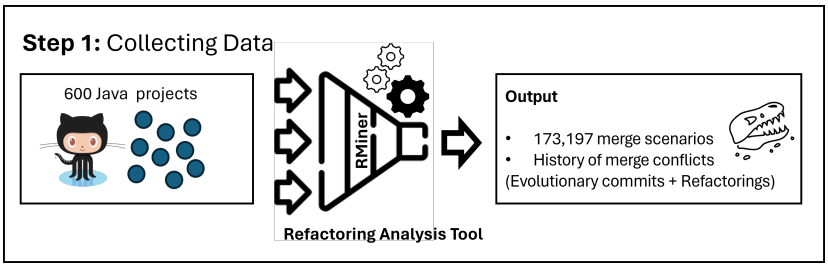


Figure 3.2: Detailing Step 1

3.3.1 Step 1 – Collecting Data

In Step 1, we applied the Refactoring Analysis tool [50], which collects all three-way merge scenarios, conflicting and non-conflicting scenarios, from our subjects and extracts data from the selected GitHub repositories. According to Figure 3.2, the Refactoring Analysis tool examines each project's commit history to identify failed merge scenarios with reported conflicts.

From the results gathered, we selected information on project names and numbers, merge commits, developers involved in merging, conflicting Java files, project histories, and refactorings performed in these projects. The history of a project contains data related to commits involved in the changes of a merge scenario that leads to a conflict, namely, evolutionary commits.

3.3.2 Step 2 – Merge Tools Execution and Environment

Since we could not integrate jFSTMerge and IntelliMerge into the Refactoring Analysis tool, we used the parent commits of each merge scenario identified in Step 1 as input for executing the merge tools. To automate this process, we developed scripts that invoked the command-line interfaces of the three merge tools and executed each merge scenario independently.

Next, we applied Python scripts to collect refactoring information and evolutionary-commit data from the Git results. We then matched the merge scenarios identified by Git with the corresponding results produced by jFSTMerge and IntelliMerge, enabling us to gather and compare data across the three tools. The merge scenarios identified by Git were used as a common baseline to align and compare the results produced by jFSTMerge and IntelliMerge.

In this study we used the following versions of the tools: RefactoringMiner 2.1, Refactoring Analysis Tool 1.0, Git Git 2.33, jFSTMerge (2021 release), and IntelliMerge 1.0.9.

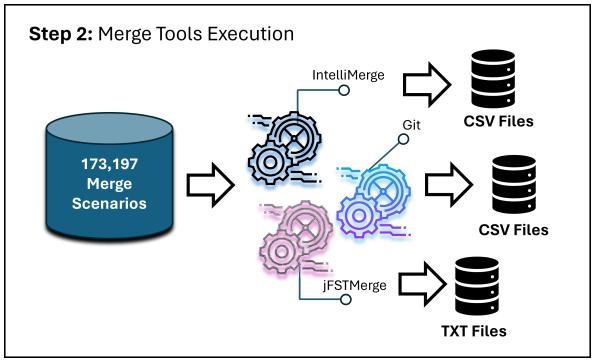


Figure 3.3: Detailing Step 2

3.3.3 Step 3 – Filtering Collected Data

Figure 3.4 presents Step 3, in which we selected the results from the three merge tools using a set of scripts. This step filters the relevant data of merge conflicts found in Step 1. As we obtained raw data from jFSTMerge and IntelliMerge, some preprocessing was necessary before filtering their results. The preprocessing scripts read the output files from jFSTMerge and IntelliMerge executions and saved the results in CSV files.

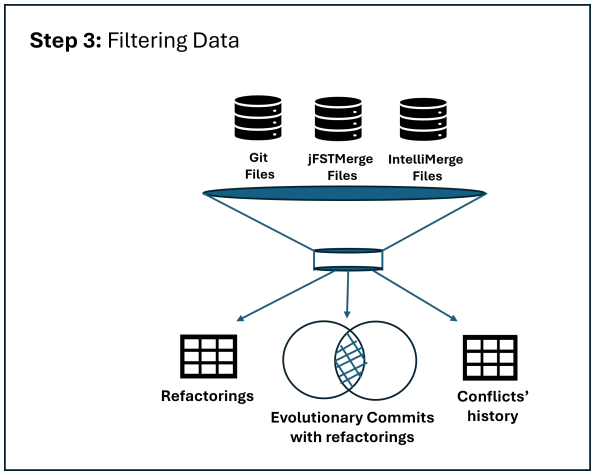


Figure 3.4: Detailing Step 3

Afterward, we collected evolutionary commits with refactorings from the Refactoring Analysis tool results by identifying overlaps between the evolutionary commit region and

the refactoring region (see Figure 2.2).

We implemented scripts to automatically analyze the results obtained in the previous step. These scripts take as input the results from Step 2 (Section 3.3.2). Considering the example in Figure 2.2, our scripts extract the following data: the merge commit that triggered the conflict (merge), the conflicting region (i.e., lines between the < < < and > > > markers in Example.java), the conflicting file (Example.java), the evolutionary commits (cmmt2 and cmmt4), the number of modified files in an evolutionary commit (one file, Example.java), the number of developers involved (two developers), the number of scenarios with refactorings in the evolutionary commits (one scenario, in cmmt2), and the specific evolutionary commit with the refactoring (cmmt2, which includes a variable renaming refactoring).

3.3.4 Step 4 – Automated Analysis

In Step 4, we analyzed the results of the three merge tools using a set of scripts. For each tool, we applied the filtered data from Step 3 to collect the variables listed in Table 3.1, then built regression models and conducted the statistical tests described in the following section.

3.4 Approach for Data Analysis

This section outlines our methodological approach for analyzing the study data. It describes the variables examined and explains how we extracted information from the data using logistic regression models.

3.4.1 Selected Variables

To identify factors associated with merge conflict likelihood, we selected five variables based on prior research [50], [82]. This selection was guided by characteristics previously identified as relevant to conflict occurrence, particularly those related to developer collaboration and code evolution. These variables were chosen for their potential to explain merge conflicts in different scenarios. The occurrence of merge conflicts is modeled as the dependent variable, while the remaining variables are treated as independent factors that may explain conflict occurrence. Table 3.1 presents the dependent and independent variables considered

in this study.

3.4.2 Statistical Analysis

In our study, we conducted a two-part statistical analysis. First, we collected and organized raw data obtained from executing the three merge tools on our dataset. We then computed descriptive statistics to characterize the general behavior of the data.

To further analyze the relationship between variables, we applied logistic regression models [36], [78]. The dependent variable is binary, representing the presence (1) or absence (0) of merge conflicts in Java files, while the independent variables include both continuous and categorical factors (f), as presented in Table 3.1. The model is defined as:

$$m = glm(co \sim f, family=binomial, data=independentVariables) \tag{3.1}$$

In Equation 3.1, *glm* denotes the function used to estimate the logistic regression model, and the *family* parameter specifies the binomial distribution, appropriate for binary outcomes. The dataset *independentVariables* dataset is a generic name for the three files containing information on the variables listed in Table 3.1. These files contain the results of Git, jFSTMerge, and IntelliMerge executions. For simplicity, we adopted this naming convention for the dataset. We developed explanatory logistic regression models using a binary dependent variable representing the presence or absence of merge conflicts. We adopted binary logistic regression due to the categorical nature of the dependent variable and its suitability for modeling the probability of merge conflict occurrence based on multiple predictors, as also applied by [31].

To explore the relationship between the variables under analysis and the studied outcome, we developed eight Logistic Regression Models (Model I - Model VIII):

- Models I to V each include a single independent variable from Table 3.1, chosen and applied in isolation to evaluate their ability to explain merge conflicts. This approach provides a clearer understanding of each variable's unique contribution to explain merge conflict occurrence;
- Model VI combined "Number of evolutionary commits" and "Number of refactorings

in parents" to explore how long-term code changes interacts with structural changes in influencing conflicts;

- Model VII further incorporated the “Number of refactorings in evolutionary commits” to capture a broader range of refactoring activities that may contribute to conflicts;
- Model VIII included four out of the five variables listed in Table 3.1, excluding the “Number of Developers” due to its high correlation with other variables ($r \geq 0.5$), which could introduce multicollinearity. This exclusion ensures the robustness of the regression analysis and prevents skewed results.

We considered odds ratios as results from these models. In logistic regression, odds ratios indicate the strength and direction of the association between predictor variables and an outcome [36]. Values greater than 1 indicate a positive association, whereas values less than 1 indicate a negative association. For example, an odds ratio of 2 suggests that a merge conflict is approximately twice as likely to occur when a given factor is present, while an odds ratio of 10 indicates a much stronger association. Confidence intervals are used to assess the statistical significance and reliability of these estimates [36].

The rationale for developing these regression models was to analyze the variables individually in Models I to V and then refine the results by selecting models with multiple variables, following the Principle of Parsimony to include only essential ones [36]. Including all variables simultaneously may lead to one dominating due to its higher statistical power, giving the false impression of equal importance. This principle ensures simplicity while preserving explanatory power. We also developed models that combine variables closely related to software evolution to provide a more coherent analysis of their combined effects. We excluded most of the tested models with p-values greater than 0.05, indicating they were not statistically significant, leaving only the eight models presented in Table 3.6. From these eight models, we selected those with lower deviance, as they provide better fit to the observed data and greater explanatory power [36].

We further explored the relationship between evolutionary commits and refactorings by grouping conflicting scenarios using the number of evolutionary commits and comparing the number of refactorings on them.

3.4.3 Time to Merge Conflicts with Refactoring Involved

To assess whether the time required to resolve merge conflicts is associated with refactorings, we collected the timestamps of the merge commits under analysis, represented in Git as Unix time in seconds, and converted the resulting time-to-merge values into minutes.

We initially applied Git to reproduce the merge scenarios using our previously collected dataset. To estimate time-to-merge, we collected timestamps from Git logs for both parent commits involved in each merge and the corresponding merge commit. For each scenario, we computed the elapsed time by subtracting the earliest timestamp among the parent commits from the timestamp of the merge commit. This approach approximates the time between the earliest contributing change and its eventual integration. A similar strategy was adopted by Vale et al. [82]. We then divided the 17,598 conflicting merge scenarios identified in Git history into two groups: those involving refactorings and those without. Finally, we compared the time-to-merge between these groups.

Since our data do not follow a normal distribution and involve independent samples, we applied the Mann-Whitney U test to determine whether the distributions, with and without refactorings, differ significantly [36]. Additionally, we calculated the effect size r for the U statistic, interpreting the results as small ($\geq$0.1), medium ($\geq$0.3), or large ($\geq$0.5) [38].

To better understand the role of evolutionary commits, we performed an exploratory analysis comparing the number of refactorings across groups of conflicting scenarios with different numbers of evolutionary commits.

3.5 Results

As a general view of merge conflicts, we present the descriptive analysis of projects executed by the merge tools. This execution revealed differences in the number of projects with merge conflicts and the total number of merge scenarios and conflicts. Table 3.3 summarizes these results. Each tool employs distinctive merge strategies, leading to different counts of projects with conflicts: 352 for Git, 244 for jFSTMerge, and 229 for IntelliMerge. Among the 169,137 merge scenarios in 352 projects with conflicts, we identified 17,598 (10.4%) cases with at least one conflict in the Git results, of which 9,219 (5.45%) involved Java file conflicts.

In the 244 projects analyzed, jFSTMerge identified 154,115 merge scenarios, with 5.20% resulting in merge conflicts. jFSTMerge detected fewer conflicts or resolved more conflicts compared to Git, leading to 8,027 conflicting merge scenarios. For IntelliMerge, we identified 229 projects with at least one merge conflict out of the executed merge scenarios. There were 3,020 merge conflicts, representing 2% of all merges.

Since jFSTMerge and IntelliMerge exclusively merge Java files, we do not have information about conflicts in non-Java files for these tools. Additionally, we identified 192 projects common to all three tools, with 2,603 shared conflicting merge scenarios.

Table 3.3: Summary of the merge outcomes from the tools

Scope of Conflicts	**Git**	**jFSTMerge**	**IntelliMerge**
Projects with conflicts	352	243	229
Merge scenarios with conflicts in Java files	9,219	8,027	3,020

Table 3.4 presents the descriptive statistics for the three tools. IntelliMerge shows a maximum of 692 conflicts in a single project and a median of 3 conflicts, yielding the lowest number of merge conflicts among the tools.

Our results show that 3,952 conflicting merge scenarios in Git involve refactorings, compared to 3,088 in jFSTMerge and 1,441 in IntelliMerge. In percentage terms, this corresponds to 43% in Git, and 32% and 37% in jFSTMerge and IntelliMerge, respectively.

Table 3.4: Distribution of Conflicts per project

	Git	**jFSTMerge**	**IntelliMerge**
Minimum value	1	1	1
1st Quartile	8.5	2	1
Median	33	5	3
3rd Quartile	131.5	14.5	11
Maximum Value	13,945	2,770	692

Table 3.5 shows that all variable values decrease from Git to IntelliMerge. For example, Git reports 159,005 evolutionary commits, whereas IntelliMerge identifies 81,812. Similar reductions are observed in the number of developers involved, refactoring regions, and refactorings in parents.

Table 3.5: Variables results for each tool

Variable	**Git**	**jFSTMerge**	**IntelliMerge**
Number of evolutionary commits	159,005	114,585	81,812
Number of refactoring regions in evolutionary commits	186,271	144,402	87,884
Number of developers in evolutionary commits	2,978	1,054	753
Number of refactorings in evolutionary commits	84,762	66,413	40,867
Number of refactorings in parents	67,127	24,739	8,492

We also analyzed the three tools by developing logistic regression models that incorporate key variables to deepen our understanding of merge conflict occurrence. When selecting independent variables, we ensured that their correlations with the dependent variable remained below 0.5, adhering to the standard for building reliable logistic regression models and avoiding multicollinearity. As a result, we excluded the variable "Number of developers" from Model VIII due to its high correlation.

We developed eight logistic regression models, as presented in Table 3.6. Coefficients marked with *** indicate statistically significant results. The results show that the variables analyzed exhibit lower statistical significance in models based on Git compared to the other tools.

3.5.1 RQ.1 - What factors are associated with the occurrence of merge conflicts?

To address RQ.1, we examined the merge conflicts through evolutionary commits. We also analyzed the three tools by developing logistic regression models by incorporating key variables to deepen our understanding of merge conflict occurrence, the dependent variable. Table 3.7 presents the general regression equation for each fitted logistic regression model.

According to the results in Table 3.6, particularly in Model II, the presence of evolution-

Table 3.6: Odds Ratios from regression considering conflict factors for each tool (***p<0.001; **p<0.01; *p<0.05)

	Model I	**Model II**	**Model III**	**Model IV**	**Model V**	**Model VI**	**Model VII**	**Model VIII**
Git tool								
Number of evolutionary commits		4.003				$1.170x10^{13}$	8.227	$8.237x10^{11}$
Number of refactoring regions					1.038***			1.002***
Number of developers	$3.827x10^{13}$							
Number of refactorings in evolutionary commits				$1.485\ x10^{11}$			1.254	1.257
Number of refactorings in parents			**9.894*** **			**1.951*** **	**1.951*** **	**1.928*** **
jFSTMerge tool								
Number of evolutionary commits		1.324***				1.214***	1.189***	1.175***
Number of refactoring regions					5.12010^{11}			$5.917x10^{10}$
Number of developers	**28.204*** **							
Number of refactorings in evolutionary commits				1.088***			1.020***	1.016***
Number of refactorings in parents			**11.937*** **			**8.065*** **	**7.916*** **	**5.562*** **
IntelliMerge tool								
Number of evolutionary commits		1.08***				1.055***	1.189***	1.175***
Number of refactoring regions					$1.23x10^{10}$			$2.352x10^{9}$
Number of developers	**7.701*** **							
Number of refactorings in evolutionary commits				1.024***			1.006***	1.005
Number of refactorings in parents			**8.840*** **			**7.249*** **	**7.172*** **	**4.798*** **

Table 3.7: Logistic regression model specifications.

Model	Specification
I	$logit(P(conflict)) = \beta_0 + \beta_1 Dev$
II	$logit(P(conflict)) = \beta_0 + \beta_1 EC$
III	$logit(P(conflict)) = \beta_0 + \beta_1 RefParent$
IV	$logit(P(conflict)) = \beta_0 + \beta_1 RefEvo$
V	$logit(P(conflict)) = \beta_0 + \beta_1 RefRegions$
VI	$logit(P(conflict)) = \beta_0 + \beta_1 EC + \beta_2 RefParent$
VII	$logit(P(conflict)) = \beta_0 + \beta_1 EC + \beta_2 RefParent + \beta_3 RefEvo$
VIII	$logit(P(conflict)) = \beta_0 + \beta_1 EC + \beta_2 RefParent + \beta_3 RefEvo + \beta_4 RefRegions$

Note: EC = Evolutionary Commits; Dev = Number of Developers; RefParent = Refactorings in Parent Commits; RefEvo = Refactorings in ECs; RefRegions = Refactoring Regions.

ary commits is associated with a 32.4% increase in the odds of conflicts in jFSTMerge and an 8% increase in IntelliMerge. However, in the Git-based models, this variable does not show a statistically significant relationship with conflict occurrence.

Detailed Analysis of the Most Relevant Models

Models I to V were initially fitted to evaluate the isolated effect of each variable on merge conflict occurrence. Subsequently, the models that provided the most meaningful explanations of the outcome variable are discussed in detail. Their interpretation should be performed in conjunction with the odds ratios reported in Table 3.6.

Model VI (Table 3.8) was developed to investigate the combined effect of the number of refactorings present in parent commits (RefParent) and the number of evolutionary commits (EC) on the occurrence of merge conflicts. The results indicate that the RefParent variable exerts a significant influence in all the tools analyzed, showing that the presence of refactorings increases the probability of conflicts during code integration.

In Git, RefParent showed a positive and statistically significant coefficient ($\beta = 0.6685$; p = 0.000019), while EC did not show statistical significance (p = 0.9942). On the other hand, in the jFSTMerge and IntelliMerge models, both variables were significant, indicating that both refactorings and the evolutionary history of the code contribute to the occurrence of conflicts. The effect of RefParent was particularly expressive in jFSTMerge ($\beta = 2.0876$;

p < 0.001), suggesting a strong association between refactorings and conflicts. These results reinforce the idea that refactoring is one of the main explanatory factors for merge conflicts and indicate that semi-structured tools are better able to capture the influence of software evolution on this phenomenon when compared to Git.

Table 3.8: Model VI Detailed

Tool	Variable	β	SE	95% CI	p-value
Git	Intercept	-2.9315	0.012	[-2.954, -2.909]	<0.001
Git	RefParent	0.6685	0.045	[0.581, 0.756]	0.000019
Git	EC	27.7888	3885.554	[-7587.757, 7643.334]	0.9942
jFSTMerge	Intercept	-3.2785	0.014	[-3.306, -3.251]	<0.001
jFSTMerge	RefParent	2.0876	0.027	[2.035, 2.140]	<0.001
jFSTMerge	EC	0.1944	0.004	[0.187, 0.202]	<0.001
IntelliMerge	Intercept	-3.9374	0.020	[-3.976, -3.899]	<0.001
IntelliMerge	RefParent	0.0538	0.002	[1.916, 2.046]	<0.001
IntelliMerge	EC	1.9809	0.033	[0.050, 0.058]	<0.001

In Table 3.9 we present Model VII, which was developed to simultaneously analyze the impact of refactorings present in parent commits (RefParent), refactorings performed in evolutionary commits (RefEvo), and the number of evolutionary commits (EC) on the occurrence of merge conflicts. The results show that the RefParent variable remained statistically significant in all tools analyzed, demonstrating that refactorings in parent commits represent the main factor associated with the occurrence of conflicts. In Git, only RefParent showed statistical significance ($\beta = 0.6685$; p = 0.00001), while RefEvo and EC did not contribute to explaining the conflicts. In contrast, in the jFSTMerge and IntelliMerge models, all variables were significant (p < 0.001).

The greatest effect was observed for RefParent ($\beta = 2.0690$ in jFSTMerge and $\beta = 1.9703$ in IntelliMerge), followed by RefEvo and EC, which also showed a positive association with the occurrence of conflicts. These results indicate that refactorings performed throughout the software's evolution exert a significant influence on merge conflicts, especially when analyzed by tools capable of interpreting structural changes in the code. Thus, Model VII expands on the evidence found in Model VI by demonstrating that both refactor-

ings in parent commits and those performed in evolutionary commits contribute to increasing the probability of conflicts, although the impact of refactorings in parent commits remains predominant.

Table 3.9: Model VII Detailed

Tool	Variable	β	SE	95% CI	p-value
Git	Intercept	-2.9315	0.012	[-2.954, -2.909]	0.0000
Git	RefParent	0.6685	3843.049	[0.581, 0.756]	0.00001
Git	RefEvo	0.2270	0.045	[-8781.620, 8782.074]	0.9999
Git	EC	27.4360	4480.616	[-7504.801, 7559.673]	0.9943
jFSTMerge	Intercept	-3.2860	0.014	[-3.313, -3.259]	<0.001
jFSTMerge	RefParent	2.0690	0.027	[2.017, 2.121]	<0.001
jFSTMerge	RefEvo	0.1739	0.001	[0.018, 0.023]	<0.001
jFSTMerge	EC	0.0203	0.004	[0.166, 0.181]	<0.001
IntelliMerge	Intercept	-3.9429	0.020	[-3.982, -3.904]	<0.001
IntelliMerge	RefParent	1.9703	0.033	[1.905, 2.035]	<0.001
IntelliMerge	RefEvo	0.0067	0.001	[0.005, 0.008]	<0.001
IntelliMerge	EC	0.0464	0.002	[0.042, 0.050]	<0.001

Finally, Model VIII, detailed in Table 3.10, was developed to simultaneously evaluate the effects of the number of evolutionary commits (EC), the number of refactorings present in parent commits (RefParent), the number of refactorings in evolutionary commits (RefEvo), and the number of refactored regions (RefRegions) on the occurrence of merge conflicts. The results demonstrate that RefParent remained the most relevant variable in all the tools analyzed. In Git, only RefParent showed statistical significance ($\beta = 0.6555$; p < 0.001), while EC, RefEvo, and RefRegions did not significantly contribute to explaining the occurrence of conflicts. In contrast, in the jFSTMerge and IntelliMerge models, the variables EC, RefParent, and RefEvo showed a positive and statistically significant association with merge conflicts (p < 0.0001). The largest effect was again observed for RefParent ($\beta = 1.7161$ in jFSTMerge and $\beta = 1.5726$ in IntelliMerge), reinforcing the role of refactorings in parent commits as the main explanatory factor. On the other hand, RefRegions did not show statistical significance in any of the semi-structured models, indicating that the number of

regions affected by refactorings does not have an independent effect when other variables related to code evolution are considered. These results suggest that the occurrence of conflicts is strongly associated with the presence of refactorings and the evolutionary history of the software, especially in tools capable of analyzing structural aspects of the code.

Table 3.10: Model VIII Detailed

Tool	Variable	β	SE	95% CI	p-value
Git	Intercept	-2.9314	0.012	[-2.954, -2.909]	0.0000
Git	EC	27.4379	3843.027	[-7504.757, 7559.633]	0.994
Git	RefParent	0.6555	0.045	[0.566, 0.745]	0.0000
Git	RefEvo	0.2277	4492.585	[-8805.077, 8805.532]	0.9999
Git	RefRegions	0.0017	0.001	[-0.000009, 0.004]	0.063
jFSTMerge	Intercept	-3.2786	0.014	[-3.306, -3.251]	<0.0001
jFSTMerge	EC	0.1619	0.004	[0.155, 0.169]	<0.0001
jFSTMerge	RefParent	1.7161	0.029	[1.659, 1.773]	<0.0001
jFSTMerge	RefEvo	0.0161	0.001	[0.013, 0.019]	<0.0001
jFSTMerge	RefRegions	24.8038	0.0001	[-0.0002, 0.0002]	0.9983
IntelliMerge	Intercept	-3.9483	0.020	[-3.987, -3.909]	<0.0001
IntelliMerge	EC	0.0410	0.002	[0.037, 0.045]	<0.0001
IntelliMerge	RefParent	1.5726	0.038	[1.497, 1.648]	<0.0001
IntelliMerge	RefEvo	0.0054	0.001	[0.004, 0.007]	<0.0001
IntelliMerge	RefRegions	25.1622	0.0001	[-0.0002, 0.0002]	0.998

Logistic Regression Models from Git Results

Our results, based on odds ratios, show that the variable `Number of refactorings in parents` (left or right) presents the strongest association with merge conflicts when considered individually. In Model V, higher values of `Number of refactoring regions` are also associated with an increased likelihood of merge conflicts. However, this variable accounts for a relatively small proportion of the variation (3.8%). The remaining variables show limited or non-significant associations with merge conflicts in the models based on Git.

The unusually large odds ratio was observed in an intermediate model that included the `Number of Developers` variable. Although this variable showed moderate correlation with other explanatory variables, its inclusion did not result in statistically significant effects and led to poorer coefficient estimates and overall model quality. Therefore, we excluded it from the final model specification. Consequently, it was excluded from the final models (Models V to VIII), which are the basis for our reported findings.

Logistic Regression Models from jFSTMerge Results

Similarly to Git results, we developed logistic regression models for jFSTMerge data. According to our results, the variables `Number of evolutionary commits`, `Number of developers`, `Number of refactorings in evolutionary commits`, and `Number of refactorings in parents` were relevant variables to the presence of merge conflicts.

We identified Models VII and VIII as promising based on their statistical performance. To compare model fit, we used explained deviance, which reflects the proportion of variation accounted for by the model. Model VII presents the highest deviance explained (27%), while Model III shows the lowest value (14.16%).

In Model III, which includes only the `Number of Refactorings in Parents` as the independent variable, the presence of refactorings in parent commits is associated with substantially higher odds of merge conflicts, with odds ratios of 9.894 for Git, 11.937 for jFSTMerge, and 8.840 for IntelliMerge. These values correspond to increases of 36%, 37%, and 14% in the odds of merge conflict occurrence for Git, jFSTMerge, and IntelliMerge, respectively. Furthermore, the variable remained statistically significant in the multivariate models, suggesting that its relationship with merge conflicts is robust even when additional factors are considered. In contrast, the `Number of Refactorings in Evolutionary Commits` shows a comparatively small effect.

Logistic Regression Models from IntelliMerge

Regarding this tool, seven of the eight developed models present statistically significant odds ratios, with the most significant values highlighted in bold. Model II shows a relatively small effect size, with the `Number of evolutionary commits` variable presenting a mod-

est association with merge conflicts. Similarly, the variable `Number of refactorings in evolutionary commits` in Model IV shows a comparatively small effect. Models I, III, VI, VII, and VIII present stronger associations.

According to the explained deviance values, Model I presents the highest value (34.31%), followed by Models VII (20.36%) and VI (16.40%), while Model III shows the lowest value (10.70%). In Model III, the `Number of Refactorings in Parents` is strongly associated with the occurrence of merge conflicts, with an odds ratio of 8.840. This indicates a substantial increase in the odds of conflict occurrence when refactorings are present in parent commits. The `Number of developers` variable also shows a significant association with merge conflicts in IntelliMerge when used as the sole variable in a regression model, indicating a substantial increase in the odds of merge conflicts.

RQ.1 - Summary of results: The 'Number of refactorings in parent commits' plays a significant role in conflicts reported by all three tools, with statistically significant results and a strong effect size.

3.5.2 RQ.2 - Which types of refactoring are most associated with merge conflicts?

Across Git, jFSTMerge, and IntelliMerge, we identified common refactorings involved in merge conflicts. A total of 62 refactoring types were detected in evolutionary commits, with an intersection of 21 refactoring types shared across the three tools.

Additionally, we identified a consistent top ten of refactoring types across Git, jFSTMerge, and IntelliMerge. Table 3.11 presents these refactoring types and their relative frequencies considering all refactorings detected in evolutionary commits by each tool. Regarding the proportion of conflicting scenarios involving refactorings, we observed that 43% of conflicting merge scenarios in Git involve refactorings. For jFSTMerge and IntelliMerge, these proportions are 32% and 37%, respectively.

RQ.2 - Summary of results: Rename Parameter, Rename Method, Rename Variable, Move Class, and Extract Method account for over 60% of the refactoring types involved in conflict history.

Table 3.11: Top ten refactoring types involved in merge conflicts by tool

Refactoring Type	**Git**	**jFSTMerge**	**IntelliMerge**
Rename Parameter	14.29%	14.81%	17.25%
Rename Method	13.63%	13.83%	17.08%
Rename Variable	13.04%	14.15%	8.37%
Move Class	12.21%	11.07%	11.43%
Extract Method	7.56%	7.58%	7.52%
Move Method	7.42%	7.14%	6.34%
Rename Attribute	3.87%	3.82%	3.97%
Move Attribute	3.87%	3.82%	3.97%
Pull Up Method	3.83%	3.85%	3.70%
Extract Variable	3.39%	3.35%	3.13%

3.6 Discussion

When developing and analyzing these models, our purpose was explanatory rather than predictive. Therefore, we focus on the strength of associations and model explanatory power, assessed through odds ratios and deviance, rather than predictive performance metrics.

By observing the models, refactorings are consistently strongly associated with merge conflicts across the three merge tools. Git, jFSTMerge, and IntelliMerge all present statistically significant odds ratios, indicating a consistent relationship between refactorings and conflict occurrence. Among them, jFSTMerge shows the highest odds ratio (11.94), followed by Git (9.89) and IntelliMerge (8.84) in Model II. Notably, the effects of refactorings remain strong and statistically significant in Model VIII, indicating consistency across models.

While all tools exhibit sensitivity to refactorings, the magnitude and stability of the effects vary across tools, with jFSTMerge showing more pronounced estimates. These differences suggest that the relationship between refactorings and merge conflicts is not uniform across merge techniques, but may reflect differences on how each tool handles structural changes.

The other factors analyzed present either modest effects or inconsistent statistical significance. The number of developers appears as a relevant factor, but less consistently than

refactorings. Although statistically significant in some models, evolutionary commits and refactoring regions show relatively small effect sizes. Taken together, these results indicate that no single factor fully explains merge conflicts. Instead, they suggest that both the temporal dimension of changes (evolutionary commits over time) and their structural characteristics (refactorings) should be considered jointly when analyzing conflict occurrence. By exploring the relationship between evolutionary commits and refactoring activity we found no clear monotonic relationship between the two variables. Reinforcing that evolutionary commits may capture a different aspect of conflict evolution, reflecting how changes accumulate throughout the development history but does not appear to present an association with the amount of refactoring activity itself. Consequently, it represents a different dimension of conflict evolution and should not be applied to infer refactoring intensity.

Another relevant finding concerns the proportion of conflicting scenarios involving refactorings. Prior work reports that approximately 22% of merge scenarios with conflicts involve refactorings when using Git tool [50]. In our dataset, this proportion varies across tools: 43% for Git, 32% for jFSTMerge, and 37% for IntelliMerge. These differences reinforce that the observed relationship between refactorings and conflicts may vary depending on the merge strategy. Additionally, some conflicts involving refactorings remain unresolved even when using IntelliMerge, as observed through manual inspection of conflict regions. These cases may be related to refactorings within method bodies or to divergent changes that are difficult to reconcile automatically [35].

Our results also suggest that not all refactorings lead to conflicts. However, refactoring types that propagate changes across multiple elements, such as renaming, moving, or extracting code, are more frequently associated with conflicts. These transformations often require coordinated updates across different parts of the system, increasing the likelihood of overlapping changes. This observation is consistent with prior studies that report frequent use of such refactorings in practice [75], [40]. In our dataset, we detected 62 refactoring types, representing a broader coverage compared to previous versions of RefactoringMiner [50], [79], [80], which may contribute to the higher proportion of refactoring-related conflicts observed.

Fig. 3.5 illustrates a merge conflict scenario[2] observed in our study across the three tools. Our findings indicate that the Rename Parameter refactoring is the most frequently

[2]Commit from error-prone project on GitHub: https://github.com/google/error-prone/commit/e55275fc

involved refactoring type in merge conflicts, as highlighted in 3.11. In this example, the parameter `suppressable` was renamed to `suppressible` in the right commit, resulting in changes that the tools could not manage. A tool that detects and highlights the effects of refactorings could greatly assist developers in making informed decisions on how to resolve conflicts.

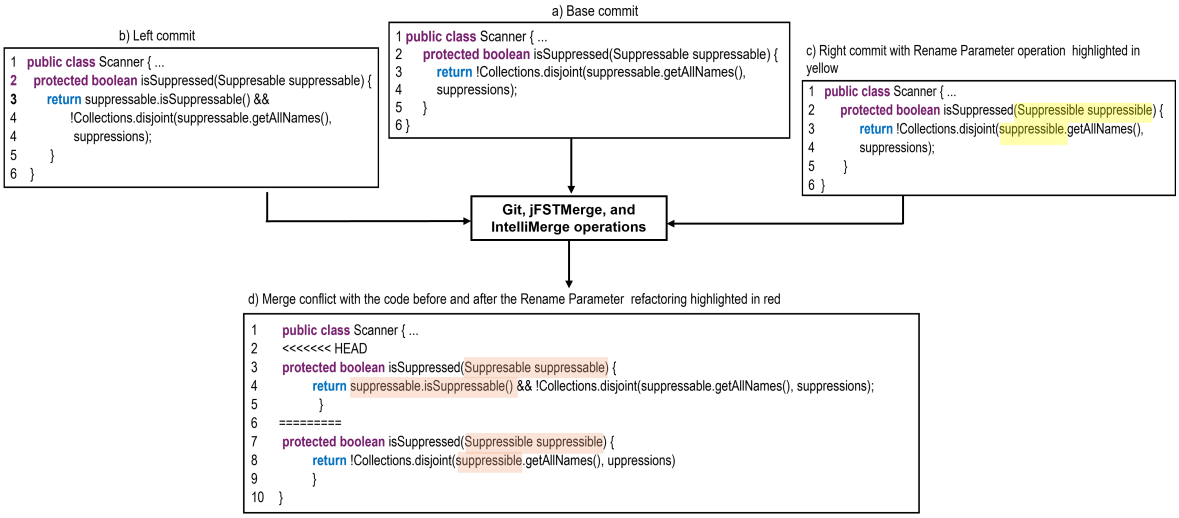


Figure 3.5: Merge conflict reported by all three tools

When comparing the three merge tools, we observe both similarities and differences. In terms of similarities, all tools confirm the strong association between refactorings and merge conflicts, particularly those occurring in parent commits. The results consistently show an increased likelihood of conflicts, reflected in high odds ratios. The most frequent refactoring types associated with conflicts are also consistent across tools, including Rename Parameter, Rename Variable, and Extract Variable.

As refactorings emerged as a significant variable in our logistic regression models, we also explored the time to merge conflicts considering this variable. This metric approximates the branch integration time, that is, the elapsed time between the most recent parent commit and the merge commit that integrates the branch. Consequently, we examined whether refactorings increase or decrease the time to merge contributions during the changes that lead to a merge conflict. Therefore, we investigated whether refactorings in evolutionary commits are associated with the time to merge conflicts.

Initially, we divided the 17,598 conflicting merge scenarios obtained from Git history into two sets: scenarios with refactoring and scenarios without refactoring. Then, we calculated

the time to merge by subtracting the timestamp of the most recent parent of the merge commit from the timestamp of the merge commit itself (the conflicting commit), using only data extracted from Git logs. This metric approximates the time required to integrate and resolve a conflicting merge scenario.

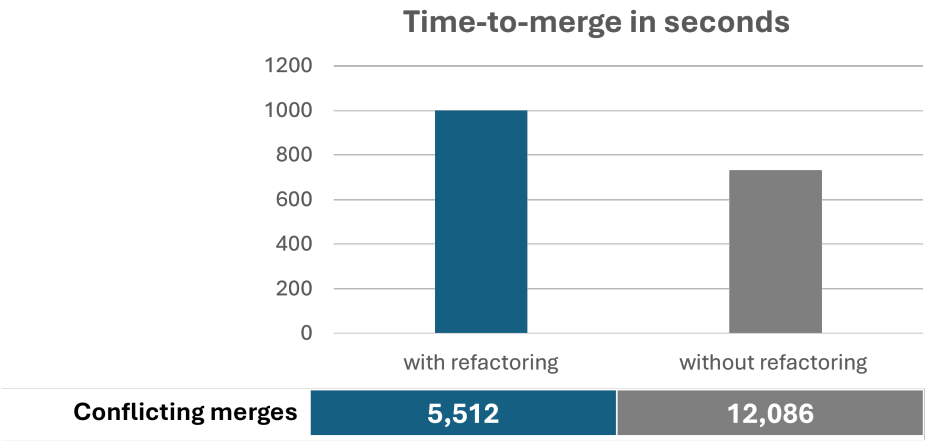


Figure 3.6: Time to merge conflicting scenarios with and without refactorings involved (median values)

The results in Figure 3.6 indicate that scenarios involving refactorings exhibit a median merge time of 1,000 seconds, or approximately 16.7 minutes, surpassing scenarios without refactorings, which have a median merge time of 733 seconds, or approximately 12.2 minutes. Although scenarios without refactorings accounted for more than 45% conflicting merge scenarios, we observed longer merge times when refactoring was involved. We also applied the Mann-Whitney U test to compare if there is a statistically significant difference in time to merge between groups with and without refactorings. Our results suggest that refactorings in parent commits are significantly associated with time-to-merge ($U - statistic = 86494520.0$, $p - value < 0.01$, $effect - size = 0.35$).

3.7 Threats to Validity

In this section, we discuss possible threats to the validity of our study. We organize these threats into construct, internal, and external validity, describing how each may affect our findings and the steps taken to mitigate them.

3.7.1 Construct Validity

This analysis is subject to several threats to validity. A limitation of our study is the reliance on tools with varying rates of false positives and negatives. For example, Git reported 55.47% false positives and 1.14% false negatives, while IntelliMerge showed 75.44% false positives and 7.57% false negatives in a previous study, indicating that Git may overestimate conflicts and IntelliMerge may underestimate them [35]. Similarly, jFSTMerge significantly reduces false positives compared to Git (30.21% vs. 43.47%) but has a higher false negative rate (20.60% vs. 9.62%), meaning some conflicts may be missed [24].

Despite these differences, the three tools consistently indicate a strong relationship between refactorings and merge conflicts. Comparing the results across multiple tools also helps reduce the influence of tool-specific biases, providing additional confidence in the observed patterns. The effect of the `Number of developers` variable should be interpreted with care, as incomplete information in Git histories may have influenced its estimated importance. Although IntelliMerge is known to exhibit relatively high false positive and false negative rates, its results are largely consistent with those obtained using jFSTMerge. This suggests that IntelliMerge is still capable of identifying relevant explanatory variables, even if its conflict detection is less accurate.

Our study relies on RefactoringMiner and the Refactoring Analysis Tool to identify refactorings, reconstruct merge scenarios, and extract history information. Although these tools have been evaluated and adopted in previous studies, inaccuracies such as false positives, false negatives, or other errors may affect the collected data. To mitigate this threat, we manually inspected a sample of the results obtained from five repositories and verified the consistency of the extracted information considering the three tools explored in this study. The outcomes of the manual inspection were consistent with the automated results, increasing our confidence in the reliability of the collected dataset.

Finally, we measured merge time solely based on Git execution logs, which may not fully capture the manual effort developers invest in resolving merge conflicts. For instance, developers may spend time understanding the code, reviewing changes, or coordinating with teammates before executing a merge. Considering this, we do not name this metric `merge effort`, in an attempt to not generalize its interpretation.

3.7.2 Internal Validity

Threats to internal validity stem primarily from the use of automated refactoring detection tools. However, Refactoring Miner 2.0 [79], which we employed, has been shown to achieve high precision and recall in detecting refactorings. Additionally, previous studies have evaluated its performance, confirming its reliability [50].

To further mitigate potential biases, we manually analyzed a sample of conflicting scenarios. While manual analysis introduces the possibility of human error, we employed a cross-validation process by verifying refactorings detected by RefactoringMiner against code resulting from merge conflicts in Git, jFSTMerge, and IntelliMerge. This triple-checking process enhances reliability by reducing misclassification risks and improving our understanding of conflict causes.

3.7.3 External Validity

Our study focuses on open-source Java projects, which could limit the generalization of our findings to other programming languages and development environments. However, Java's widespread use in large-scale software development and research supports its relevance for studying software evolution and merge conflicts. Future work could extend this analysis to other languages to assess the broader applicability of our findings.

Finally, our study specifically examines merge conflicts using Git, jFSTMerge, and IntelliMerge. While these tools cover different merging strategies, our results may not generalize to other merge tools. Future studies could explore additional tools and conflict resolution strategies to further validate our conclusions.

3.7.4 Statistical Conclusion Validity

Our statistical analyses assume independence among merge scenarios. However, some scenarios originate from the same project and may share characteristics such as developers' programming styles, commit frequency, and the nature of merge conflicts, which may introduce correlation among observations. Additionally, although several regression models were evaluated, we mitigated the risk of spurious findings by reporting effect sizes, confidence intervals, and selecting models based on deviance and statistical significance. Finally, the

large number of analyzed merge scenarios increased the statistical power of our analyses, reducing the likelihood of failing to detect existing effects.

3.8 Concluding Remarks

In this chapter, we conducted a comprehensive tool-based study of merge conflicts, focusing on the factors that contribute to their occurrence. Our analysis of Git, jFSTMerge, and IntelliMerge revealed both similarities and differences in how these tools relate to conflict detection. The results underscore that refactorings are consistently associated with merge conflicts across tools. While the number of evolutionary commits emerged as a statistically significant variable, it presented a relatively small effect size. Similarly, the extent of refactoring regions showed a limited association with conflict occurrence. In contrast, the presence of refactorings in parent commits is strongly associated with merge conflicts, indicating that structural changes play an important role in these scenarios.

The mere presence of refactorings is a strong indicator of potential merge conflicts. This finding motivate further investigation into the relationship between refactorings and merge conflicts, explored in the next chapter.

Chapter 4

Towards a Deeper Understanding of Refactorings and Merge Conflicts

Refactorings are essential for improving code readability, structure, and maintainability, and developers often apply them before or during feature implementation. Given the inevitability of merges in collaborative environments that rely on version control systems, refactorings may introduce additional challenges. Because they alter the structure of the code, refactorings can interfere with textual merge strategies. Furthermore, merges involving structural changes are generally more difficult to resolve automatically.

Prior studies have investigated the relationship between refactorings and merge conflicts [32], [33], [35], [50]. However, these studies primarily focus on improving merge processes or detecting refactoring-related conflicts in merge scenarios, without quantitatively modeling this relationship. In this chapter, we go beyond existing work by statistically modeling the association between refactorings and merge conflicts while controlling for variables related to change magnitude within merge scenarios across Git, jFSTMerge, and IntelliMerge.

We further investigate the role of refactorings in merge conflicts, with particular attention to refactoring tactics. Especially, we compare pure refactorings, which follow performs only structural transformations, with floss refactorings, which combine structural changes with other modifications. We classify merge scenarios according to these tactics and compare the types of refactorings across the three tools. Finally, through statistical analysis, we examine the relationship between refactorings and time-to-merge in conflict scenarios.

4.1 Study Goal and Research Questions

To better understand merge conflicts based on the results reported in Chapter 3, we extend prior investigations on the association between refactorings and merge conflicts by modeling this relationship while controlling for change magnitude in merge scenarios. We pose the following research questions to this study:

- **RQ.3 - Is refactoring independently associated with merge conflicts?**

  We first investigate the association between refactorings and merge conflicts using Git results as a baseline. To examine this question, we fit logistic regression models to assess whether refactoring is associated with merge conflicts or whether the size of changes in a commit acts as a confounding variable. We use the same dataset of merge conflicts previously employed, which includes information on total lines of code churn and the total number of files changed in commits, resulting in 169,980 merge scenarios from GitHub.

- **RQ.4 - Does the association between refactoring and merge conflicts vary across merge tools?**

  Building on the results of RQ.3, we examine whether the association between refactorings and merge conflicts varies depending on the merge tool used. We extend the dataset from RQ.3 by incorporating results from jFSTMerge and IntelliMerge for the same merge scenarios and include the merge tool as a dummy variable. This results in a dataset of 509,940 merge scenarios. We then fit logistic regression models with interaction terms to test whether the relationship between the predictor variable and the binary outcome depends on the merge tool.

- **RQ.5 - Which refactoring tactic, pure or floss, is more associated with merge conflicts?**

  To answer this question, we first classify the refactorings involved in merge scenarios collected to answer the RQ.3 as either pure or floss and categorize commits accordingly. This classification is performed automatically based on evolutionary commits from our dataset.

- **RQ.6 - How are merge conflicts distributed across refactoring tactics (pure vs. floss), considering the types of refactorings involved?**

  To answer this question, we categorize identified refactorings by type, building upon the pure and floss classification obtained in RQ.3.

- **RQ.7 - Do pure and floss refactorings differ in merge conflict resolution time?**

  We examine whether different refactoring tactics are associated with differences in merge conflict resolution time. This question focuses on analyzing temporal differences in merge conflict resolution, complementing the previous analyses of occurrence and association.

Our goal is to determine whether refactoring is an independent factor associated with conflict occurrence or merely related to the size of modifications in commits. To this end, we incorporate two variables related to change magnitude as control variables, analyze how this association varies across merge tools (Git, jFSTMerge, and IntelliMerge), identify the refactorings most frequently involved in merge conflicts, and examine the relationship between refactoring tactics and merge conflicts by observing both the frequency of each tactic in conflicts and the time required to merge conflicting commits under each tactic. We structure our investigation in five steps, progressively examining the association, consistency, granularity, and temporal aspects of the relationship between refactorings and merge conflicts.

4.2 Approach to Analyze Refactoring Impacts on Merge Conflicts

In this Section, we present the approach applied to analyze refactorings in merge conflicts regarding association between these two variables, variation of the effect of this association by tool, the decomposition of refactoring effects by refactoring type, and behavioral and temporal consequences of refactorings on merge conflicts according to the tactic applied. The scripts and datasets applied are publicly available [1].

We applied the four steps illustrated in Figure 4.1 for this study.

[1] https://github.com/lqleal/refactoringsinmerges

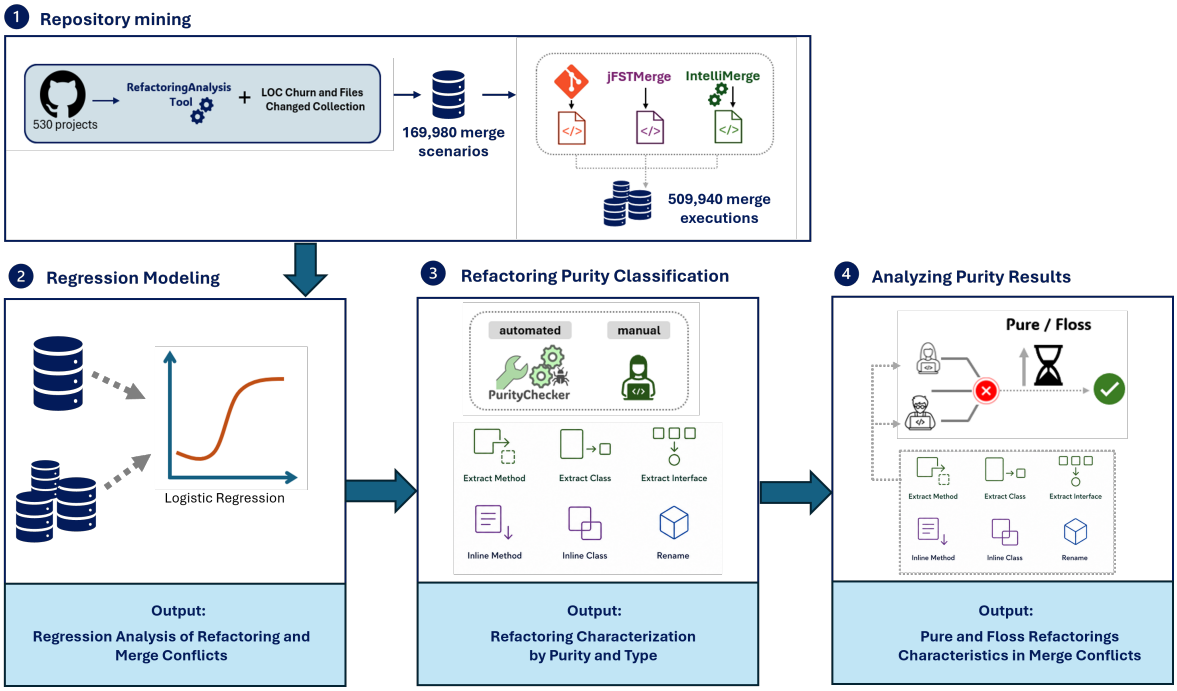


Figure 4.1: Steps to analyze refactorings

In Step 1, we collected data from GitHub repositories, totaling 169,980 merge scenarios. Using these scenarios, we reconstructed the merges by applying the jFSTMerge and IntelliMerge tools, resulting in 509,940 merge executions, which serve as input to the following steps. In Step 2, we performed logistic regression analyses to investigate the relationship between refactorings and merge conflicts while controlling for variables related to the magnitude of the changes as confounders.

In Step 3, we used the data obtained in Step 1 as input to the PurityChecker tool to collect information regarding the presence of pure and floss refactorings in the merge scenarios. In this step, we also conducted a manual analysis to validate the results produced by PurityChecker and assessed refactoring purity based on refactoring types.

Finally, in Step 4, using the same dataset generated from the purity analysis in Step 3, we analyzed the estimated time to merge conflicts with respect to the purity of refactorings.

4.2.1 Variables

We extend prior investigations on the association between refactorings and merge conflicts by modeling this relationship while controlling for change magnitude in merge scenarios. To

this end, we incorporate change magnitude as a control variable, analyze how the association varies across merge tools (Git, jFSTMerge, and IntelliMerge), identify the refactorings most frequently involved in merge conflicts, and examine the relationship between refactoring tactics and merge conflicts. The variables used to address this goal capture the scope of changes applied during merge scenarios and are summarized in Table 4.1.

Table 4.1: Variables applied in this study

Variable	**Description**
Merge conflict occurrence *(dependent variable)*	A binary variable representing the presence or absence of a merge conflict in Java files within a merge scenario, where 1 indicates the occurrence of at least one merge conflict and 0 indicates no conflict.
LOC churn	The total number of lines added and deleted in a merge scenario.
Files changed	The number of files modified in a merge scenario.
Presence of refactoring in parents	A binary variable indicating whether refactorings are present in the parent commits of a merge. The value is 1 if at least one parent commit contains a refactoring, and 0 otherwise.

4.2.2 Statistical Analysis

In this study, we conducted a two-part statistical analysis using the collected and organized data obtained from executing the merge tools Git, jFSTMerge, and IntelliMerge. First, we computed descriptive statistics to characterize the general behavior of the data. Then, to gain deeper insights, we applied logistic regression models.

We used a binary dependent variable representing the presence of merge conflicts in Java files (*co*), and a set of continuous and categorical independent variables, including LOC churn, files changed, and the presence of refactorings in parent commits (f) [36]. The regression model follows the formulation presented in Equation 3.1 in Section 3.4.2. The relationship between merge conflicts and the variables listed in Table 4.1 was analyzed using the model defined in Equation 4.1:

$$m = \text{glm}(co \sim f,\ \text{family} = \text{binomial},\ \text{data} = iv) \tag{4.1}$$

The dataset used in this analysis, referred to as *iv*, is a generic name representing three files containing the variables listed in Table 4.1, corresponding to the results of Git, jFSTMerge, and IntelliMerge executions.

To investigate the relationship between refactorings and merge conflicts (RQ.1), we developed three logistic regression models. The first model evaluates the isolated association between refactorings and merge conflicts. The second model incorporates variables related to change magnitude, namely LOC churn and Files Changed. In the third model, we exclude the variable Files Changed due to its moderate correlation with LOC churn ($r \geq 0.5$), which could introduce multicollinearity. This decision aims to improve model stability and avoid biased estimates.

For RQ.2, we extend the analysis by incorporating the merge tool as a variable and introducing interaction terms. *Refactoring * Tool* represents both the main effects and their interaction, and f denotes control variables related to change magnitude. While models based on a single tool capture average effects, the interaction model allows us to examine whether the association between refactorings and merge conflicts varies across tools. Equation 4.2 presents the logistic model fitted. This allows us to assess whether the effect of refactorings differs significantly across merge tools.

An interaction model is used when the effect of a predictor on the outcome is not constant but depends on the level of another variable [36]. In this context, interaction terms enable us to capture tool-specific effects, meaning that the impact of refactoring on merge conflicts may differ depending on the merge tool used.

$$m = \text{glm}(co \sim \textit{Refactoring} * \textit{Tool} + f,\ \text{family} = \text{binomial},\ \text{data} = \text{iv}) \tag{4.2}$$

We applied a chi-square test to analyze the association between pure and floss refactorings and merge conflicts. The chi-square test is a statistical method used to evaluate whether there is a significant difference between observed and expected frequencies of categorical variables. We categorized merge scenarios in our dataset as PURE, FLOSS, or NONE, and cross-tabulated these groups with the binary variable `conflict` (0 for non-conflicting and 1 for conflicting scenarios). This test was used to address RQ.3.

4.2.3 Analyzing Pure/Floss Refactoring in Merge Conflicts

To enhance our understanding of the relationship between refactorings and merge conflicts, we performed an automated classification of refactorings involved in merge conflicts, according to the purpose of their implementation. As a validation step, we also conducted a manual analysis on a sample of the data used in the automated classification.

The automated classification of refactorings was performed using the PurityChecker API in conjunction with RefactoringMiner 3.0. The analysis is conducted on individual commits by considering the differences between a commit and its parent. First, RefactoringMiner is executed to detect refactorings introduced in the analyzed commit. Then, PurityChecker classifies these refactorings as pure or floss, depending on whether they occur in isolation or together with other types of changes.

To validate the results of the automated classification, we conducted a manual analysis following the three steps presented in Figure 4.2. We collected a sample of evolutionary commits involving refactorings and compared each commit with its immediate predecessor. In some cases, the predecessor of a commit with refactoring may not itself involve refactorings, as illustrated in Figure 2.2 by `cmmt1`. Our goal was to assess whether the refactorings involved in conflicts were correctly classified according to their intended category (pure or floss).

Step 1: Collect test pairs

We examined evolutionary commits to identify test pairs. For example, considering the Figure 4.3: `p1` and `p2` represent the parent commits, `m` is the result of merging `p1` and `p2`, and `base` is the common ancestor. The commits located between base and `p1` as well as

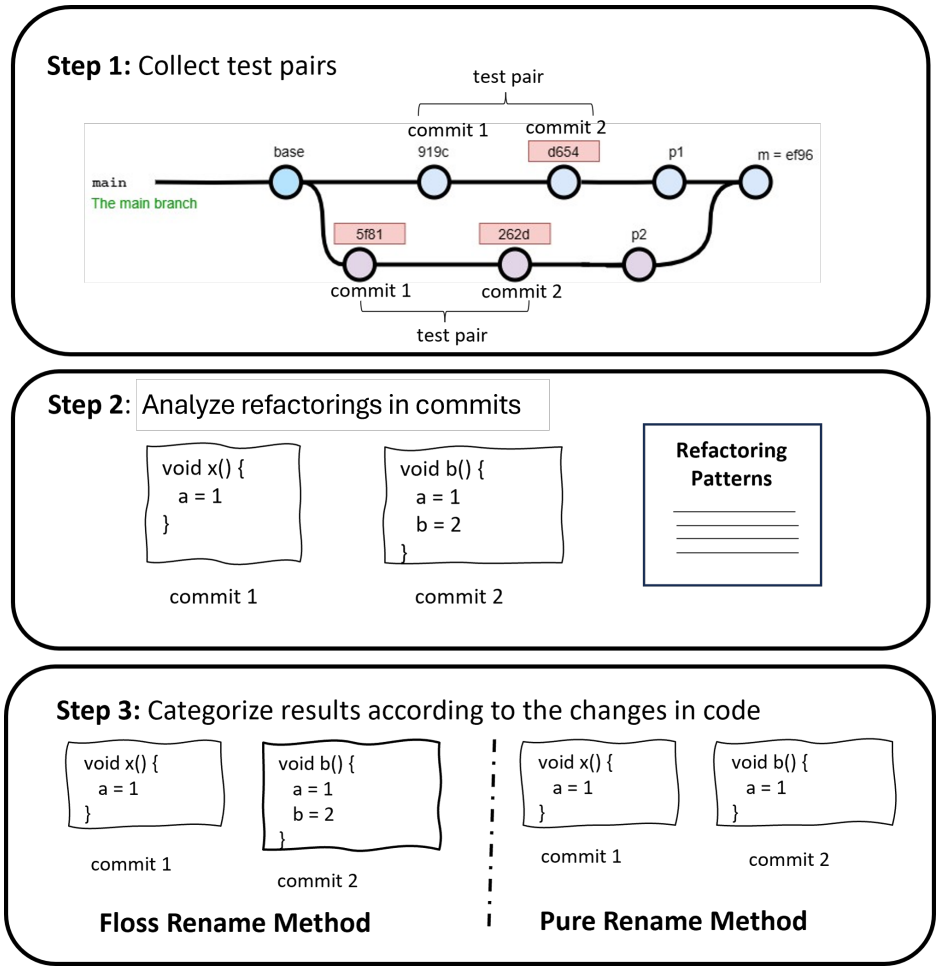


Figure 4.2: Steps applied in manual analysis

base and `p2` are the evolutionary commits.

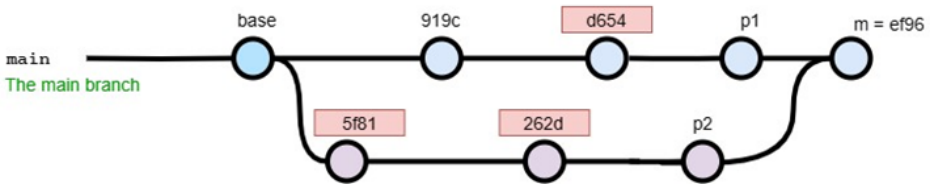


Figure 4.3: Analyzed elements in a merge scenario

Commits containing refactorings are indicated by red boxes enclosing the commit number. According to the constraints presented before, commits (919c → d654) and (5f81 → 262d) are valid test pairs.

Step 2: Analyze refactorings in commits

Considering Step 2 presented in Figure 4.2, we analyzed Java classes by observing the transition between the previous and the current commit, looking for refactorings in the code. These refactorings were expected to follow established refactoring patterns. In this study, refactoring patterns are grounded in [37]. For automatic detection, we rely on RefactoringMiner that implements a subset of this patterns. This is a complementary approach since the catalog in [37] provides a comprehensive definition of the refactoring types and RefactoringMiner focuses on the detection of patterns through code analysis. Deviations from these patterns were interpreted as evidence of additional functional changes, characterizing Floss Refactoring.

Step 3: Categorize results according to code changes

In this step, we classified the observed refactorings. We analyzed whether the refactorings followed their expected definitions [37] or whether additional edits were introduced in the refactored files. Deviations from the refactoring definitions described in the previous section were considered indicative of floss refactorings.

Examples of operations classified as floss refactorings include the introduction, removal, or modification of statements within a method or class body, changes to conditional expressions, and the addition or removal of methods beyond the scope of the refactoring definition [37].

We selected a sample of commits and manually classified them as involving either pure or floss refactorings. Accordingly, we labeled `Commit 2` as containing either a pure or floss refactoring, as illustrated in Step 3 of Figure 4.2.

Finally, we computed and analyzed the number of commits classified as pure or floss. The classification results were subsequently reviewed to identify and resolve any discrepancies.

Manual Analysis

To validate the results of PurityChecker, we randomly extracted a sample comprising pairs of evolutionary commits on Git results. Our population had 11,187 evolutionary commits with refactoring collected in Git results, as shown in Table 3.3.

To guarantee representativeness of the manually analyzed sample, we calculated the sample size based on the population proportion of evolutionary commits collected in Git tool results. We applied the sample formula using the population proportion (p_0), the desired confidence level (95% which corresponds to $z \cong 1.96$, and E for the tolerated error percentage), as we can see below:

$$n = \frac{z^2 \cdot p_0(1 - p_0)}{E^2}$$

where:

$$p_0 = \frac{11,187}{159,005} = 0.0703$$

$$z = 1.96$$

$$E = 0.05$$

Considering a 95% confidence interval and a 5% marging of error, the ideal sample size is 91 commits to ensure representativeness with 95% confidence and a 5% margin of error. We randomly extracted a sample of 100 pairs of commits and analyzed them according to the steps presented in Figure 4.2.

4.2.4 Metrics Related to Change Magnitude

To understand the impact of change magnitude on merge conflicts, we include two metrics that capture the size of modifications in merge scenarios: lines of code (LOC) churn and the number of files changed. `LOC churn` measures the amount of code added and removed in a commit and has been used as an indicator of change size and development risk [82], [58]. Similarly, the `Number of files changed` reflects how broadly modifications are distributed across the codebase and has been associated with coordination complexity and integration challenges [82], [20].

We collected these metrics across all merge scenarios in our dataset using a shell script. This script re-executes the Git tool and retrieves information from each merge commit using the `git show --numstat` command. The output of this command is used to compute (i) the number of files changed, given by the number of lines returned per execution, and (ii) the LOC churn, calculated as the sum of added and deleted lines reported in each line of the output.

These metrics are adopted in empirical software engineering studies as proxies for change magnitude and have been shown to influence software quality and integration outcomes [82], [58], [20]. In the context of merge conflicts, controlling for change magnitude is essential, as larger or more dispersed changes may independently increase the likelihood of conflicts. Therefore, we include `LOC churn` and the `Number of files changed` as control variables to better isolate the association between refactorings and merge conflicts.

4.2.5 Metric Related to the Purity of Refactorings in a Conflicting Merge Scenario

We define the `floss ratio` of a merge conflicting scenario as the proportion of refactoring operations performed in To validate that also include non-refactoring changes. Refactoring operations were classified using the PurityChecker API. The ratio is computed at the merge scenario level by aggregating all refactoring operations associated with each merge conflict instance.

We compute the `floss ratio` for each conflicting scenario as follows (Equation 4.3):

$$\text{floss ratio} = \frac{\sum \text{floss refactorings in a conflicting merge scenario}}{\sum \text{refactorings in a conflicting merge scenario}} \tag{4.3}$$

Merge scenarios with `floss_ratio` equal to zero are classified as `pure`, whereas those with values greater than zero are classified as `floss`. Values between 0 and 1 indicate scenarios with a mixture of pure and floss refactorings, capturing the degree of refactoring interleaving.

4.3 Results

In this section, we present the results of this study by addressing five research questions. We begin by examining whether refactorings are independently associated with merge conflicts (RQ.1). We then analyze how this association varies across the Git, jFSTMerge, and IntelliMerge tools (RQ.2). Next, we investigate how merge conflicts are distributed across refactoring tactics (pure vs. floss), which refactoring types are more frequently associated with each tactic (RQ.4). Finally, we explore how refactoring tactics (pure vs. floss) relate to

merge conflict resolution time (RQ.5).

4.3.1 RQ.3 - Is refactoring independently associated with merge conflicts?

Considering the results obtained in a prior exploratory study and summarized on the previous section, the presence of refactorings is associated with an increase of merge conflicts. Notably, the large number of refactorings identified in parent commits (Table 3.5) suggests that refactoring activity is a common characteristic of merge scenarios, rather than an exceptional event. This observation motivates the investigation of refactorings as a potential factor associated with merge conflicts.

This research question re-evaluates associations under controlled change magnitude to address potential confounding effects. To analyze whether refactoring is a strongly associated factor in the occurrence of merge conflicts, we applied our previous dataset of 173,258 merges scenarios to collect variables that control change size in merges. This allows us to assess whether the association between refactoring and merge conflicts persists after controlling for the size of code changes. To do so, we collected two variables (`loc_churn_total` and `files_changed_total`), added them to our previous Git dataset, and produced logistic regression models to answer our question by analyzing Git conflicts occurring in Java files.

To answer RQ.1, first, we wanted to know the raw association to understand if merges involving refactoring exhibit higher conflict rates than merges without refactoring. Then, we increased our model to observe if refactoring remains associated with merge conflicts after controlling for change size.

Association between merge conflicts and refactorings

To deepen our analysis, we examined the relationship between `conflict_java` and `any_refactoring` variables. Initially, we performed a descriptive analysis by constructing a contingency table. Contingence table or cross-tabulation is commonly used as an exploratory analysis before logistic regression to examine the conditional probabilities between

Presence of Refactoring	No Conflict	Conflict	Conflict Rate
0	155,503	5,320	3.3%
1	4,924	4,233	46.23%

Table 4.2: Contingency table

the two variables, as it allows the examination of conditional probabilities between categorical variables. Table 4.2 presenting the contingency table between `any_refactoring` and `conflict_java`. The descriptive results shows that only about 3% of merges without refactorings result in conflicts, whereas this percentage rises to approximately 46% when refactorings are present.

Afterwards, we developed a logistic regression model considering the presence or absence of conflicts related to the refactoring occurrence on a merge scenario. The equation (1) exhibit how we generate the model.

$$\textit{conflict_java} \sim \textbf{\textit{any_refactoring}}\textit{, family=binomial, data=dataset} \tag{4.4}$$

The results indicate that the `any_refactoring` variable presents an odds ratio of 25.12 (95% CI [23.92, 26.40]), with a pseudo R^2 of 0.08 and a p_value lower than 0.01. This suggests a regression model with reasonable explanatory power and indicates a strong association between the `any_refactoring` variable and the occurrence of merge conflicts. The presence of refactoring is associated with a substantial increase in the odds of merge conflicts and the descriptive results in Table 4.2 also supports this result. While only about 3% of merges without refactorings result in conflicts, this percentage rises to approximately 46% when refactorings are present, which helps explain the high odds ratio observed in the regression model.

Controlling for change size

After observing refactorings relationship with merge conflicts, we analyzed java conflicts with control variables for change size to determine whether the conflict risk is caused by the nature of refactoring which may add or delete lines in different parts of the code and change code structure or simply by the number of lines changed. Table 4.3 presents the

Variable	Group	Q1	Median	Q3
LOC churn	No conflict	21.0	81.0	313.0
LOC churn	Conflict	44.0	159.0	544.0
Files changed	No conflict	3.0	5.0	12.0
Files changed	Conflict	4.0	7.0	18.0

Table 4.3: Distribution of LOC churn and number of files changed by merge conflict occurrence

descriptive statistics of `loc_churn_total` and `files_changed_total` grouped by merge conflict occurrence. The results show that merges leading to conflicts tend to involve substantially larger code churn and a higher number of modified files.

We aim to understand whether commits involving high code churn that do not involve refactorings have the same likelihood of causing merge conflicts as large commits that include refactorings. To investigate this, we constructed a second logistic regression model in which `conflict_java` is the dependent variable, and `loc_churn_total` and `files_changed_total` are included as control variables alongside `any_refactoring`.

Because of high correlation between `loc_churn_total` and `files_changed_total`, we excluded the second variable and performed the logistic regression model. As a result, the variable `loc_churn_total` presented an odds ratio of 1.09 (95% CI [1.04, 1.09]), while `files_changed_total` presents an odds ratio of 1.10 (95% CI [1.07, 1.14]) indicating that increases in these variables are associated with slightly higher odds of merge conflicts. On the other hand, the variable `any_refactoring` presents an odds ratio of 24.13 (95% CI [22.96, 25.35]) indicating that the presence of refactoring is associated with higher odds of merge conflicts.

We also performed a model with `any_refactoring` and `files_changed_total`, which presented similar results to the previous model considering the two independent variables analyzed (odd ratio of 24.37 and 1.20, respectively). By observing the results, while increases in `loc_churn_total` and `files_changed_total` are associated with slightly higher odds of merge conflicts, the presence of refactorings shows a substantially stronger effect. This suggests that refactorings

remain a strong predictor of merge conflicts, even after controlling for the size of code changes.

> *RQ1 - Summary of results:* The presence of refactoring in a merge scenario remains significantly associated with merge conflicts, even after controlling for the size of code changes.

4.3.2 RQ.4 - Does the association between refactoring and merge conflicts vary across merge tools?

Considering the results of RQ.1, refactoring is highly correlated with merge conflicts when controlling for the size of code changes. To assess the association between merge tools and merge conflicts, we compared results across tools. First, we observed the contingency table considering the presence of merge conflicts in merge scenarios for jFSTMerge and IntelliMerge.

Presence of Refactoring	No Conflict	Conflict	Conflict Rate
0	154,883	5,940	3.7%
1	4,924	3,515	38.4%

Table 4.4: Contingency table for jFSTMerge results

Table 4.4 presents the contingency table between `any_refactoring` and `conflict_java` for jFSTMerge, which shows that 3.7% of merges without the presence of refactoring lead to merge conflicts, whereas this percentage rises to approximately 38.4% when refactorings are present. In Table 4.5, we present IntelliMerge results. Tables 4.4 and 4.5 also show higher conflict rates in merge scenarios involving refactorings compared to those without refactorings.

Presence of Refactoring	No Conflict	Conflict	Conflict Rate
0	158,297	2,526	1.5%
1	7,895	1,262	13.8%

Table 4.5: Contingency table for IntelliMerge results

Then, we applied an interaction model to investigate how these three different tools would function in the same merge scenario. As Git, jFSTMerge, and IntelliMerge implement differ-

ent merge strategies, refactoring may impact each tool in a different manner. In other words, given the same merge scenario, different tools may exhibit different sensitivity to refactoring. This allows us to compare how different tools respond to refactoring under identical conditions.

We fitted a logistic regression model including an interaction term between refactoring and tool. Considering the variables controlling for code changes, we observed modest effects. Both `loc_churn_total` and `files_changed_total` presented statistically significant associations with merge conflicts in Java files ($p < 0.001$). However, their corresponding odds ratios (OR ≈ 1.07) indicate that these effects are relatively small in magnitude.

Conversely, the results showed the strongest association is observed for Git (OR=24.10, 95% CI [22,91, 25,29], followed by jFSTMerge (OR=15.5, 95% CI [13.8, 17.5]), and IntelliMerge (OR=9.5, 95% CI [8.3, 10.9]) with a pseudo R^2 of 0.16, and p_values lower than 0.001 for the three variables. It reinforces the highly association of refactoring in merge conflicts with controlled size change.

RQ.2 - Summary of results: The effect of refactoring on merge conflicts differs across tools. While refactoring increases the odds of conflict by 24 times in Git, this effect is reduced to 15.5 times in jFSTMerge and 9.5 times in IntelliMerge, as indicated by the significant interaction term.

4.3.3 RQ.5 - Which refactoring tactic, pure or floss, is more associated with merge conflicts?

The results presented so far reinforce the importance of conducting a detailed study of refactorings in the context of merge conflicts. To address this research question, we analyze merge scenarios, merge commits with or without conflicts, and characterize them based on the refactorings present in their immediate parent commits.

To identify and classify refactorings, we rely on the set of evolutionary commit pairs collected in our previous analysis. These commits represent the changes leading up to merge scenarios and were used as input to the PurityChecker API. The tool classifies commits into

three categories: pure, non-pure, and none, which we interpret as pure refactorings, floss refactorings, and no refactoring detected, respectively.

Out of the 26,562 evolutionary commit pairs involving refactorings, not all met the criteria required to associate an evolutionary change with a refactoring candidate potentially responsible for behavioral modification. We further categorized scenarios as `pure` or `floss` based on floss ratio values of 0 and 1, respectively. Scenarios with floss ratio values between 0 and 1 were excluded, as they represent cases with mixed refactoring tactics. Figure 4.4 present the `floss_ratio` for each a merge scenario.

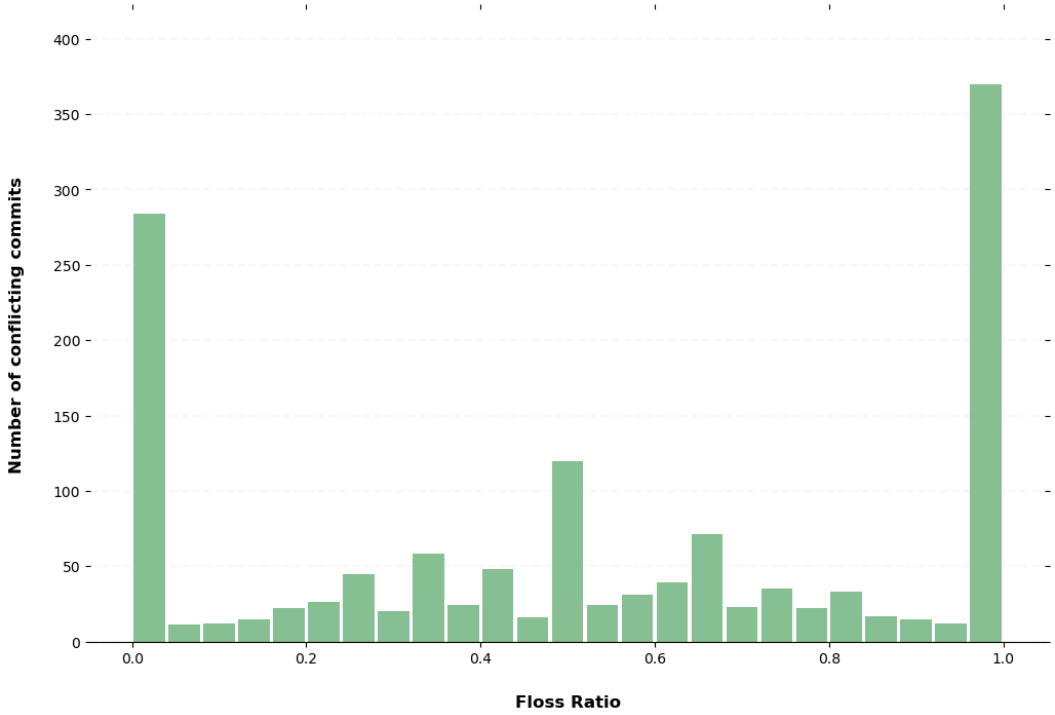


Figure 4.4: Floss Ratio Results

Additionally, the PurityChecker API detects only nine refactoring types, which further reduced the number of classified commits. Considering these constraints, we obtained 1,455 evolutionary commits classified as either `pure` or `floss`. Among them, 549 commits contained exclusively `pure refactorings`, while 906 contained only `floss refactorings`.

We then analyzed a separate dataset of merge scenarios. Each merge scenario was labeled according to the presence and refactoring tactic in its parent commits. We applied a chi-square test to assess differences in conflict occurrence across refactoring groups. We found

that merge scenarios involving `floss refactorings` exhibited a conflict rate of 57%, while those involving `pure refactorings` exhibited a similar rate of 55%. In contrast, merges without refactoring, which are our baseline, showed a substantially lower conflict rate of 9.5%.

The chi-square test indicates a statistically significant difference across groups (Chi^2 = 3531.72, p < 0.001). These results suggest that the presence of refactoring is strongly associated with merge conflicts. However, we observe no meaningful difference between `floss` and `pure` refactorings in terms of conflict occurrence.

Manual Checking of Refactorings' Purity

To validate the outcomes produced by the automated approach using the PurityChecker API, we also carried out a manual inspection of a selected sample comprising 100 pairs of evolutionary commits. This manual analysis was independently performed based on our predefined criteria for purity classification, which were defined in terms of `pure` and `floss refactorings`. Despite the limited sample size, it enabled a detailed examination of the tool's outputs and served as a cross-validation step, providing evidence regarding the tool's reliability. The reviewers agreed with the results of the tool, reinforcing its precision and accuracy previously reported.

In this section, we selected three representative examples of code snippets that present our manual analysis and the results of the tool's classification of refactorings. For each example, we present the code before and after refactoring, the expected type of refactoring, the tool's output, and the manual analysis interpretation. In our first example, two Extract Methods were manually evaluated, as shown in Figure 4.5.

Two methods were extracted from the `find` method: `findByClassName` and `findById`. These two methods were introduced in a new version of the method `find`. In line 5, highlighted in yellow, a condition was added to avoid a `NullPointerException` that had been overlooked in the previous version of the code. The execution of PurityChecker resulted in a non-pure refactoring classification, reporting the description of its execution as: *"Replacements are not justified – non-mapped inner nodes."*. Based on this close inspection of the code snippet containing the Extract Method-related modifications, we observed changes to the internal structure and behavior. We concluded that it was a `floss`

Commit dfa17c4a18cb9585681102770db8b7117eab9531
before the Extract Method **refactoring**

```
1 public static <T extends Descriptor> T find(Collection<? extends T>
2                                             list, String className)
{
3   for (T d : list) {
4       if (d.getClass().getName().equals(className))
5           return d;
6       }
7   for (T d : list) {
8       if (d.getId().equals(className))
9             return d;
10      }
11   return null;
12 }
```

Commit 929adc516289d5396234ecd180be45e8d249c449
after the Extract Method **refactoring**

```
1  public static <T extends Descriptor> T findByClassName(Collection<? extends T>
list,
2                                                          String className)
3  {
4     for (T d : list) {
5         if (d.getClass().getName().equals(className))
6             return d;
7     }
8     return null;
9  }
```

```
1  public static <T extends Descriptor> T findById(Collection<? extends T> list,
2                                                  String id)
3  {
4     for (T d : list) {
5          if (d.getId().equals(id))
6              return d;
7     }
8     return null;
9 }
```

```
1 public static <T extends Descriptor> T find(Collection<? extends T> list,
2                                               String string)
3  {
4      T d = findByClassName(list, string);
5      if (d != null) {
6         return d;
7      }
8      return findById(list, string);
9 }
```

Figure 4.5: Extract Method from an Evolutionary Commit on Catroid project

`refactoring`, as agreed upon by the tool.

Another example, presented in Figure 4.6 shows the before and after versions of a code through a Push Down Method transformation. In this Push Down Method refactoring, the `onCreateOptionsMenu` method was moved to the subclass and modified beyond what is expected to preserve externally observable behavior. Specifically, three statements were removed from the version before the refactoring: the initialization of the armButton and connectButton attributes (lines 5 and 6), and the invocation of the `queryConnectionState()` method (line 7).

The PurityChecker tool classified this refactoring as non-pure, reporting: *"Replacements cannot be justified – with non-mapped leaves or nodes."*. By analyzing the code, we confirmed that it is a `floss refactoring`.

Commit f33eb605efb2d55848d0991f7789003f6a626383
before the Push Down Method **refactoring**

```
1  public abstract class SuperActivity extends Activity {
2  // ...
3     public boolean onCreateOptionsMenu(Menu menu) {
4             getMenuInflater().inflate(R.menu.menu_super_activiy, menu);
5             armButton = menu.findItem(R.id.menu_arm);
6             connectButton = menu.findItem(R.id.menu_connect);
7             drone.MavClient.queryConnectionState();
8             return super.onCreateOptionsMenu(menu);
9     }
10    // ...
11 }
```

Commit 155f524deead57bae4a0996f16c12e283c268a13
after the Push Down Method **refactoring**

```
1  public class NewUIActivity extends SuperActivity {
2  // ...
3       public boolean onCreateOptionsMenu(Menu menu) {
4             getMenuInflater().inflate(R.menu.menu_newui, menu);
5             return super.onCreateOptionsMenu(menu);
6       }
7        // ...
8 }
```

Figure 4.6: Push Down Method from an Evolutionary Commit on DroidPlanner project

Finally, we analyzed an example of a `pure refactoring` involving the Extract

Method refactoring. Figure 4.7 presents modifications to the method body and the creation of a new method. In the version before the Extract Method, the code was implemented in a single method named `logErrorAndSendFailureResponse`. After the refactoring, the developer split this method, and the new method `sendFailureResponse` received the statements previously located in lines 4 and 5. Although the structure of the code changed, its semantics were preserved. Our manual analysis therefore classified this case as a `pure refactoring`.

The PurityChecker result also identified this case as a pure refactoring, returning the message *"All the mappings are matched! - all mapped."*. This indicates that the tool correctly interpreted the changes, leading to the same classification as the manual analysis.

Commit 358e51701e1a58b0c19f63aaf9e62a59165e97ed
before the Extract Method **refactoring**

```
1 private void logErrorAndSendFailureResponse(String errorMessage,
2                                             Message<?> respondTo) {
3       logger.error(errorMessage);
4       Message<?> reply = respondTo.
5                          failureResponse(RequestFailureReason.UNKNOWN);
6       MessagingService.instance().send(reply, respondTo.from());
7 }
```

Commit dfd0aebf73faa8f910e68ff5d50f91fae4f2669e
after the Extract Method **refactoring**

```
1 private void logErrorAndSendFailureResponse(String errorMessage,
2                                             Message<?> respondTo) {
3       logger.error(errorMessage);
4       sendFailureResponse(respondTo);
5 }
6
7 private void sendFailureResponse(Message<?> respondTo) {
8       Message<?> reply = respondTo.
9                          failureResponse(RequestFailureReason.UNKNOWN);
10      MessagingService.instance().send(reply, respondTo.from());
11 }
```

Figure 4.7: Extract Method from an Evolutionary Commit on cassandra project

These three examples illustrated the results of our manual analysis, which were equivalent to the tool's results and may indicate its accuracy.

RQ.3 - Summary of results: Floss refactorings appear slightly more frequently in merge conflicts than pure refactorings, but this difference is not statistically significant.

4.3.4 RQ.6 - How are merge conflicts distributed across refactoring tactics (pure vs. floss), considering the types of refactorings involved?

To answer this question, we collected all occurrences of refactoring types associated with the `pure` and `floss` classifications in our dataset.

Regarding the most representative refactoring types, Split Method, Move and Rename

Method, and Inline Method were predominantly classified as floss refactorings. Conversely, Push Down Method, Move Method, and Pull Up Method were most frequently associated with pure refactorings in the classified evolutionary commits. Table 4.6 presents the complete results of our analysis.

Table 4.6: Pure and Floss Refactorings by Refactoring Type Involved in Conflicts

Refactoring Type	**Pure Refactoring (%)**	**Floss Refactoring (%)**
Extract and Move Method	33.85	66.15
Extract Method	37.00	63.00
Inline Method	27.10	**72.90**
Move and Inline Method	36.10	63.90
Move and Rename Method	19.68	**80.32**
Move Method	**64.07**	35.93
Pull Up Method	**57.86**	42.14
Push Down Method	**94.48**	5.52
Split Method	3.81	**96.19**

Following, we detailed the results presented in the previous Table. Figure 4.8 presents these results. Considering the absolute values by tactic, we observe Move Method, Pull Up Method, and Extract and Move Method as refactorings more applied when performing `pure refactorings`. Regarding `floss refactoring`, Extract Method, Move Method, and Extract and Move Method were more applied. By detailing the results in Table 4.6, we obtained the results in Figure 4.8. Considering the absolute values by tactic, we observe Move Method, Pull Up Method, and Extract and Move Method as refactorings more applied when performing `pure refactorings`. Regarding `floss refactoring`, Extract Method, Move Method, and Extract and Move Method were more applied.

RQ.4 - Summary of results: Floss refactorings are more frequent in merge conflicts than pure refactorings. Among pure refactorings, Move Method is the most frequent type, while Extract Method is the most frequent among floss refactorings.

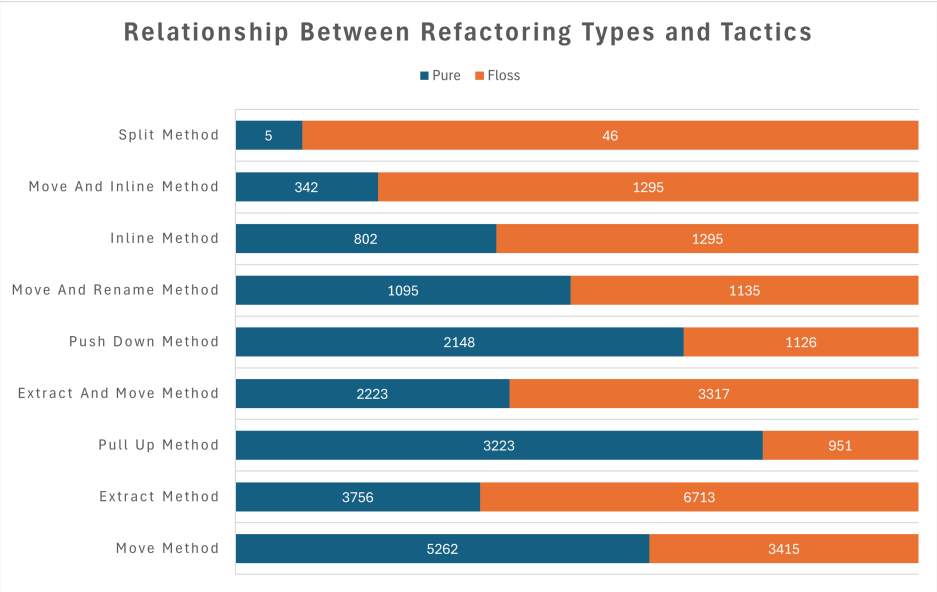


Figure 4.8: Occurrences of Refactoring Types and the Tactic Applied

4.3.5 RQ.7 - Do merge scenarios involving only pure refactorings differ in resolution time from those with floss involvement?

To deepen our investigation, we collected time-to-merge data and applied the Mann-Whitney U test to compare merge scenarios involving `pure` refactorings and those with `floss` involvement. We considered `floss` involvement in a merge scenario according the computed `floss_ratio`, with results illustrated in Figure 4.4. We limited our analysis to Git-based merge times.

Our results revealed no statistically significant difference in merge times between `pure` scenarios and those with `floss` involvement ($U = 1,005,558.5, p_value = 0.22$), with a negligible effect size ($e = 0.021$). The median merge time for scenarios with only `pure refactorings` was 62.56 minutes, while the median for scenarios with `floss` involvement was 63.47 minutes.

Overall, although conflicts involving `pure` refactorings tend to be resolved slightly faster than those involving `floss`, the observed difference is minimal and not practically significant.

RQ.5 - Summary of results: While floss refactorings are more frequently associated with merge conflicts, there is no statistically significant difference between the tactics applied on the time to merge.

4.4 Discussion

In this chapter, we advance prior work by statistically modeling the probability of merge conflicts, estimating the effect of refactorings while controlling for change magnitude variables within merge scenarios. In addition, we decompose this effect by identifying the most common refactoring types involved in conflicts and by analyzing the relationship between refactoring tactics and merge conflicts.

Our results show that refactorings are strongly associated with merge conflicts. Although refactorings aim to improve code quality, they often introduce structural changes that may interact with concurrent modifications. Prior work has primarily investigated refactoring-related conflicts from a detection and resolution perspective. Mahmoudi et al. [50] identify refactorings involved in merge conflicts by analyzing their overlap with conflicting regions, while Dig, Mazoor, and Johnson [33], Ellis, Nadi, and Dig [35], as well as Shen et al. [73], focus on improving merge processes through refactoring-aware techniques. In contrast, our study adopts a statistical modeling perspective, quantifying the association between refactorings and merge conflicts. Rather than focusing on resolving conflicts, we aim to explain their occurrence by estimating the effect of refactorings while controlling for relevant factors.

Another important finding is the consistency of refactoring-related conflicts across Git, jFSTMerge, and IntelliMerge. The strong association observed across tools suggests that refactoring-related conflicts are not tool-specific, but instead reflect a general relationship between refactorings and merge conflicts. This finding has practical implications for improving merge tools, indicating that limitations in handling refactorings may contribute to the persistence of such conflicts.

The refactoring types most frequently involved in conflicts across the three merge tools are not merely local edits. Many of them, such as Move Class, Move Method, Move Attribute, Rename Method, Rename Attribute, and Pull Up Method, tend to propagate changes across multiple references, files, or class hierarchy relationships. This propagation increases

the likelihood of interference with concurrent changes. A plausible explanation is that these refactorings require coordinated updates across different parts of the system, making them particularly prone to merge conflicts.

Structural propagation affects multiple and often non-adjacent parts of the code, including method calls, attributes, imports, and hierarchical relationships. Each merge tool handles this propagation differently. Unstructured merge tools, such as Git, operate at the textual level and do not recognize refactorings, making them unable to properly handle such changes. Semistructured approaches, such as jFSTMerge and IntelliMerge, operate at the syntactic level, improving the handling of local conflicts but still lacking support for tracking distributed changes across the system.

Existing merge approaches, including unstructured, semistructured, and structured techniques, primarily focus on resolving syntactic or localized structural conflicts. However, they still struggle to handle the propagation of changes introduced by refactorings across multiple program elements. Refactorings may span multiple commits and affect different parts of the codebase, whereas merge tools typically operate locally and do not model refactorings as distributed transformations over time.

This limitation becomes particularly relevant in light of recent work that represents refactorings as interconnected sequences of changes, suggesting that their effects extend beyond the scope of a single merge scenario [18]. As a result, handling structural propagation remains an open challenge for current merge tools. In this context, merge approaches can be classified into three levels of code change comprehension: (i) textual, which operates on lines of code; (ii) locally structured, which leverages syntactic representations such as ASTs or graphs; and (iii) globally structured, which would require tracking semantically related changes across the system and remains largely unexplored.

According to previous studies on refactoring tactics, certain types of refactorings are more common in pure refactoring, particularly those with a higher impact on system structure and modularity [56]. These tend to correspond to medium and high-level refactorings. Additionally, as projects evolve, developers are more likely to perform structural and intensive refactorings separately from feature development, increasing the prevalence of pure refactoring [60].

In contrast, floss refactoring typically involves low to medium-level refactorings and re-

flects a different execution rhythm, characterized by frequent and incremental code changes. Previous studies report floss refactoring as more frequent and less time-intensive, as it is often interleaved with other code modifications [4], [60]. As a result, refactorings such as Rename Method, Split Method, and Extract/Inline Method are more commonly observed in this context [60].

Our results are consistent with these findings. Refactorings such as Move Method, Pull Up Method, and Push Down Method align with patterns observed in pure refactoring. For instance, Pull Up Method involves relocating behavior across the class hierarchy, which typically requires prior dependency analysis and affects multiple parts of the system. Consequently, these refactorings tend to be associated with more complex merge scenarios.

Conversely, refactorings such as Split Method, Extract Method, and Inline Method are more frequently observed in conflicts involving floss refactoring. These operations are generally more localized and incremental, which may explain their association with smaller and more frequent changes. Our classification results are consistent with this distinction between refactoring tactics.

Although the study by Noei et al. [60] provides a comprehensive analysis of refactoring rhythms and tactics, it does not estimate the time required to perform pure versus floss refactoring. Instead, it focuses on frequency and temporal distribution. Our study complements these findings by providing evidence on merge resolution time, showing that conflicts involving pure refactorings tend to take longer to resolve than those involving floss refactorings.

Overall, conflicts involving pure refactorings tend to be resolved slightly faster than those involving floss refactorings. While our study does not explicitly classify refactorings according to their scope or abstraction level, the observed differences in resolution time may suggest that floss refactorings involve additional integration effort. However, this difference is not statistically significant, indicating that the refactoring tactic has limited impact on merge resolution time.

To further investigate this result, we conducted a qualitative analysis of Pull Request discussions associated with the same merge scenarios classified as pure or floss by PurityChecker. Using LLM-assisted analysis of PR comments, review discussions, and merge-related messages, we identified explicit delay signals associated with both refactoring tactics.

For pure refactorings, the identified evidence was primarily related to merge conflicts,

branch synchronization, CI/test failures, and coordination required for code integration. In contrast, floss refactorings were associated with a broader range of delay signals, including Pull Request reopening, rework, additional review cycles, and increased coordination between developers and reviewers.

These findings suggest that, despite the similar merge times observed across both refactoring tactics, the underlying sources of integration effort may differ. In particular, floss refactorings appear to involve a broader combination of technical and social coordination challenges, whereas pure refactorings are predominantly associated with technical integration issues.

Although the differences in merge time were not statistically significant, the qualitative evidence suggests that different refactoring tactics may impose distinct integration load on development teams. While pure refactorings appear to require technical integration effort, floss refactorings may demand additional coordination and review activities before integration can be completed.

4.5 Threats to Validity

In this section, we present possible threats to the validity of our study, categorized into construct, internal, and external validity, and address factors that may affect our findings and how we mitigate them.

4.5.1 Construct Validity

A potential source of bias lies in the classification of refactorings into pure and floss categories using the PurityChecker API, which could affect the observed differences in conflict occurrence and time-to-merge between these tactics. To mitigate this, we employed triangulation by combining automated and manual approaches. The automated analysis enabled large-scale identification of refactorings, while manual inspection of a sample was used to validate the results. The outcomes of the manual analysis were consistent with those produced by the API and aligned with its reported precision and recall [61], strengthening the reliability of the classification.

The PurityChecker API detects only nine types of refactorings, which represents another

limitation of our study. Although these types correspond to some of the most commonly applied refactorings, this limitation cannot be disregarded. This potential underestimation may have attenuated the observed differences between pure and floss refactorings.

4.5.2 Internal Validity

Our results concerning refactoring classification rely on Refactoring Miner 2.0 [79] and 3.0 [6]. In case Refactoring Miner provide incorrect information, this might be reflected in time-to-merge results. However, the tool present high precision and recall in detecting refactorings and was broadly applied in researches, which makes it more reliable.

In addition, this study included the results of the PurityChecker tool [61] in the analysis of pure and floss refactorings involved in merge conflicts. To mitigate the risk of bias introduced by the automated classification, we performed a manual cross-validation on a sample of the data. The results were consistent with the automated classification, suggesting that the potential impact of this bias may be limited.

4.5.3 External Validity

The use of the PurityChecker API to classify refactorings into pure and floss categories introduces threats to external validity. This classification relies on heuristics and assumptions that may not generalize to projects with different development practices or commit structures, and may not accurately capture how refactorings are performed in other contexts. Furthermore, the tool has been primarily evaluated on open-source Java projects, which may limit its applicability to other programming languages or industrial environments. As a result, our findings may not generalize to other languages or development ecosystems.

Another threat concerns the merge tools used in this study. Since Git, jFSTMerge, and IntelliMerge differ in how they detect and handle conflicts, inaccuracies or limitations in these tools may affect the observed results. Consequently, the reported associations may vary depending on the characteristics and precision of the merge tool applied. However, the same general relationship between refactorings and merge conflicts was consistently observed across all three tools. This consistency suggests that our findings are not specific to a single merge tool and may generalize to other tools that adopt similar merging strategies.

Additionally, although we include control variables related to change magnitude (`loc_churn_total` and `files_changed_total`), other unobserved factors (e.g., developer experience or project-specific practices) may be associated with merge conflicts. These factors may affect the generalizability of our findings to different development settings.

4.6 Concluding Remarks

In this chapter, we examine merge conflicts and their relationship with refactorings. Our results show that refactorings are strongly associated with merge conflicts, even after controlling for change magnitude. Furthermore, this association is observed consistently across the three merge tools analyzed in this study.

Additionally, we find that floss refactorings are more frequently present in conflicting scenarios than pure refactorings. Although conflicts involving pure refactorings tend to require more time to be merged, the observed differences present a low practical effect.

Since refactorings are commonly applied in software projects, our findings underscore the need to improve merge processes and tools to better accommodate refactoring-related changes without compromising project quality or delivery timelines.

Chapter 5

Related Work

In recent years, there has been growing interest in studying merge conflicts. Various studies presented different merge tools with new techniques, some of which included semantic analysis and refactoring-aware behavior. Despite the number of merge tools, the results of the studies overlook several interesting aspects, such as the actual impact of refactoring on merge conflicts. This chapter provides an overview of the existing literature that explores factors related to merge conflicts, a comparison of merge tools, and the connection between merge conflicts and refactoring. This review aims to shed light on the interplay between these two crucial concepts and highlight the gaps in current knowledge, some of which we analyzed, thereby contributing to the field.

5.1 Factors and Indicators Related to Merge Conflicts

Related research on factors influencing merge conflicts analyzed the challenges of resolving conflicts [82] and explored different aspects and programming languages [31]. An interesting finding lies on the identification of a dependency between conflicting and non-conflicting codes, addressing as a research agenda a possible contribution of refactorings to more complex conflicts [82]. A complementary study [31] focused on factors influencing merge conflicts in Ruby and Python MVC projects, examining aspects such as modularity, size, and timing of developer contributions, but did not address the specific variables considered in our study. Our research addresses these gaps by analyzing variables not previously explored, offering a complementary perspective.

According to [74], changes in the merge scenario can influence merge conflicts, with method signatures and bodies being more commonly related to conflicts and refactoring operations. Our study investigated the conflicting scenarios, while observing refactoring operations. Additionally, we identified and presented variables directly related to merge conflicts, highlighting the most relevant ones in this context.

Time to merge was analyzed as a dependent variable to understand factors contributing to longer conflict resolution duration [82]. Our research presents new findings, focusing on collecting and analyzing time to merge in scenarios with and without refactorings and presenting refactoring as a predictive factor for merge conflicts.

Our findings also complement those reported in [62] on the influence of refactorings on merge effort, defined as the difference in the number of actions performed across the parent and base branches. That study highlighted how refactorings increase merge effort, often leading to additional lines of conflicting code. Our research contributes to this understanding by demonstrating that the time to merge is significantly higher when refactorings are present in evolutionary commits, which encompass a broader set of commits compared to the more narrowly focused effort study [62].

The study in [59] examined the barriers developers face when dealing with merge conflicts and concluded that the understandability of conflicting codes is the most important factor in resolving them. Considering these results, our research may offer insights into the conflicts involving refactorings, potentially contributing to resolving this issue.

5.2 Comparison and Evaluation of Merge Tools

IntelliMerge [73] was developed to resolve conflicts related to refactoring by explicitly detecting and resolving them through its algorithm. The tool was analyzed across 1,070 merge cases from 10 well-known open-source Java projects, finding that IntelliMerge reduced conflicts by 50% compared to Git and jFSTMerge tools. Our study expanded on this work by comparing IntelliMerge with Git and jFSTMerge, using a larger sample and focusing on predefined variables to assess these tools through logistic regression models.

RefMerge was developed as an evolution of the MolhadoRef merge tool, addressing its limitations by implementing algorithmic improvements and utilizing RefactoringMiner for

refactoring detection [35]. A study comparing Git, IntelliMerge, and RefMerge found that RefMerge reduced conflicts in 25% of merge scenarios, while IntelliMerge reduced conflicts by 24%. However, IntelliMerge struggled with the number of refactoring operations in a merge scenario, and RefMerge introduced conflicts when handling move-related refactorings. Our study focuses on comparing Git, jFSTMerge, and IntelliMerge to analyze how code changes impacts merges rather than evaluating the precision of these tools.

Other studies compared tools that apply different merge approaches. Consider two approaches, for example, structured and semistructured merge [26] and unstructured and semistructured merge [8], [24], [73]. One study compared the unstructured, semistructured, and structured merge approaches to find the advantages of structured merge and discover what combination of these approaches in a tool was more suitable for the merge conflict problem [71]. These studies also differ from ours because they analyze false positive and false negative conflicts, looking for correctness in the analyzed tools. Our study compared results from three tools looking for patterns that can lead to conflicting prediction factors.

In an attempt to prevent developers from managing spurious conflicts, researchers developed semistructured tools or improved existing ones. Recent research proposed SESAME, a semistructured merge tool that leverages language-specific syntactic separators to improve accuracy on merging by inferring structure without parsing [25]. Despite the reduction of merging conflicting files and regions, this new tool's results still compromise the intended program behavior, failing to detect some merge conflict.

A recent related study made significant contributions to the improvement of merge tools by analyzing merge conflicts through the definition of a clean merge (or clean integration) as one that preserves the intended program behavior [70]. The results offered a new perspective on how merge conflicts should be assessed, emphasizing the importance of distinguishing correct merges from incorrect ones, even when no textual conflict is reported. Additionally, the study investigated how merge tools behave in the presence of refactorings, revealing that tools like IntelliMerge that are designed to handle such cases, did not perform as well as previously claimed. In a broader and more representative dataset, IntelliMerge was found to be rarely applicable and frequently produced incorrect merges.

The study also analyzed which merge tools save developers the most time, proposing a cost model that considers the number of correct, incorrect, and unhandled merges [70].

The best tools are those that reduce incorrect merges (which have the highest cost) and minimize unhandled ones. In contrast, our study differs by focusing on how refactorings specifically impact the merge process. Instead of evaluating merge tools directly, we analyze the difference in merge time between conflicts that involve refactorings and those that do not. Although IntelliMerge underperformed in large-scale evaluations, we chose to include it in our analysis due to its explicit design as a refactoring-aware tool. Its structure allowed us to explore how merge tools interpret and react to refactorings, offering valuable insights into the characteristics and challenges of merge scenarios involving such changes.

While the study [70] offers a comprehensive evaluation of merge tools using a rigorous methodology, it does not directly analyze the role of refactorings in merge scenarios. Refactorings are acknowledged as a potential source of conflicts and incorrect merges, and tools such as IntelliMerge are included in the evaluation. However, the study does not investigate the types, frequency, or specific impact of refactorings on merge outcomes. In contrast, our research focuses on understanding the relationship between refactorings and merge conflicts, complementing the related work by providing a more detailed view of how structural changes to the code contribute to merge complexity.

5.3 Impact of Refactorings on Merge Conflicts

Regarding software merging and refactoring, we found some studies that relate these concepts [32], [73], [50]. MolhadoRef [32] is a tool proposed considering object-oriented refactorings since their global changes result in merge conflicts. MolhadoRef is a refactoring-aware SCM system that works by recording change actions (refactorings and edits) required to create one version and replaying them while merging versions to reduce merge conflicts. This strategy was proposed to increase software development efficiency and decrease the time and effort needed for software maintenance. However, the tool does not handle all refactorings, such as the Extract Method.

To address the presented flaw in the MolhadoRef tool, IntelliMerge [73] is a merging tool that supports refactorings of the Extract Method type. Conflicts relating to refactoring are explicitly detected and resolved by the algorithm. In this study, we evaluated IntelliMerge, yielding the following outcomes: IntelliMerge demonstrates a comparable performance to

the analyzed tools concerning conflict resolution. However, despite being a refactoring-aware tool, it encountered challenges in managing various conflict-related refactorings and exhibited a similar proportion of floss refactoring compared to the analyzed tools.

An experimental investigation examined 3,000 repositories [50] to explore the context previously suggested [32]. The association between refactorings and merge conflicts, which indicates that refactorings are involved in at least 22% of merge conflicts, stands out as one of the most significant findings. In contrast, our study focuses on examining refactorings in merges by applying three tools. We conducted a comprehensive evaluation that encompassed both manual and automatic assessments. Our results pointed out that Git presented almost 43% of merge conflicts involving refactorings.

Considering the results presented in the studies previously mentioned [50], [73], RefMerge was developed as an evolution of MolhadoRef trying to improve the weaknesses of this tool [35]. RefMerge applied changes in its algorithms and used the RefactoringMiner tool as a refactoring detector to obtain better results. A study comparing Git, IntelliMerge, and RefMerge showed that these tools have strengths and weaknesses. Conversely, our studies compare three tools, including Git and IntelliMerge, aimed at comprehending the influence of refactorings in merges, not intending to analyze the precision of the tools.

Chapter 6

Conclusions

In this study, we investigate factors related to merge conflicts in collaborative development environments. By examining the evolutionary contributions of developers across a large number of projects, we aim to understand how these dynamics are associated with the occurrence of merge conflicts. Our findings provide insights into how development practices relate to merge conflicts and may contribute to improving collaboration and efficiency in software development processes.

Our research employed two main approaches. In the first approach, we analyzed 507,411 merge scenarios, including 28,645 conflicting scenarios, of which 20,266 were unrelated to Java files. These analyses, conducted using three merge tools, focused on conflict detection and considered five variables associated with merge conflicts. We evaluated the behavior of Git, jFSTMerge, and IntelliMerge in handling merge scenarios. Our results indicate that refactorings in parent commits are strongly associated with merge conflicts across all three tools. We also identified the refactoring types most frequently involved in merge conflicts, including Rename Parameter, Rename Method, Rename Variable, Move Class, Extract Method, Rename Attribute, Move Attribute, Pull Up Method, and Extract Variable.

We also analyzed the time required to merge conflicting scenarios based on Git history. The results show that scenarios involving refactorings in parent commits present longer merge times on average than those without refactorings.

Since refactorings emerged as a variable consistently associated with merge conflicts across the analyzed merge tools, we statistically modeled the relationship between refactorings and merge conflicts while controlling for variables related to change magnitude. Our re-

sults show that refactorings remain strongly associated with merge conflict occurrence even after accounting for these factors. Additionally, we classified refactorings as pure or floss based on an analysis of 26,562 pairs of evolutionary commits. This classification enabled us to further examine how different refactoring tactics are associated with merge conflicts.

Our results show that floss refactorings are more frequently involved in merge conflicts than pure refactorings. In our dataset, floss refactorings account for 65% of conflicting scenarios, while pure refactorings account for 45%. Among floss refactorings, Extract Method is the most frequent type, whereas Move Method is the most frequent among pure refactorings. When comparing how the same refactoring types are distributed across tactics, we observe that Split Method occurs more frequently as floss refactoring, while Push Down Method is more commonly observed as pure refactoring.

Regarding merge time, conflicts involving floss refactorings present slightly longer resolution times on average than those involving pure refactorings. However, this difference is not statistically significant.

Overall, these findings highlight that merge activities are closely related to refactoring practices. Given that refactorings are commonly applied in software projects, our results underscore the importance of improving merge processes and tools to better support refactoring-related changes.

Considering our findings, the contributions outlined for developers, tool builders, and researchers emphasize strategies for managing merge conflicts in collaborative software development:

For Developers: Refactorings are strongly associated with merge conflicts, and developers should plan and coordinate them carefully to mitigate risks. Although beneficial for long-term code quality, refactorings may introduce hidden challenges for code integration. In this context, floss refactorings are associated with slightly longer merge times and tend to appear more frequently in conflict-prone histories, potentially increasing the likelihood of conflicts through multiple smaller changes across files. Effective conflict management requires awareness of refactorings and their trade-offs in collaborative development.

For Tool Builders: Our findings highlight the importance of incorporating refactoring awareness into merge tools to improve conflict detection and resolution. Given the strong association between refactorings and merge conflicts, tools can leverage this information

to alert developers about increased conflict risk and encourage coordination. Incorporating refactoring-aware heuristics may enhance both detection and resolution, particularly for textual merge tools. Additionally, as refactorings may influence time-to-merge, tools could adapt their strategies based on the presence and tactic of refactoring. Overall, enabling merge tools to detect refactorings in parent commits and adjust their behavior accordingly can improve conflict management in collaborative development environments.

For Researchers: Our findings open opportunities for further research on how different refactoring patterns are associated with merge conflicts. Future work may explore how refactoring characteristics interact with collaborative factors, such as the number of developers involved, to influence conflict occurrence. Additionally, distinguishing between pure and floss refactorings enables more nuanced analyses of collaborative development, supporting improved conflict prediction models, developer-centered studies, and the creation of benchmark datasets for reproducibility. Our results also motivate the development of refactoring-aware merging algorithms that better propagate changes across branches and handle refactoring-related conflicts. Finally, refactorings should be treated as a multifaceted factor in merge research, considering their types, tactics, and context of occurrence.

6.1 Future Work

Our research sets the stage for future investigations. For future work, we suggest a study on the influence of well-executed refactorings on the severity of merge conflicts, exploring their complexity. Another aspect that deserves investigation is to explore how developers' approaches to refactoring, such as the order of the refactorings, affect the occurrence of merge conflicts.

Based on our observations during this research, we suggest the development of a novel merge tool that could significantly ease the burden on developers when handling refactorings in merge conflicts. This tool, inspired by the limitations of IntelliMerge and the literature on semantic merge tools, could recognize refactorings, suggest the intentions behind the refactoring operations, and send an alert with this information. Future research could also build on our results by incorporating the nine refactorings into developing merge tools.

Devising a git flow that brings improvements to the process regarding refactoring imple-

mentation, representing promising future work. Code integration practices, which consider when and how to include refactorings, discuss floss refactoring practices to separate the tidy operations from refactoring operations in different integration steps.

Building on our findings that refactoring operations are involved in a significant portion of merge conflicts, we also suggest to explore Pull Requests (PRs) within this context. A potential line of investigation is whether PRs containing refactorings operations have a significantly higher likelihood of triggering merge conflicts compared to those focused on new functionalities or bug fixes.

Investigating the impact of different refactoring strategies across a broader range of programming languages and development contexts would help assess whether our conclusions hold beyond Java-based projects. However, it is equally important to consider the influence of different team sizes, collaboration styles, or development methodologies on the occurrence and resolution of merge conflicts in relation to refactorings. Further studies in these areas could provide valuable insights. Another promising direction would be to refine merge tools to better handle the specific challenges presented by floss refactorings, potentially reducing the time required for conflict resolution. Finally, examining how the combination of pure and floss refactorings affects the long-term software quality, including code smells and maintainability, would provide a more comprehensive understanding of their trade-offs.

6.1.1 AI Application in Repositories

Using June 29, 2021, the launch date of the GitHub Copilot Preview [84], [85], as a temporal milestone for the availability of AI-assisted programming tools , the repositories applied in this research were classified according to the temporal distribution of their merges. Of the 499 repositories presenting three-way merges, 380 (76.2%) showed only merges prior to the benchmark, 118 (23.6%) confirmed merges both before and after the benchmark, and only 1 repository (0.2%) showed only merges after the considered data. These results indicate that most of the developed projects were predominantly developed in a period prior to the widespread availability of AI-based programming assistance tools. Although the repositories classified as mixed may have been exposed to such tools during part of their evolution, the temporal data alone do not allow us to conclude that there was interference in the use of AI in code production. These pieces of evidence suggest that the dataset was predominantly

created in a context when AI-assisted development was not common. Therefore, the risk of significant bias introduced by code generated using AI is considered limited in our study.

This observation opens opportunities for future work investigating merge and refactoring activities in repositories with different levels of code generated by AI. In particular, future studies could compare repositories predominantly developed by humans, repositories containing both human and AI-assisted contributions, and repositories largely generated by AI-based development tools or autonomous programming agents. Such analyses may help identify differences in development practices, merge characteristics, and refactoring patterns associated with different degrees of AI participation.

Bibliography

[1] Chaima Abid, Vahid Alizadeh, Marouane Kessentini, Thiago do Nascimento Ferreira, and Danny Dig. 30 years of software refactoring research: A systematic literature review. *arXiv preprint arXiv:2007.02194*, 2020.

[2] Paola Accioly, Paulo Borba, and Guilherme Cavalcanti. Understanding semi-structured merge conflict characteristics in open-source java projects. *Empirical Software Engineering*, 23(4):2051–2085, 2018.

[3] Paola Accioly, Paulo Borba, Léuson Silva, and Guilherme Cavalcanti. Analyzing conflict predictors in open-source java projects. In *Proceedings of the 15th International Conference on Mining Software Repositories*, pages 576–586, 2018.

[4] Mansi Agnihotri and Anuradha Chug. Understanding refactoring tactics and their effect on software quality. In *2022 12th International Conference on Cloud Computing, Data Science & Engineering (Confluence)*, pages 41–46. IEEE, 2022.

[5] 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.

[6] Pouria Alikhanifard and Nikolaos Tsantalis. A novel refactoring and semantic aware abstract syntax tree differencing tool and a benchmark for evaluating the accuracy of diff tools. *ACM Transactions on Software Engineering and Methodology*, 34(2), January 2025.

[7] Sven Apel, Olaf Leßenich, and Christian Lengauer. Structured merge with auto-tuning:

balancing precision and performance. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 120–129, 2012.

[8] 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.

[9] Gabriele Bavota, Bernardino De Carluccio, Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, and Orazio Strollo. When does a refactoring induce bugs? an empirical study. In *2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation*, pages 104–113. IEEE, 2012.

[10] Gabriele Bavota, Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, and Fabio Palomba. An experimental investigation on the innate relationship between quality and refactoring. *Journal of Systems and Software*, 107:1–14, 2015.

[11] Kent Beck, Mike Beedle, Arie Van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, et al. Manifesto for agile software development. 2001.

[12] Kent Beck and Martin Fowler. *Planning extreme programming*. Addison-Wesley Professional, 2001.

[13] Valdis Berzins. Software merge: Models and methods for combining changes to programs. In *ESEC'91: 3rd European Software Engineering Conference, ESEC'91 Milan, Italy, October 21–24, 1991 Proceedings 3*, pages 229–250. Springer, 1991.

[14] Christian Bird and Thomas Zimmermann. Assessing the value of branches with what-if analysis. In *Proceedings of the 20th International Symposium on Foundations of Software Engineering*. ACM, 2012.

[15] Caius Brindescu. How do developers resolve merge conflicts? an investigation into the processes, tools, and improvements. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 952–955, 2018.

[16] Caius Brindescu, Iftekhar Ahmed, Carlos Jensen, and Anita Sarma. An empirical investigation into merge conflicts and their effect on software quality. *Empirical Software Engineering*, 25:562–590, 2020.

[17] Caius Brindescu, Iftekhar Ahmed, Rafael Leano, and Anita Sarma. Planning for untangling: Predicting the difficulty of merge conflicts. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pages 801–811. IEEE, 2020.

[18] Aline Brito, Andre Hora, and Marco Tulio Valente. Refactoring graphs: Assessing refactoring over time. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 367–377. IEEE, 2020.

[19] Yuriy Brun, Reid Holmes, Michael D Ernst, and David Notkin. Proactive detection of collaboration conflicts. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 168–178, 2011.

[20] Yuriy Brun, Reid Holmes, Michael D Ernst, and David Notkin. Early detection of collaboration conflicts and risks. *IEEE Transactions on Software Engineering*, 39(10):1358–1375, 2013.

[21] Jim Buffenbarger. Syntactic software merging. In *Software Configuration Management*, pages 153–172. Springer, 1993.

[22] Carlos Carbonera, Kleinner Farias, CéSar Graeff, and Robson Silva. Software merge: A two-decade systematic mapping study. In *Proceedings of the XXXVII Brazilian Symposium on Software Engineering*, pages 99–108, 2023.

[23] Guilherme Cavalcanti, Paola Accioly, and Paulo Borba. Assessing semistructured merge in version control systems: A replicated experiment. In *2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–10. IEEE, 2015.

[24] Guilherme Cavalcanti, Paulo Borba, and Paola Accioly. Evaluating and improving semistructured merge. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):1–27, 2017.

[25] Guilherme Cavalcanti, Paulo Borba, Leonardo dos Anjos, and Jonatas Clementino. Semistructured merge with language-specific syntactic separators. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, pages 1032–1043, 2024.

[26] Guilherme Cavalcanti, Paulo Borba, Georg Seibt, and Sven Apel. The impact of structure on software merging: Semistructured versus structured merge. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1002–1013, 2019.

[27] Diego Cedrim, Alessandro Garcia, Melina Mongiovi, Rohit Gheyi, Leonardo Sousa, Rafael De Mello, Baldoino Fonseca, Márcio Ribeiro, and Alexander Chávez. Understanding the impact of refactoring on smells: A longitudinal study of 23 software projects. In *Proceedings of the 2017 11th Joint Meeting on foundations of Software Engineering*, pages 465–475, 2017.

[28] Jonatas Clementino, Paulo Borba, and Guilherme Cavalcanti. Textual merge based on language-specific syntactic separators. In *Anais do XXXV Simposio Brasileiro de Engenharia de Software*, Porto Alegre, RS, Brasil, 2021. SBC.

[29] Léuson Da Silva, Paulo Borba, Toni Maciel, Wardah Mahmood, Thorsten Berger, João Moisakis, Aldiberg Gomes, and Vinícius Leite. Detecting semantic conflicts with unit tests. *Journal of Systems and Software*, 214:112070, 2024.

[30] Massimiliano Di Penta, Gabriele Bavota, and Fiorella Zampetti. On the relationship between refactoring actions and bugs: a differentiated replication. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 556–567, 2020.

[31] Klissiomara Dias, Paulo Borba, and Marcos Barreto. Understanding predictive factors for merge conflicts. *Information and Software Technology*, 121:106256, 2020.

[32] Danny Dig, Kashif Manzoor, Ralph Johnson, and Tien N Nguyen. Refactoring-aware configuration management for object-oriented programs. In *29th International Conference on Software Engineering (ICSE'07)*, pages 427–436. IEEE, 2007.

[33] Danny Dig, Kashif Manzoor, Ralph E Johnson, and Tien N Nguyen. Effective software merging in the presence of object-oriented refactorings. *IEEE Transactions on Software Engineering*, 34(3):321–335, 2008.

[34] JoÃ£o Pedro Henrique Santos Duarte. The impact of language independence on structured merge accuracy and efficiency. DissertaÃ§Ã£o (mestrado em ciÃªncia da computaÃ§Ã£o), Universidade Federal de Pernambuco, Recife, PE, Brasil, 2025.

[35] Max Ellis, Sarah Nadi, and Danny Dig. Operation-based refactoring-aware merging: An empirical evaluation. *IEEE Transactions on Software Engineering*, 49(4):2698–2721, 2022.

[36] Andy P Field, Jeremy Miles, and Zoë Field. Discovering statistics using r/andy field, jeremy miles, zoë field., 2012.

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

[38] Catherine O Fritz, Peter E Morris, and Jennifer J Richler. Effect size estimates: current use, calculations, and interpretation. *Journal of experimental psychology: General*, 141(1):2, 2012.

[39] 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.

[40] Yaroslav Golubev, Zarina Kurbatova, Eman Abdullah AlOmar, Timofey Bryksin, and Mohamed Wiem Mkaouer. One thousand and one stories: a large-scale survey of software refactoring. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1303–1313, 2021.

[41] Luciana De Q. L. Gomes, Melina Mongiovi, Sabrina Souto, and Everton L. G. Alves. The hidden challenges of merging: A tool-based exploration. In *2025 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 587–597, 2025.

[42] James Ivers, Robert L Nord, Ipek Ozkaya, Chris Seifried, Christopher S Timperley, and Marouane Kessentini. Industry experiences with large-scale refactoring. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1544–1554, 2022.

[43] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M German, and Daniela Damian. The promises and perils of mining github. In *Proceedings of the 11th working conference on mining software repositories*, pages 92–101, 2014.

[44] Bakhtiar Khan Kasi and Anita Sarma. Cassandra: Proactive conflict minimization through optimized task scheduling. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 732–741. IEEE, 2013.

[45] Miryung Kim, Dongxiang Cai, and Sunghun Kim. An empirical investigation into the role of api-level refactorings during software evolution. In *Proceedings of the 33rd international conference on software engineering*, pages 151–160, 2011.

[46] Miryung Kim, Thomas Zimmermann, and Nachiappan Nagappan. A field study of refactoring challenges and benefits. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, pages 1–11, 2012.

[47] Olaf Leßenich, Janet Siegmund, Sven Apel, Christian Kästner, and Claus Hunsen. Indicators for merge conflicts in the wild: survey and empirical study. *Automated Software Engineering*, 25:279–313, 2018.

[48] Hui Liu, Yuan Gao, and Zhendong Niu. An initial study on refactoring tactics. In *2012 IEEE 36th Annual Computer Software and Applications Conference*, pages 213–218. IEEE, 2012.

[49] Wardah Mahmood, Moses Chagama, Thorsten Berger, and Regina Hebig. Causes of merge conflicts: A case study of elasticsearch. In *Proceedings of the 14th International Working Conference on Variability Modelling of Software-Intensive Systems*, pages 1–9, 2020.

[50] 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.

[51] Shane McKee, Nicholas Nelson, Anita Sarma, and Danny Dig. Software practitioner perspectives on merge conflicts and resolutions. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 467–478. IEEE, 2017.

[52] Tom Mens. A state-of-the-art survey on software merging. *IEEE transactions on software engineering*, 28(5):449–462, 2002.

[53] Tom Mens and Tom Tourwé. A survey of software refactoring. *IEEE Transactions on software engineering*, 30(2):126–139, 2004.

[54] Jaziel S Moreira, Everton LG Alves, and Wilkerson L Andrade. An exploratory study on extract method floss-refactoring. In *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, pages 1532–1539, 2020.

[55] Emerson Murphy-Hill and Andrew P Black. Refactoring tools: Fitness for purpose. *IEEE software*, 25(5):38–44, 2008.

[56] Emerson Murphy-Hill, Chris Parnin, and Andrew P Black. How we refactor, and how we know it. *IEEE Transactions on Software Engineering*, 38(1):5–18, 2011.

[57] Meiyappan Nagappan, Thomas Zimmermann, and Christian Bird. Diversity in software engineering research. In *Proceedings of the 2013 9th joint meeting on foundations of software engineering*, pages 466–476, 2013.

[58] Nachiappan Nagappan and Thomas Ball. Use of relative code churn measures to predict system defect density. In *Proceedings of the 27th international conference on Software engineering*, pages 284–292, 2005.

[59] Nicholas Nelson, Caius Brindescu, Shane McKee, Anita Sarma, and Danny Dig. The life-cycle of merge conflicts: processes, barriers, and strategies. *Empirical Software Engineering*, 24(5):2863–2906, 2019.

[60] Shayan Noei, Heng Li, Stefanos Georgiou, and Ying Zou. An empirical study of refactoring rhythms and tactics in the software development process. *IEEE Transactions on Software Engineering*, 49(12):5103–5119, 2023.

[61] Pedram Nouri. Puritychecker: A tool for detecting purity of method-level refactoring operations. Master's thesis, Concordia University, 2023.

[62] André Oliveira, Vânia Neves, Alexandre Plastino, Ana Carla Bibiano, Alessandro Garcia, and Leonardo Murta. Do code refactorings influence the merge effort? In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 134–146. IEEE, 2023.

[63] Daniel Oliveira, Wesley K. G. AssunÃ§Ã£o, Alessandro Garcia, Ana Carla Bibiano, MÃ¡rcio Ribeiro, Rohit Gheyi, and Baldoino Fonseca. The untold story of code refactoring customizations in practice. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 108–120, 2023.

[64] William F Opdyke. *Refactoring object-oriented frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[65] Moein Owhadi-Kareshk, Sarah Nadi, and Julia Rubin. Predicting merge conflicts in collaborative software development. In *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–11. IEEE, 2019.

[66] Matheus Paixão, Anderson Uchôa, Ana Carla Bibiano, Daniel Oliveira, Alessandro Garcia, Jens Krinke, and Emilio Arvonio. Behind the intents: An in-depth empirical study on software refactoring in modern code review. In *Proceedings of the 17th International Conference on Mining Software Repositories*, pages 125–136, 2020.

[67] Dewayne E Perry, Harvey P Siy, and Lawrence G Votta. Parallel changes in large-scale software development: an observational case study. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 10(3):308–337, 2001.

[68] Shaun Phillips, Jonathan Sillito, and Rob Walker. Branching and merging: an investigation into current version control practices. In *Proceedings of the 4th international*

workshop on cooperative and human aspects of software engineering, pages 9–15, 2011.

[69] Anita Sarma, David F Redmiles, and Andre Van Der Hoek. Palantir: Early detection of development conflicts arising from parallel code changes. *IEEE Transactions on Software Engineering*, 38(4):889–908, 2011.

[70] Benedikt Schesch, Ryan Featherman, Kenneth J Yang, Ben Roberts, and Michael D Ernst. Evaluation of version control merge tools. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, pages 831–83, 2024.

[71] Georg Seibt, Florian Heck, Guilherme Cavalcanti, Paulo Borba, and Sven Apel. Leveraging structure in software merge: An empirical study. *IEEE Transactions on Software Engineering*, 48(11):4590–4610, 2021.

[72] Saad Shafiq, Christoph Mayr-Dorn, Atif Mashkoor, and Alexander Egyed. Balanced knowledge distribution among software development teamsâobservations from open- and closed-source software development. *Journal of Software: Evolution and Process*, 36(8):e2655, 2024.

[73] Bo Shen, Wei Zhang, Haiyan Zhao, Guangtai Liang, Zhi Jin, and Qianxiang Wang. Intellimerge: A refactoring-aware software merging technique. *Proc. ACM Program. Lang.*, 3(OOPSLA), oct 2019.

[74] Bowen Shen, Muhammad Ali Gulzar, Fei He, and Na Meng. A characterization study of merge conflicts in java projects. *ACM Transactions on Software Engineering and Methodology*, 32(2):1–28, 2023.

[75] Danilo Silva, Nikolaos Tsantalis, and Marco Tulio Valente. Why we refactor? confessions of github contributors. In *Proceedings of the 2016 24th acm sigsoft international symposium on foundations of software engineering*, pages 858–870, 2016.

[76] Leuson Da Silva, Paulo Borba, Wardah Mahmood, Thorsten Berger, and JoÃ£o Moisakis. Detecting semantic conflicts via automated behavior change detection. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 174–184, 2020.

[77] Leonardo Sousa, Diego Cedrim, Alessandro Garcia, Willian Oizumi, Ana C Bibiano, Daniel Oliveira, Miryung Kim, and Anderson Oliveira. Characterizing and identifying composite refactorings: Concepts, heuristics and patterns. In *Proceedings of the 17th International Conference on Mining Software Repositories*, pages 186–197, 2020.

[78] Sandro Sperandei. Understanding logistic regression analysis. *Biochemia medica*, 24(1):12–18, 2014.

[79] Nikolaos Tsantalis, Ameya Ketkar, and Danny Dig. Refactoringminer 2.0. *IEEE Transactions on Software Engineering*, 48(3):930–950, 2022.

[80] Nikolaos Tsantalis, Matin Mansouri, Laleh M Eshkevari, Davood Mazinanian, and Danny Dig. Accurate and efficient refactoring detection in commit history. In *Proceedings of the 40th international conference on software engineering*, pages 483–494, 2018.

[81] Gustavo Vale, Heitor Costa, and Sven Apel. Predicting merge conflicts considering social and technical assets. *Empirical Software Engineering*, 29(1):24, 2024.

[82] Gustavo Vale, Claus Hunsen, Eduardo Figueiredo, and Sven Apel. Challenges of resolving merge conflicts: A mining and survey study. *IEEE Transactions on Software Engineering*, 2021.

[83] Bernhard Westfechtel. Structure-oriented merging of revisions of software documents. In *Proceedings of the 3rd international workshop on Software configuration management*, pages 68–79, 1991.

[84] Beiqi Zhang, Peng Liang, Xiyu Zhou, Aakash Ahmad, and Muhammad Waseem. Demystifying practices, challenges and expected features of using github copilot. *International Journal of Software Engineering and Knowledge Engineering*, 33(11n12):1653–1672, 2023.

[85] Xiyu Zhou, Peng Liang, Beiqi Zhang, Zengyang Li, Aakash Ahmad, Mojtaba Shahin, and Muhammad Waseem. Exploring the problems, their causes and solutions of ai pair programming: A study on github and stack overflow. *Journal of Systems and Software*, 219:112204, 2025.