Received: 23 January 2021 | Revised: 8 January 2022 | Accepted: 16 February 2022

DOI: 10.1002/smr.2447

SPECIAL ISSUE - EMPIRICAL PAPER

Journal of Software: Evolution and Process WILEY

Exploring the relationship between refactoring and code debt indicators

Rusen Halepmollasi | Ayse Tosun

Faculty of Computer and Informatics Engineering, Istanbul Technical University, Istanbul, Turkey

Correspondence
Rusen Halepmollasi, Faculty of Computer and Informatics Engineering, Maslak, Istanbul 34469, Turkey.
Email: halepmollasi@itu.edu.tr

Abstract

Refactoring, which aims to improve the internal structure of the software systems preserving their behavior, is the most common payment strategy for technical debt (TD) by removing the code smells. There exist many studies presenting code smell detection approaches/tools or investigating their impact on quality attributes. There are also studies that focus on refactoring techniques, their relation with quality attributes, tool supports, and opportunities for them. Although there are several studies addressing the gap between refactoring and TD indicators, the empirical evidence provided is still limited. In this study, we examine the distribution of 29 refactoring types among the different projects and their relation with code smells or faults. We explore the refactoring types that are most commonly performed together and other activities performed with refactorings. We conduct a large exploratory study with automatically detected 57,528 refactorings, 37,553 smells, 27,340 faults, and 134,812 commits of 33 Java projects. Results show that some refactoring types are more commonly applied by developers. Our analysis indicates that refactorings usually remove or do not affect the code smells, and this contradicts with the previous studies. Also, the commits in which refactoring(s) is performed are three times more fault inducing than those without refactoring.

KEYWORDS
code smells, refactoring, software faults, technical debt

1 | INTRODUCTION

Software change management and maintenance are crucial activities of the development life cycle that gets more complex over time. Project teams, constrained by schedule, resources, and budget, get into debt to meet customer expectations, while they face additional costs later along the maintenance process.[1] *Technical debt* (TD) metaphor was first coined by Cunningham[2] in 1993 to explain the need for refactoring to non-technical stakeholders. He states that TD is "not quite right code" which we postpone making right, and working on not-quite-right code counts as the interest on the debt. Moreover, Tom et al[3] explored the TD metaphor and outlined the first framework to provide a consolidated understanding of the TD phenomenon and to discuss its positive and negative consequences. Although TD increases productivity and speed in the short term, its long-term impact on software artifacts at different stages of the life cycle, such as additional cost during maintenance, has been observed as more crucial.[4–6]

Debt is quite inevitable in software development,[7] and it is even deliberately incurred under some conditions.[4,8] It is usually incurred when the effort to build the software in the shortest possible time using the least amount of resources is preferred over the effort to maintaining the quality attributes of the system.[6] In companies whose main goal is to deliver business value, there is a need to choose between activities such as adding new features and fixing defects and repaying accumulated TD.[7,9] Due to the growing complexity and size of software systems over time,

the negative impact of TD on systems becomes more critical, and refactoring as the most common strategy for TD repayment becomes more significant. Therefore, conceptually knowing the accumulated TD that damages business value through software systems is not enough to prevent and repay it. According to Brown et al,[10] it is essential that the long-term management of the debt should be taken into consideration to obtain short-term benefits in increasingly complex software systems. Moreover, not only the interest of the debt but also the principal have to be paid by managing both intentional and unintentional TD, that is, knowing how to deal with it. In recent years, research studies on TD management has been prominent on enhancing flexibility, increasing productivity, and reducing the negative impact of debt, that is, keeping a balance between the cost and value of software systems.[11–13]

A recent systematic literature review[11] presents a set of activities for TD management that prevent potential TD from being incurred or that keep it at a reasonable level. Authors report that TD identification and TD repayment are among the most mentioned and investigated TD management activities. TD identification detects accumulated various debt types due to intentional or unintentional technical decisions in a software system through certain approaches such as static code analysis. TD repayment mitigates or removes the principal and/or interest of the identified debt by solving its detected causes through various techniques, for example, refactoring, rewriting, and re-engineering, without changing the behavior of the system. Also, the authors state that *code debt* and its management have received the highest attention from the software engineering community.[11]

Code smells and software faults are considered and studied as the most common indicators of code debt. Moreover, the most popular repayment strategy for them is refactoring, because it is supported by a large number of tools,[14] and it is able to mitigate the most common types of debt in systems such as code debt and design debt.[15] In recent years, code debt, the indicators identifying it, and refactoring as its repayment strategy are increasingly discussed. There are several studies presenting code smell detection approaches, code smell detection tools, and/or investigating the impact of code smells on quality attributes, specially maintenance and evolution.[16–20] Also, various studies have been published on refactoring techniques, their impact on quality attributes, tool support, and opportunities for them.[21–26]

A recent tertiary literature review[15] highlights that there is still a gap in identifying the relationship between refactoring and code debt indicators, and to the best of our knowledge, only a few studies address this gap.[27–31] Although these studies discuss the relationship between refactoring and code debt indicators, they are limited in terms of the analyzed code debt indicator; the size of the studied projects; use of tools for detecting refactoring actions, code smells, and/or faults; the granularity of the observed phenomenon; and other issues related to introduction of code smells.

We listed those studies in Table 1 with respect to the mentioned issues. Three of these studies[27–29] focus on the correlation of refactoring activities with faults, while two of these studies[30,31] focus on the relation between refactorings and code smells. However, no study investigates the refactoring activities, and their relationships with the other coding activities, and debt indicators such as code smells and faults altogether. Furthermore, the size of the investigated projects is limited, and this imposes a threat on the generalizability of the findings. While the studies by Weißgerber and Diehl[27] and Bavota et al[28,30] were performed on three open-source Java projects, the study of Gatrell and Counsell[29] was conducted on a subset of an industrial C# project only. Moreover, although Cedrim et al[31] analyzed the change history of 23 open-source Java projects, they identified 16,566 refactoring activities only.

The analysis made with the support of tools are often able to achieve a better level of accuracy and are more valid in terms of replication and comparison with other studies.[14] Nevertheless, the refactoring activities were identified without using any tools by Weißgerber and Diehl,[27] and identified with a custom made, bespoke tool by Gatrel et al.[29] Also, Bavota et al[28,30] detected refactorings with a release-based tool, namely,

TABLE 1 Relevant studies with similar goals to ours

	Weißgerber and Diehl[27]	Bavota et al.[28]	Gatrell and Counsell[29]	Bavota et al.[30]	Cedrim et al.[31]	Our study
Analyzed relation	Refactoring—faults	Refactoring—faults	Refactoring—faults	Refactoring—code smells	Refactoring—code smells	Refactoring—code smells and Refactoring—faults
Dataset	3 open-source Java projects	3 open-source Java projects	1 industrial C project	3 open-source Java projects	23 open-source Java projects	33 open-source Java projects
Refactoring detection	Class-level method by introduced	RefFinder (12,922 refactorings)	A bespoke tool by (1791 refactorings)	RefFinder (12,922 refactorings)	RMiner (16,566 refactorings)	RMiner (57,528 refactorings)
Number of refactoring types	6 refactoring types	52 refactoring types	15 refactoring types	52 refactoring types	10 refactoring types	29 refactoring types
Code smell/fault detection	Faults: bug databases and bug report mails	Faults: SZZ	Faults: the fault history in the control system	Code smells: rule based	Code smells: rule based	Code smells: Ptidej; Faults: Open SZZ

RefFinder.[32] Thus, there is a possibility that some refactoring activities cannot be detected when many changes occur between these versions as the tool detects refactoring activities between consecutive releases. Also, it is not possible to know if the refactoring, detected between two releases, is actually implemented on the commits that induced the fault(s). Similarly, in four of the studies, faults and code smells affecting the code components were identified either without tool support (Weißgerber and Diehl[27] and Gatrell and Counsell[29]) or by rule-based approaches (Bavota et al[28] and Cedrim et al.[31]).

Previous works also indicate that some refactoring activities can be neutral and some can be more harmful than others when analyzing its impact on the density of code smells or faults.[28,30,31] However, most of those works did not consider that even if a refactoring type introduces new code smells or faults, it may not be relevant in software projects where the occurrence of that refactoring activity is limited. In other words, all of the studies except Cedrim et al[31] do not investigate the distribution of refactoring activities in projects. Also, in the study,[31] the empirical results are not supported by statistical tests to verify the differences between distributions in projects for each refactoring type. Moreover, according to Tufano et al,[20] code smells are generally introduced by developers when adding new features or implementing enhancing existing ones because of maintenance activities. Therefore, in the commits where the refactoring activities are considered harmful, there is a possibility that developers dealt with other issues. For such cases, it is crucial to investigate whether the other activities are performed with refactoring activities, in order to better understand the cause of introducing code smells. Yet no studies have investigated this aspect.

In the scope of this paper, we aim to cope with the limitations of previous studies, and investigate the refactoring activities, and their relationship with code debt indicators in a larger context, at a coarser granularity, and with a more detailed analysis on the software changes. To accomplish this, we explore so far the largest and the most comprehensive Technical Debt Dataset[33] that consists of analysis of 134,812 commits from 33 projects. The data includes commit and file-based information about code smells, refactorings, all the Jira issues, and the fault-inducing commits extracted with the SZZ algorithm. The research on TD identification has also gained momentum with the introduction of this new dataset.[34–37] Using Technical Debt Dataset,[33] we conducted an exploratory study on 33 open-source Java projects in which 57,528 refactoring activities including 29 different refactoring types were detected at commit level by RMiner tool, 669 fault-inducing commits, and 8538 fault-fixing commits were identified by SZZ, and 37,553 code smells were extracted from the commits by Ptidej. Using the links between all these measures, commits, and files of the software projects, we address the following main research question: "To what extent is refactoring related to code debt indicators (code smells and faults) in the software projects?" We refined the main research question with a set of subquestions[38] and formulated the following complementary research questions:

- *RQ1: What kind of refactoring activities are commonly observed across software projects?*
  - *RQ1.1: Which refactoring activities are commonly co-occurred?*
- *RQ2: Does refactoring activities reduce the density of code smells?*
  - *RQ2.1: What are the other activities performed with refactoring activities that touch smelly commits?*
- *RQ3: How often do refactoring activities relate to fault-inducing/fault-fixing activities?*

We define RQ1 to observe which type of refactoring activities developers frequently perform in software projects. We also would like to see if some refactoring activities are performed together during software changes. Moreover, we define RQ2 to analyze whether refactoring activities reduce the density of code smells, as prior studies show conflicting results regarding the effect of refactoring on the removal of code smells.[30,31] We also check if the effect of refactoring on a software change can be individually mapped to a code smell. Finally with RQ3, we would like to understand the distribution of fault-fixing and fault-inducing activities among the refactoring activities. Our results show that some some refactoring types (e.g., Move Class, Extract Method, Rename Method/Variable, and Pull Up Method) are more frequently applied, while some specific types (e.g., Replace Attribute, Parameterize Variable, Move And Rename Attribute, and Rename Package) are hardly ever applied. Furthermore, our findings show that refactorings usually remove or do not affect the code smells, and this mostly contradicts with the previous studies. Another observation of our study is that other activities were also implemented by developers in commits where refactoring actions were implemented. Also, some of those (e.g., new task development) might be associated with the introducing of new code smells. Moreover, fault-inducing activities together with refactoring activities are rarely observed. However, we find that the commits in which refactoring activities are performed are three times more fault inducing than no refactoring activities are performed. Finally, while refactoring and fault-fixing activities are performed in the same commits in only 10% to 12% of the cases, in the rest of the cases, refactorings are performed separately from the fault-fixing changes.

1.1 | Structure of the paper

Section 2 summarizes previous works on refactoring, code debt indicators, and their relationships. In Section 3, we present the investigated research questions. Section 4 describes our empirical study design in which we present data preprocessing and data analysis phases to answer each RQ. The obtained results of the study over the considered projects are reported and discussed in Section 5, while in Section 6, we analyzed and discussed the threats that could affect the validity of our study. Finally, Section 7 concludes the paper.

2 | RELATED WORK

The research community has widely studied refactoring and code debt indicators such as code smells, faults from different perspectives. In this section, we discuss previous studies related to code smells, faults, and refactoring.

2.1 | Evolution of code smells

Code smells are violations of coding design principles and indicates that something is wrong deeper into the software system rather than the visible problem itself.[39] Different from other defects, a code smell is quick to spot and gives a hint that the code has to be examined deeply to see whether there is an underlying problem, rather than a certainty.[20,40]

Fowler made an important contribution to the literature by organizing and categorizing 28 types of smells.[39,40] Lacerda et al[15] report that Duplicated Code/Code Clone, God/Large Class, Feature Envy, Long Method, and Long Parameter List are the most studied smells presented by Fowler for several reasons, such as tools available for their detection, the frequency of smell occurrence, and popularity among practitioners.

Code smells, one of the symptoms of TD, have negative consequences during the system evolution and can negatively affect the overall maintainability of the system. In addition to suggesting tools and approaches to detect code smells, over the recent years, the research community has extensively investigated the evolution of code smells,[16,41–43] their impact on change and/or fault proneness of source code elements,[18,19,44–46] and why/when/how code smells are introduced.[20]

Olbrich et al[16] analyzed the evolution of God Class and Shotgun Surgery code smells detected by the application of an automated approach, on two large-scale open-source projects and investigated their impact on the frequency and size changes. According to their results, there are different phases of software evolution during which the number of smells could be increasing or decreasing, and those phases do not depend on the size of the systems. Furthermore, the authors observe that the classes with the examined smells have a higher frequency of change than classes without smells, and hence, those classes may need more maintenance.

Chatzigeorgiou and Manakos[43] also analyzed the presence and evolution of code smells, namely, Long Method, Feature Envy, God Class, and State Checking, identified by a tool on two open-source software systems. The results of their study show that (i) the total number of instances of code smells increases during the evolution of software systems; (ii) in most cases, when code smells appear in a version, they persist in the latest version of the systems as well; (iii) code smells not only appear during software evolution but also appear in a module right from its initial construction as a direct consequence of inefficient initial analysis and design activities; and (iv) developers rarely perform appropriate refactoring operations to remove code smells.

Khomh et al[18] investigated whether classes with code smells are more change prone and fault prone than classes without smells. They provide empirical evidence that the classes infected by smells are more change and fault prone with respect to the classes not infected by any smell. Similarly, Palomba et al[19] empirically investigated the relations between the occurrence of code smells in software systems and change proneness and fault proneness. They confirm the results provided by Khomh et al[18] on a larger set of code smells and software systems. However, they observe that applying refactoring to remove code smells does not always reduce the change proneness and fault proneness of classes.

Tufano et al[20] argue when code smells are introduced into the projects and discuss the factors that led to the introduction of smells over the version histories of 200 open-source projects. The authors analyzed over half a million commits to identify issues associated with commits that introduced smells on software projects. They report that about 65% of code smells are introduced during enhancement activities such as implementing new features or extending the functionality of the existing ones, not as a result of its evolution. Moreover, their results indicate that 80% of smells were not removed afterwards, and only 9% were removed as a consequence of refactoring activities.

2.2 | Refactoring activities on code changes

In order to remove a detected code smell, compatible refactoring activity is applied for that smell.[39] The term refactoring was first defined in 1990 by Opdyke[47] as reorganization strategies that support changes in the software components. Fowler provides a good overview of the refactoring and proposes different refactoring activities to improve code design and remove code smells.[39] According to Fowler,[40] refactoring is a change made in the internal structure of the software without changing the observable behavior to improve external quality.

Several researchers studied the detection of changes and refactoring activities occurred in multiple versions of software projects. Demeyer et al[21] present four heuristics based on source code metrics to identify three refactoring activities. Each heuristics is represented as combination of change metrics, which are on method size, class size, and inheritance, of two subsequent system snapshots. According to their preliminary results on three case studies, the approach can be applied in practice to focus on which parts of a system have changed and how they have changed.

Xing and Stroulia[22] propose UMLDiff algorithm based on design-level changes and developed a tool, namely, JDEvAn,[23] that automatically detects different kinds of refactoring activities by implementing the algorithm. The authors also reported a detailed case study on three pairs of subsequent major Eclipse releases and studied what fraction of code modifications are refactorings and which types are the most frequent ones.[24] Their state that about 70% of structural changes may be due to refactorings that involve a variety of restructuring types, ranging from simple element renamings to substantial reorganizations of the containment and inheritance hierarchies.

Murphy-Hill et al[26] performed an extensive study of refactoring application on a variety of datasets spanning a large number of developers. Among the various observations of the study, one shows that the developers often perform refactoring activities. Specifically, the authors state that the usage of floss refactoring tactic is more common than the usage of root-channel refactoring tactic among developers. In previous work, Murphy-Hill and Black[25] present the terms floss refactoring and root-channel refactoring. Floss refactoring utilizes daily refactoring techniques occurs before the changes such as adding new features or bug fixing without much change in the structure of the code. Root-canal refactoring consists of a costly and planned pure refactoring to improve the structure of the source code.[25] Another significant finding of the study is that many refactoring activities are performed manually without the help of tools, which is both time consuming and error prone.[26]

2.3 | Relation between refactoring and code debt indicators

As common wisdom, refactoring has been adopted to eliminate code smells identified through different detection approaches and to prevent the occurrence of faults. However, in the last two decades, while the body of research on both refactoring and fault or smells has been growing rapidly, there are a limited number of studies empirically analyzing the relation between refactoring and code smells or faults.[15,30,31]

Weißgerber and Diehl[27] studied the correlation between refactoring activities and bug reports opened in the following 5 days and whether transactions with a high refactoring ratio are less error prone than others. For detection of class-level refactorings[48] in the commit history of CVS repositories, they mined the version histories of three open-source Java projects. Then, they computed the ratio of refactoring changes over other changes and relate those to bug data. The paper reports that in all three projects, there are no days that only contain refactorings, that is, refactorings often occur together with other types of changes. Moreover, they found no strong correlation: While it is more common to observe that bug ratios do not increase after high ratios of refactoring changes, there has also been cases where bug ratios increased after refactorings.

Bavota et al[28] analyzed three different datasets trying to understand to what extent refactoring activities induce faults. They used an open-source tool to automatically detect 15,008 refactoring operations and used the SZZ algorithm to determine whether it is likely that refactorings induced a fault. According to their results, refactorings involving hierarchies such as pull-down method induce faults very frequently, whereas other kinds of refactorings are likely to be harmless in practice.

Gatrell and Counsell[29] investigate and report whether refactoring is beneficial in terms of maintenance of software systems and in turn leads to less change-prone and fault-prone classes. For this aim, they studied a subset of a commercial C# system for a 12-month period consisting of 7489 classes. They identified 1791 refactorings detected during the middle 4 months of the subset that is decomposed into three 4-month intervals and then examined the same classes in three 4-month periods in order to investigate the impact of the detected refactoring activities on change proneness or fault proneness. The authors also compared those classes with the remaining classes of the system in which no refactoring operations were performed in the same period. They found that classes in which refactoring was applied are less change prone and fault prone during and after the period of refactoring, compared with before the period of refactoring. Also, the study states that simpler refactoring types are common, and more complex structural refactoring types are rare.

Bavota et al[30] analyzed whether refactorings occur on code elements that certain indicators (e.g., structural quality metrics, the code smells detected by tools) suggest a need for refactoring, along with the change history of three Java open-source projects. They collected refactorings using the projects' major versions and discussed that refactoring processes generally do not focus on code components for which software quality metrics suggest that they may need refactoring. They report that 42% of the refactoring actions are applied in elements with a code smell, in which only 7% of refactorings remove smells.

Cedrim et al[31] mined the evolution history of 23 open-source projects to investigate how often refactoring activities affect the density of code smells. The study based on the analysis of 10 different refactoring types collected between commits and their effects on 13 different types of code smells. According to their results, although 79.4% of the refactoring operations are performed in elements with code smells, 9.7% of the operations removed code smells and 33.3% induced new smells. Moreover, they characterize refactoring-smell patterns and observe those typical refactoring activities such as Move Method and Pull Up Method induce the surfacing the God Class smell.

3 | RESEARCH QUESTIONS

The main objective of this study is to empirically explore refactoring activities and types, and investigate the relationship between refactoring activities and code debt indicators. In this paper, we refer to code refactoring activities performed on source files as refactoring activities, such as

renaming, extracting, and pulling up methods/classes. In this context, we define a primary research question: "To what extent is refactoring related to code debt indicators (code smells and faults) in the software projects?" To clarify the research question and provide the scientific underpinning, we combine a set of different subquestions each of which focuses on a different research strategy/methodology and classify them according to the guidelines provided by Easterbrook et al.[49] To this end, the following research questions are formulated and studied:

- RQ1: *What kind of refactoring activities are commonly observed across software projects? (Frequency and distribution)*
  - *RQ1.1: Which refactoring activities are commonly co-occurred? (Frequency and distribution)*

  With RQ1, we aim to provide an insight about the refactoring types that developers frequently perform over software changes on software systems. This preliminary research question is motivated by the need to observe whether frequently performed refactoring types differ among software projects. This provides the context on which the results of other research questions can be interpreted. Moreover, although refactoring activities are generally applied to improve the maintainability, understandability, and readability of code, different types of refactoring have different effects. In this context, reporting the most commonly occurred refactoring types may explain the reasons and motivation behind the developers' decision to implement specific refactoring activities. Therefore, we believe that the findings for RQ1 may help researchers to investigate why some refactoring types are performed more than the others(e.g., to remove code smells, to ensure maintainability, to improve software quality, etc.), and whether it is related to the tool support that developers have during refactoring through future studies. Also, by answering RQ1.1, we would like to identify which refactoring types are usually co-occurred in the analyzed software projects. This would also show if refactoring types are more likely applied in combination with each other or individually.
- *RQ2: Does refactoring activities reduce the density of code smells? (Relationship)*
  - *RQ2.1: What are the other activities performed with refactoring activities that touch smelly commits? (Description and classification questions)*

  The goal of the RQ2 is to analyze refactoring operations occurring over the history of a software project with the aim of investigating whether refactoring activities are related to the density of code smells. The question examines the statistical relationship between refactoring and code smells and hence does not target a causality. Previous work[31] found that refactoring activities often do not remove code smells and even introduce new code smells into the software systems. Authors identified the consecutive commits according to the order of they were merged into the main branch to generate the commit history of software projects. In other words, they analyzed the refactoring activities–code smells relationship between a commit (c_{i+1}) merged into the main branch and its parent commit (c_i) merged into the main branch before that commit (c_i). However, in such an analysis, we observed that the components, that is, modified files, in consecutive commits are often different. This poses a threat on the previous findings, as the code smells in some components which were refactored in c_{i+1} are compared with the smells in other components modified in c_i. In this paper, we would like to investigate the effect of refactoring on code smell density with a different methodology explained in Section 4.2.2. Our methodology ensures that the refactorings and code smells applied to the same files are compared through descriptive statistics.

  Furthermore, with RQ2.1, we aim to empirically investigate whether developers address other tasks while they are performing refactoring activities that touch smelly commits. In other words, it is interesting for us to observe if developers implement other tasks such as a new feature or improvement on commits while they are performing refactoring activities. If there are several tasks addressed in a single commit, then the code smells introduced in that commit should not be directly matched to the refactoring, because code changes cannot be specifically distinguished for the refactoring only. Thus, we also interpret the results of RQ2 by looking at this perspective.
- *RQ3: How often do refactoring activities relate to fault-inducing/fault-fixing activities? (Frequency and distribution)*

  Refactoring activities are applied to prevent the occurrence of faults by eliminating code smells without changing the behavior of software systems. However, in some cases, while developers perform refactoring, they unintentionally may induce faults into the system.[27,28] Moreover, in some cases, while the developers fix the faults, they may intentionally perform refactoring on the code fragment that is too bad to be fixed.[50] This research question investigates the relation between commits on which refactoring activities are performed and fault-fixing and/or fault-inducing commits.

4 | STUDY DESIGN

4.1 | Dataset

The dataset used in this study is the Technical Debt Dataset originally created by Lenarduzzi et al.[33] It consists of 33 projects belonging to a well-known ecosystem, namely, Apache Software Foundation (ASF) repository* that are older than three years and developed in Java. Each of these projects contains at least 500 commits, more than 100 classes, and has a minimum of 100 issues. The dataset consists of measurement data

TABLE 2 SQL tables of Technical Debt Dataset used in this study

SQL table name	Descriptions
Git Commits	The commit information retrieved from the git log.
Git Commits Changes	The changes performed in each commit, including the old path of the file, the new path of the file, and the type of changes reported in the git log.
Refactoring Miner	List of refactoring activities applied in the commits.
SZZ Fault-Inducing Commits	The results from the execution of the SZZ algorithm.
Jira Issues	Jira issues for the analyzed projects.
Sonar Issues	All of the SonarQube issues, the antipatterns, and code smells detected by Ptidej.

extracted through four tools, namely, PyDriller,[51] Ptidej,[52] RMiner,[53] and SonarQube,[†] executed on the commits performed between March 1, 2000, and July 1, 2019. Moreover, the SZZ algorithm[54] has been used to identify the fault-inducing and fault-fixing commits from the version history for each of the projects. The dataset contains various important information about projects on nine SQL tables: Projects, Git Commits, Git Commits Changes, Refactoring Miner, SZZ Fault-Inducing Commits, Jira Issues, Sonar Issues, Sonar Measures, and Sonar Rules.

Because we focus on the analysis of the relationship between refactoring activities and code debt indicators, namely, code smells and faults, we selected only the tables related to faults, issues, and refactoring and listed them in Table 2. In Section 4.2, we explain the corresponding tables and the units of the analysis in detail for each RQ.

4.1.1 | Additional dataset package

When performing our analysis specifically for RQ2, we had to crawl additional data from the projects' repositories. These data consist of the following:

- All files modified in the commits, their modified dates and commit hash values, and an indication of whether each file at the specified date and commit includes code smells. This is new information, because in the Sonar Issues table, code smells are reported on the commits over all the files regardless of whether these files were modified in those commits or not. So we may encounter that a file has a code smell in a commit although that file is not modified in that commit. To solve this, we matched the files with the corresponding commits in which they were affected by the code smells.
- Commit hash values and the issue IDs, issue types associated with these commits. This link was also not available in the existing Jira and Git Commits tables, but we need this information to understand the type of issues related to the changes in the commits if an issue ID is provided in the commit messages.

In Section 4.2.2, particularly in *Code Smells* and *Other Issues* subtitles, we present our methodology in more detail. The additional dataset we extracted is also publicly available through CSV in Halepmollasi and Tosun.[55]

4.2 | Data preprocessing and data analysis

To answer all RQs, we need the refactoring activities detected over the change histories of all the software projects. On the other hand, the rest of the data tables and associated columns in those differ depending on the RQ. Thus, this section reports our methodology for each RQ in two phases: (i) data preprocessing in terms of picking the required tables, filtering the associated units of analysis, and if necessary, collecting new data, and (ii) data analysis methods.

4.2.1 | What kind of refactoring activities are commonly observed across software projects? (RQ1)

Preprocessing

To address RQ1 and RQ1.1, we use the Refactoring Miner table that includes 57,528 refactoring activities on 11,723 unique commits over 33 projects. In this table, there are 29 refactoring types, and some examples of them are *Extract Method*, *Extract Superclass*, *Move And Rename Attribute*, *Move Class*, *Move Method*, *Pull Up Attribute*, *Pull Up Method*, *Rename Class*, *Change Package*, and so on.

Refactoring activities

The table Refactoring Miner in TD dataset contains the list of refactoring activities applied in the repositories through an open-source tool, namely, RMiner.[53] The tool identifies different types of refactoring activities in the history of Java projects through the application of a two-step approach. First, it extracts classes, methods, and fields that are added/removed/replaced between two successive commits based on a lightweight version of the UMLDiff[22] algorithm. Next, the tool extends and uses a set of rules defined by Biegel et al[56] to discriminate the different types of refactoring activities. Additional rules were able to detect refactoring activities with 98% and 87% of precision and recall, respectively.[53]

Analysis

We first counted the number of each of the 29 refactoring types performed in all projects and divided them by the total number of refactoring activities to get which types of refactoring activities are applied more by the developers over all 33 projects. This ratio gives us a general view over all the projects about the distribution of different refactoring types. Moreover, we repeated this calculation for each of the projects individually to investigate whether the refactoring types are common and often performed across different software projects.

When comparing the distribution of refactoring activities across the projects, Scott–Knott ESD[57] test is applied to verify the differences between distributions. The test produces statistically distinct ranks at the significance level (default $\alpha = 0.05$) using a hierarchical clustering algorithm. The test ranks and clusters the refactoring types into significantly different groups such that the refactoring types in the same group have no significant difference, whereas the refactoring types clustered in distinct groups have significant differences in terms of their occurrence ratio. Therefore, the Scott–Knott test does not have an overlapping problem, that is, there is no possibility of one or more refactoring types to be classified in more than one group.

To answer RQ1.1, we extracted association rules to detect the co-occurred refactoring activities in the same commit. Association rule mining that is a widely used data-mining method generates all significant association rules in a given database.[58] Association rules generated are used to discover correlations and co-occurrences between items in a given database. In this paper, when identifying the patterns highlighting co-occurring refactoring types, the dataset is composed of a sequence of different types of refactoring activities performed together in the commits. An association rule between two disjoint refactoring types is defined in the form of $R_{left} \rightarrow R_{right}$, and it has two restrictions of $R_{left}, R_{right} \subset I$ and, $R_{left} \cap R_{right} \neq \emptyset$, where R_{left} is antecedent; R_{right} is consequent; and I represents the set of items. The rule indicates that if a refactoring type $r_i \in R_{left}$ is performed in a commit, then another refactoring type $r_j \in R_{right}$ is also performed within the same commit. In our study, the relationship between r_i and r_j is bidirectional, as opposed to the typical association rules in which the antecedent precedes the consequent. In other words, the relation identified by the association rule only indicates that the two refactoring types occur in the same commit, as opposed to one precedes the other in two consecutive commits. As expected, for a dataset with a lot of rule possibilities, the association rule mining also generates numerous rules, and hence, confidence and support measures that filter invalid rules are required. The support of an association rule is the percentage of transactions that contain $R_{left} \cup R_{right}$, that is, an indication of how frequently the itemset appears in the dataset. Confidence is the ratio of the number of transactions that contain $R_{left} \cup R_{right}$ to the number with R_{left}. That are expressed as

$$Sup(R_{left} \rightarrow R_{right}) = \frac{(R_{left} \cup R_{right})}{T}, Conf(R_{left} \rightarrow R_{right}) = \frac{(R_{left} \cup R_{right})}{R_{left}}, \quad (1)$$

where T is the total number of refactoring activities detected from the repository.

In this study, we employ a well-known algorithm, namely, Apriori,[59] for efficient association rule discovery. Apriori algorithm, in which the validity of an association rule is determined by its support and confidence, adopts support-based pruning approach to reduce the impact of the exponential growth of candidate itemsets. When obtaining the commonly co-occurred refactoring types, we determine the support value through experiments we performed on all support values of the resulting association rules.

4.2.2 | Does refactoring activities reduce the density of code smells? (RQ2)

Preprocessing

To address RQ2 and RQ2.1, we used *Git Commits*, *Git Commits Changes*, *Refactoring Miner*, and *Sonar Issues* tables. We removed six projects in total, that is, five projects (accumulo, ambari, atlas, aurora, and batik) for which there is no code smell analysis and one project (httpcomponents-client) with incorrect project name information, from the dataset. To increase the validity of our results, when investigating the impact of refactoring activities on the density of code smells, both refactoring and code smell analysis of a commit should be performed. Therefore, the examined refactoring activities of the projects were limited to the date range during which code smell analysis was also performed. Moreover, when answering RQ2.1, we used *Jira Issues* table to investigate the other activities performed together with the refactoring activities performed on smelly commits.

Commit–parent relation over files

The table Git Commits in the dataset contains a commit hash and its parent hash according to the order in which they were merged into the main branch when generating the commit history of software projects. However, we considered this commit–parent relationship cannot be used in our analysis. As seen in Figure 1, in two subsequent commit pairs based on the commit–parent relationship, there is a possibility that the files in the commit and its parent may differ from each other. Therefore, in our method, when analyzing the code smells of a file before the application of a refactoring, we do not consider this relationship. Instead, for each file *f* modified in a commit due to a refactoring, we have to find the latest commit in which file *f* was modified before the application of refactoring. For instance, in Figure 1, let *commit 123ee* be a commit that a refactoring is performed. The files modified on this commit are *file1* and *file4*. Therefore, we must select *commit 123bb* for *file1* and *commit 123cc* for *file4* to calculate their code smells before the application of refactoring. Please note that if we look at the consecutive pair of commits, we do not compare the same files: from Figure 1, on *commit 123ee*, *file1* and *file4* were changed while on its parent, that is, *commit 123dd*, *file5* and *file6* were changed.

Code smells

The table Sonar Issues contains both issues, that is, bugs, code smells, and vulnerabilities, detected with the SonarQube tool and code smells detected with the Ptidej tool on the same projects for a predefined time periods. In this study, we investigated only the commits that contain code smells and antipatterns detected with Ptidej, which is an open-source Java-based reverse engineering tool suite.[33,52] Ptidej performs on any object-oriented system through the use of the Pattern and Abstract-Level Description Language (PADL) metamodel, Primitives, Operations, Metrics (POM) framework, and SAD. PADL describes object-oriented systems and patterns with a unified language that includes all constructs found in object-oriented programming languages. POM implements more than 60 metrics on PADL metamodels and SAD requires the POM module to detect occurrences of code smells in the systems. Also, Ptidej has the module DEtection and CORrection (DECOR)[60] to identify code smells and antipatterns and employs a well-structured procedure by means of rule cards. In this study, we use only the code smells identified by Ptidej. The tool reports a total of 18 distinct code smells and antipatterns in the projects, such as LongMethod, LargeClass, LazyClass, Blob, AntiSingleton, and Speculative-Generality.

In the Sonar Issues table, we assume that the Ptidej tool has been executed in a periodical fashion, and all the code smells associated with all the files are reported. We checked whether each file modified and recorded in the Git Commits Changes table includes code smell(s) based on the information reported in Sonar Issues. Then, we matched the code smell(s) with the corresponding files' commit dates. This matching is done to obtain an improved distribution of code smells in the analyzed files and to increase the matching precision by linking the smells to the commits in which the corresponding file were last changed. An example of the matching is illustrated in Figure 2. Let us examine *file1* which was modified in commit *123bb* on March 1, 2010, and check if any code smell was induced into this file during that commit. When analyzing the code smells between March 1, 2010, and May 15, 2010, on *file1*, we observe that three code smells, namely, cs1, cs2, and cs3, were detected on April 9, 2010. This means that *file1* was affected by code smells on the last commit before the Ptidej report on April 9. Therefore, those code smells must be matched to the file version at the commit *123bb*.

Other issues

When analyzing RQ2.1, we used Jira Issues table that contains an issue_key, a creation date, and a resolution date. To identify the link between a commit and an issue, we first searched whether the commit message of each commit in the Git Commits table contains issue_key information. For example, the issue_key "ACCUMULO-109" is searched in the commit message "ACCUMULO-109: fix off-by-one error reading the max timestamp from the root tablet." If such a pattern was found, then we matched the issue_type corresponding to that issue_key in the Jira Issues

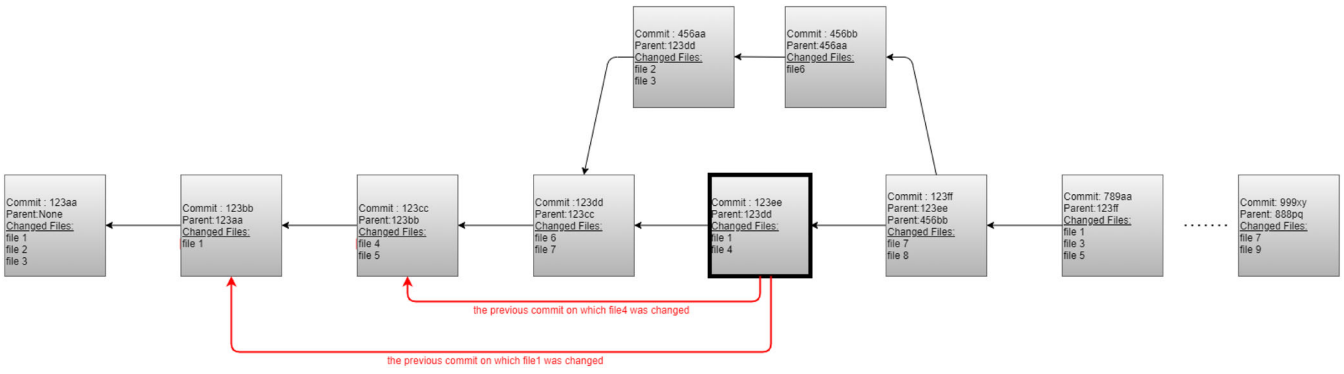


FIGURE 1 A visual representation of the commit–parent relation defined over files in this study

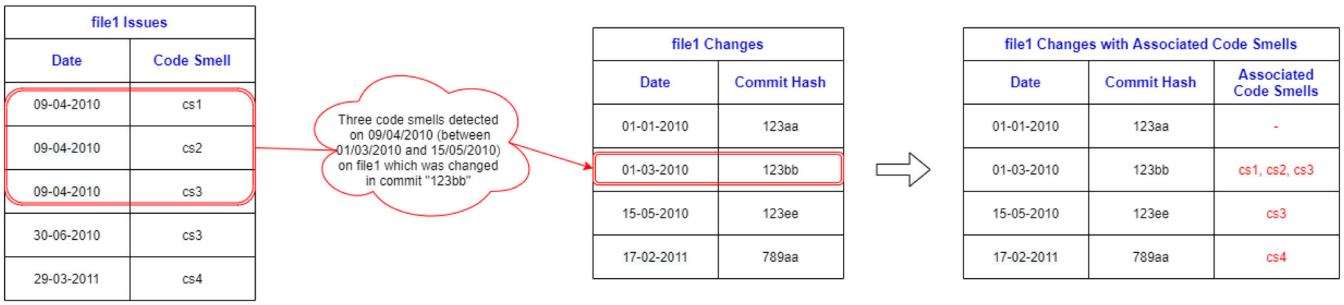


FIGURE 2 File–commit–code smell matching

table as the issue_type of that commit. When it was possible to identify the link between a commit and an issue, such issue_key was linked to the commit. We were able to automatically assign issue types to 42.4% (57,192 out of 134,812) of the commits. Next, we filtered out the commits in which refactoring activities were not detected. Consequently, we have obtained a distribution of issue types over all commits on which refactoring activities were performed.

Analysis

To answer RQ2, we analyze the impact of refactoring activities on code smell densities, using the data described in the data preprocessing. Knowing the list of files affected by each code smell in each commit, we are able to identify if refactoring activities (i) are able to remove code smells from a system, (ii) introduce code smells into a system, and (iii) do not affect the code smells. In other words, we can argue how given refactoring activity can be classified according to its impact on the density of code smells in files. As explained above, it is achieved by examining the presence of code smells before the application of refactoring (CSBR) and code smells after the application of refactoring (CSAR) over the files.

Suppose that R_i is a refactoring activity performed on a commit c_j in which a file f_k was modified. To categorize R_i, we calculate $CSAR_i[c_j(fk)]$ that is the number of code smells associated with f_k on c_j after the application of refactoring and $CSBR_i[c_{j-1}(f_k)]$ that is the number of code smells associated with f_k on c_{j-1} before the application of refactoring, where c_{j-1} is the previous commit in which f_k was modified. If (i) $CSBR - CSAR > 0$, then R_i is classified as *removing* refactoring activity because it decreases the number of smells associated with f_k; (ii) $CSBR - CSAR < 0$, then R_i is classified as *introducing* because it increases the number of smells affecting f_k; (iii) $CSBR - CSAR = 0$, then R_i is classified as *neutral* refactoring activity because performing R_i does not decrease or increase the number of smells in f_k.

To give more insight on the analysis, we present an example that refers to the reduction of code smells based on the previously depicted commit–parent relation in Figure 1. Suppose R_i is a refactoring activity (e.g., R_1: Refactoring Type1) performed in commit *123ee* in which *file1* was changed. Moreover, Figure 2 shows that *file1* in commit *123ee* was affected by the code smell *cs3*. In this way, the number of code smells affecting the *file1* in commit after the application of R_1 is calculated as $CSAR_1[commit123ee(file1)] = 1$. On the other hand, the previous commit modified *file1* is commit *123bb*. As *file1* was affected by three code smells at that commit, it is calculated as $CSBR_1[commit123bb(file1)] = 3$. Hence, R_1 is categorized as a *removing* refactoring activity for this selected file.

To address RQ2.1, after categorizing refactorings according to their impact on the density of code smells, we identified other activities performed together with these refactorings that touch smelly commits. Then, we computed the percentage of commits in which both refactoring activities were detected and other activities according to Jira issue_key such as new feature, improvement or task/subtask.

4.2.3 | How often do refactoring activities relate to fault-inducing/fault-fixing activities? (RQ3)

Preprocessing

To answer RQ3, we used the Refactoring Miner table to identify refactoring operations on the projects' commit history and the SZZ Fault-Inducing Commits table to determine whether refactorings were performed on fault-inducing or fault-fixing commits. The SZZ Fault-Inducing Commits table lists the results of the Open SZZ execution.[‡] The Open SZZ algorithm is an implementation of the SZZ algorithm[61] that is widely used to identify fault-inducing changes from a history of a software project.[62]

SZZ algorithm starts analyzing the fault-fixing commits, that is, commits known to fix a fault reported on an issue tracking system, to trace back to their fault-introducing commits, that is, commits containing the code changes that induce a fault into the system. We matched the commit hashes identified as fault inducing and fault fixing according to the SZZ table with the commit hashes in which refactoring activities were detected according to the Refactoring Miner table.

Analysis

To answer RQ3, we analyze to what extent commits in which refactoring was performed could also include fault-fixing and fault-inducing changes, and hence, for each commit, we calculate the following:

- PureR: only the number of commits a refactoring is performed, that is, these commits are not fault inducing or fault fixing.

$$PureR = [Refactoring \setminus (FaultInducing \cup FaultFixing)]. \quad (2)$$

- PureFF: the number of fault-fixing commits only (not fault inducing and not refactoring).

$$PureFF = [FaultFixing \setminus (Refactoring \cup FaultInducing)]. \quad (3)$$

- PureFI: the number of fault-inducing commits only (not fault fixing and not refactoring).

$$PureFI = [FaultInducing \setminus (Refactoring \cup FaultFixing)]. \quad (4)$$

- R&FI: the number of commits in which refactoring was performed, and the commits were also fault inducing.

$$R\&FI = [(Refactoring \cap FaultInducing) \setminus FaultFixing]. \quad (5)$$

- R&FF: the number of commits in which refactoring was performed, and the commits were also fault fixing.

$$R\&FF = [(Refactoring \cap FaultFixing) \setminus FaultInducing]. \quad (6)$$

- FF&FI: the number of fault-inducing and fault-fixing commits in which no refactoring activity was performed.

$$FI\&FF = [(FaultFixing \cap FaultInducing) \setminus Refactoring]. \quad (7)$$

- R&FF&FI: the number of fault-inducing and fault-fixing commits in which refactoring was also performed.

$$R\&FF\&FI = (Refactoring \cap FaultInducing \cap FaultFixing). \quad (8)$$

Then, we report the counts and percentages of commits belonging to each of these groups to see the ratio of refactoring activities performed on fault-inducing or fault-fixing commits over all the projects.

5 | ANALYSIS OF THE RESULTS

In this section, we report the analysis of the results achieved in our empirical study for each RQ, respectively, and highlight the major findings.

5.1 | What kind of refactoring activities are commonly observed across software projects? (RQ1)

For addressing RQ1, we present a bubble chart and list descriptive statistics on the refactoring activities occurring over the history of all software projects in the dataset. The analysis of the refactoring types that developers perform commonly in the projects provides the context on which the results of other research questions can be interpreted. We observe that the commits in which at least one refactoring activity was performed are only a small part of all the commits, with a percentage of approximately 9%. This proportion supports the previous results on the limited adoption of refactoring in practice.[17,30,63]

Examining in more depth, we present the distribution of refactoring types according to the number of the refactoring application over the selected projects in Table 3. The first column lists each refactoring type, and it is followed by the absolute numbers and percentages of the detected refactoring activity over all the commits across all the projects, respectively. Table 3 reports that the Move Class detected in 20.1% of the refactoring activities in the commits is the most frequently applied type. Furthermore, Figure 4 shows how the distribution of refactoring

TABLE 3 Occurrence of different refactoring types

Refactoring type	Number of ref.	Percentage of ref.
Change Package	52	0.1%
Extract And Move Method	1796	3,1%
Extract Class	259	0.5%
Extract Interface	260	0.5%
Extract Method	6712	11.7%
Extract Subclass	37	0.1%
Extract Superclass	653	1.1%
Extract Variable	1328	2.3%
Inline Method	1152	2.0%
Inline Variable	194	0.3%
Move And Rename Attribute	4	0.01%
Move And Rename Class	184	0.3%
Move Attribute	4728	8.2%
Move Class	11,582	20.1%
Move Method	4772	8.3%
Move Source Folder	825	1.4%
Parameterize Variable	129	0.2%
Pull Up Attribute	1635	2.8%
Pull Up Method	3328	5.8%
Push Down Attribute	484	0.8%
Push Down Method	954	1.7%
Rename Attribute	1122	2.0%
Rename Class	2675	4.7%
Rename Method	8251	14.3%
Rename Package	154	0.3%
Rename Parameter	1903	3.3%
Rename Variable	2200	3.8%
Replace Attribute	10	0.02%
Replace Variable with Attribute	145	0.3%
Overall	57,528	100.0%

types varies and which ones are more common among different software projects. In this figure, different colors represent different refactoring types, and the number of refactoring activities detected are used as the bubble sizes. As seen in Figure 3, Move Class is also the most common refactoring activity performed by developers in most projects (12 out of 33). According to Fowler,[40] moving a class to a package that is more related in form or functionality can help remove package-level dependencies and make classes easier to reuse. Our findings also show that developers aim to improve maintainability, understandability, and usability of source code while performing refactoring. This result is also in line with a prior study investigating the relationship between refactoring activities and quality attributes.[15] On the other hand, Move And Rename Attribute is the least applied types with a rate of 1% only.

Moreover, Rename Method and Extract Method are the other refactoring activities that are most commonly occurred, with 14.3% and 11.7%, respectively. Also, we report that Rename refactoring activities account for 28.4% of all refactoring activities. This result confirms what was previously reported by Murphy-Hill et al.[26] They stated that the Rename refactoring activities are among the most frequently applied refactoring activities, because they are mostly performed automatically with tools.[26] Automating the refactoring process includes identifying refactoring opportunities and performing refactoring activities.[14] Furthermore, Extract Method that is applied by creating a new method and copying the relevant code part into the new method is a step in many other refactoring actions, and this activity eliminates the rough edges in a software code. It is recommended for decomposing a large and complex method or eliminating duplicated code as the fewer the number of lines in a method, the easier it would be to understand. Extract Method that is associated with Duplicated Code, Long Method, Feature Envy, and so on assures a motivation for removing those code smells.[40] Du Bois and Mens quantitatively evaluate whether refactoring improves software

quality by comparing postrefactoring and prerefactoring measurements and found that Extract Method affects software maintainability.[64] Silva et al. report that Extract Method is one of the most applied refactoring activities as it is a versatile refactoring activity that is applied for several purposes such as improving maintainability, increasing software quality, and reducing code smells.[65] The analysis results we presented in Table 3 and Figure 3 also report that Extract Method is commonly applied by developers and confirm the results of Silva et al. When a class contains most of the data used by a method, it is moved to that class to reduce or eliminate the dependency on the other class the method is originally in. Therefore, Move Method affects software quality improvement and maintainability enhancement[66] and is a widely performed refactoring activity.

To support our empirical results, we compare the refactoring types with the Scott–Knott test by ranking their percentage of occurrence, clustering them in nonoverlapping groups, and identifying significant differences among the groups. Figure 4A shows the results for the Scott–Knott test among all refactoring types. Different colors (black, red, green, and blue) of the plot indicate different Scott–Knott test ranks, and the coloring of the lines represents distinct clusters of the refactoring types. When two refactoring types are not significantly different, they are illustrated with the same color. In the figure, the refactoring activities are ranked starting from the most common to least applied in all considered projects, that is, Rename Method, Move Class, and Extract Method shown in black are more commonly performed types. Although we see different occurrence ratios in Table 3 for Move Class (20%), Rename Method (14%), and Extract Method (11%) refactoring types, they are grouped in the same cluster according to the Scott–Knott results. Also, Extract Method is the only refactoring type that is applied at least once in each project. On the other hand, interestingly, we observe that 17 out of 29 refactoring types represented in blue including Push Down Method and Extract Super Class are the least performed refactoring types in most projects. These results statistically confirm the results shown in Figure 3. Also, this finding indicates that developers tend to implement refactoring activities that aim to remove common code smells[15,67] such as code clone, god class, long method, and which have tool support.[14,15,25]

Also, in Figure 4B, the projects considered are clustered into nonoverlapping groups in terms of the differences in the distribution of refactoring activities. If projects are not significantly different, they are clustered into the same group. In this context, there are four different groups on some of which various refactoring activities are applied and on some are hardly ever applied. In projects "Atlas" and "Mina-Sshd" with similar refactoring distributions, all refactoring types were applied by developers, except Rename Package. Besides, we observe that although "Beam" is the project with the most refactoring activity, 13 refactoring types were not performed at all, that is, 88% of the refactoring activities performed were limited to six types, namely, Extract Method, Move Attribute, Move Class, Move Method, Rename Class, and Rename Method.

To address RQ1.1, we identified five pairs of refactoring activities that are most commonly performed together by developers. Table 4 reports the co-occurrences of refactoring activities detected across considered projects according to the association rules. A frequent pattern represents

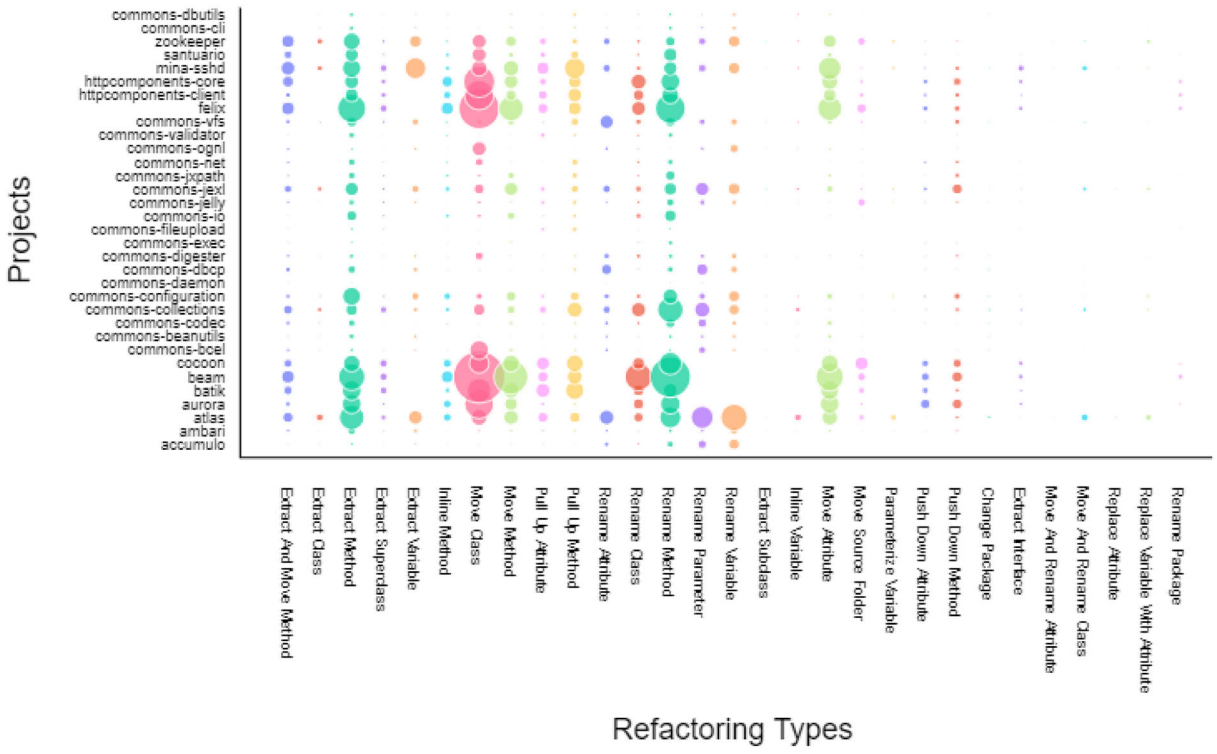
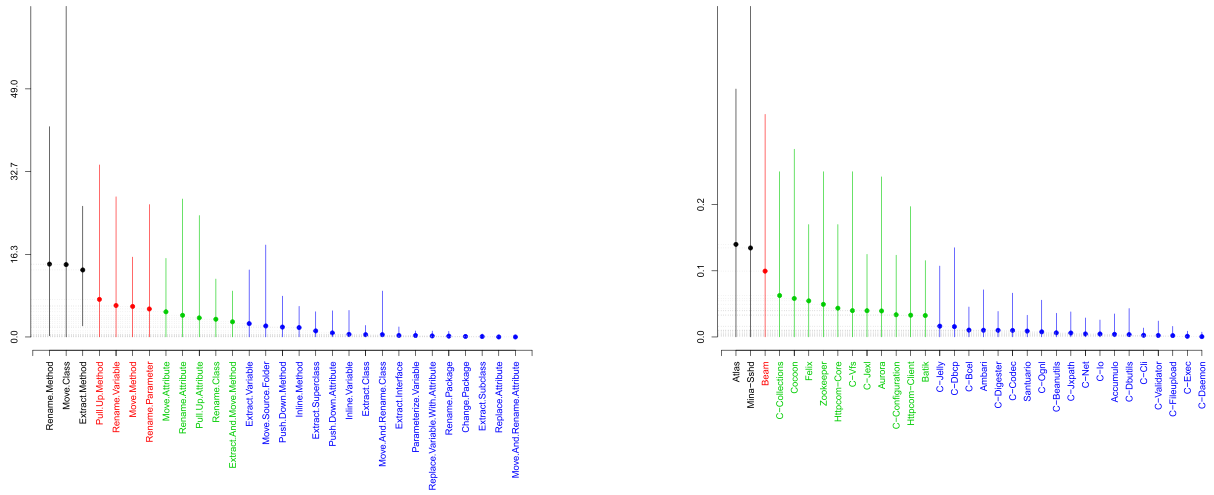


FIGURE 3 Distribution of refactoring types



(A) over refactoring types **(B)** over projects label

FIGURE 4 Scott–Knott test results for RQ1

TABLE 4 Co-occurred refactoring activities: Result of the association rule mining

Item Set 1	Item Set 2	Supp	Conf
Pull Up Attribute, Extract Superclass	Pull Up Method	0.018	0.88
Pull Up Attribute	Pull Up Method	0.025	0.76
Pull Up Attribute	Extract Superclass	0.021	0.63
Pull Up Method	Extract Superclass	0.027	0.62
Move Attribute	Move Method	0.04	0.43

a set of items whose support is greater than the predefined min_support threshold. Using a minimum threshold, we aim to identify meaningful association rules. The support values reported by the Apriori algorithm were experimented from 0.002 to 0.02 by steps of 0.001, and the best value is selected as 0.018. The co-occurrences are sorted based on the confidence values reported by the algorithm. Pull Up Method refactoring activity is very often performed (88%) with Pull Up Attribute and Extract Superclass. However, the item sets have 0.018 support value, that is, the item set appeared in the dataset with a frequency of 1.8%. This indicates that these three refactoring types are rarely implemented than the other refactoring types. For example, the Extract Superclass refactoring was implemented only at a percent of 1.1% (Table 3). However, not surprisingly, pairwise co-occurrence between this and Pull Up Attribute and Pull Up Method refactoring activities are commonly observed according to their confidence. When two different classes perform similar tasks using the same method and attributes, a shared superclass is created for them and all the identical attributes and methods are moved to this superclass to prevent duplicated code. Once a shared abstract superclass is created with Extract Superclass refactoring, Pull Up Attribute and Pull Up Method are also used to move the common functionality to a superclass. Also, we observed that Pull Up Attribute and Pull Up Method are performed together, even if Extract Superclass refactoring is not occurred. When two subclasses are created and extended independently of one another, they may have methods and attributes that perform similar work. In this case, similar methods are searched in superclasses and are regulated to match each other. When copying a method to the superclass, if the method has attributes that exist in the subclasses but not in the superclass, Pull Up Attribute refactoring is performed to provide getters and setters in the subclasses and to declarate the getters abstractly in the superclass.

According to the results in Table 4, although the highest number of co-occurrences based on the support value is between Move Attribute and Move Method that are similar refactoring activities, the identified co-occurrences are less based on the level of confidence. The reason is that those refactoring activities are frequently applied among the other refactoring activities, that is, Move Method and Move Attribute are reported with 8.3% and 8.2% in Table 3, respectively. This co-occurrence pair is naturally observed together: When a method is more frequently used in another class than the class in which it was created, this method can be moved into the class that uses it. In this case, the attributes used by the method should also be moved. In general, if those attributes are used only by that method, they must be moved.

5.2 | Does refactoring activities reduce the density of code smells? (RQ2)

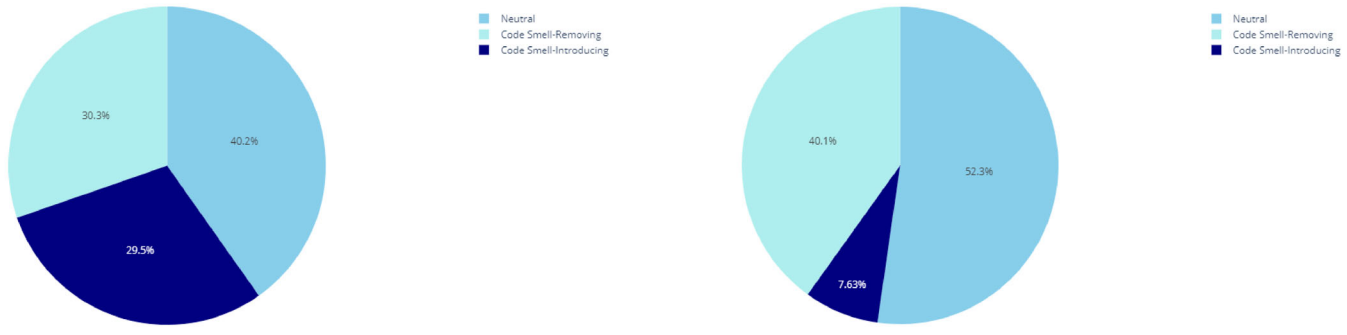
To answer RQ2, the number of detected refactoring activities were identified as 24,570 after filtering out the projects and selecting time periods including code smell analysis on the projects. We observe that 47.3% of those refactoring activities touch the components (files) that were affected by at least one code smell. The percentage simply indicates that refactoring activities could be performed by developers to remove code smells. In Figure 5, we present the overall results over all considered projects to assess to what extent the refactoring activities affect the density of code smells across the projects. Figure 5A shows that 30.3% of the refactorings removed the existing code smells, whereas 29.5% of the refactorings introduced new code smells into the analyzed projects. The rest (40.2%) of the refactorings are categorized as neutral that does not affect the number of smells in a commit, that is, neither introduce new smells nor remove existing ones. Please note that inconclusive refactorings, which remove a code smell affecting a file in a commit and at the same time introduce another new code smell in that file, are categorized as neutral because the overall activity in that commit is neutral. These results reveal that in most cases, the density of code smells does not change after refactoring or it is higher than the code smell density before refactoring.

However, this is a quite surprising finding, considering the common wisdom that refactoring activities are generally performed to reduce the density of code smells. In order to better understand such phenomenon, we analyzed the factors that introduce the code smells and found that some of the files with refactoring detected were newly added to the system, and hence, those were usually categorized as introduced if they affected by code smells. For instance, in commit *2035dc9*, Move Method refactoring was detected in the *UtfHelpper* class, belonging to *org.apache.xml.security.c14n.implementations* package of the Santuario project. As reported in Table 3, Move Method, one of the most commonly occur refactoring activities, is performed when a method is used more in another class than in its current class, by moving it to the class in which the method is used more frequently.[39,68] However, examining the change of the commit, we identified that the class was a new one added to the system. Certainly, in such cases, it is not possible to compare the effect of refactoring as there were no changes before refactoring activity.

Therefore, to allow a fair comparison, we have to distinguish between the cases in which newly added files are included into the analysis (Figure 5A) and they are not included (Figure 5B). As seen in Figure 5B, when newly added files are not included into the analysis, the percentage of refactorings categorized as introduced decreases to 7.6%, while the percentage of refactorings categorized as removed increases to 40.1%. In other words, although the effect of refactoring on code smell density is mostly inconclusive/neutral, the cases where refactoring activities decrease the code smell density are much more than those where refactorings increase the smell density.

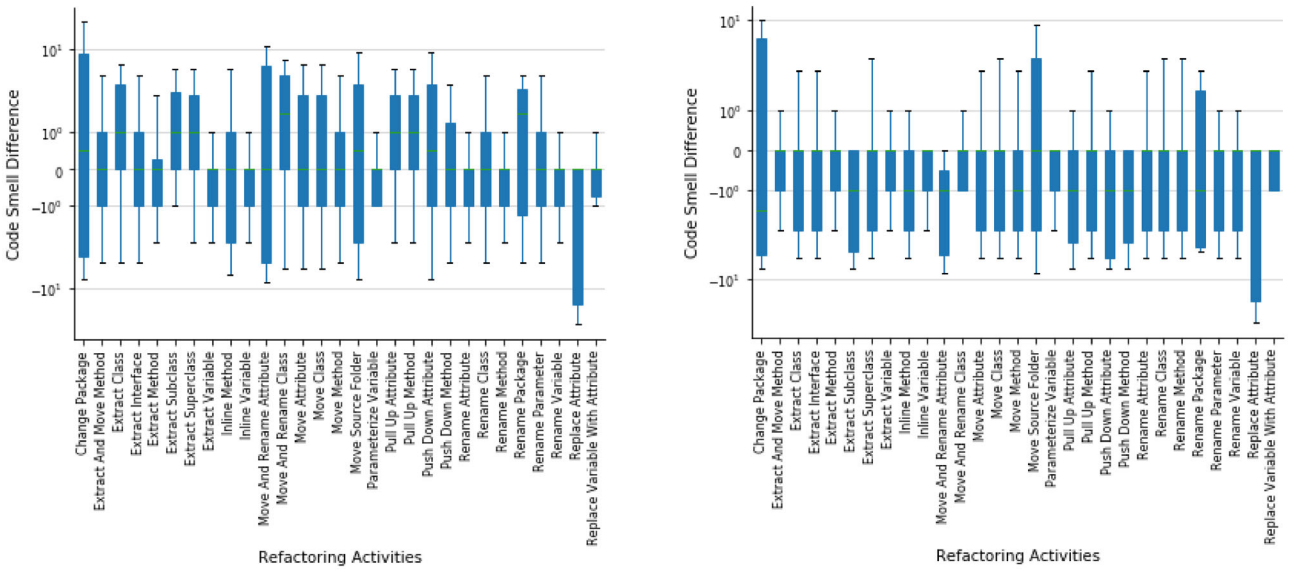
Figure 6 also illustrates the boxplots, depicted in semilog scale, of the distribution of the differences between the density of code smells before and after refactoring. The results are shown for both cases, including files with no prior commit, that is, newly added files, and removing files with no prior commit. The plots in Figure 6A clearly show that the median value of the difference in the number of code smells is equal to or greater than zero for all the refactoring types. Moreover, 10 of the 29 refactoring types including Change Package, Extract Class, Extract Subclass, Extract Superclass, Move and Rename Class, Move Source Folder, Pull Up Attribute, Pull Up Method, Push Down Attribute, and Rename Package tend to introduce code smells. On the other hand, in Figure 6B, in all cases, the median value is equal to or less than zero, that is, the box plots are also tilted down, except for Change Package, Rename Package, and Move Source Folder. Similar to the previous figure, we observe that in Figure 6, when newly added files are removed from the analysis, all refactoring types usually contribute to mitigate the density of code smells.

To address RQ2.1, we calculate the percentage of Jira issues assigned to commits where refactoring activities are performed on smelly files. Please note that during data preprocessing, we only considered the commits which we could match with the issues in Jira. Figure 7 shows the



(A) including the files with no prior commit **(B)** removing the files with no prior commit

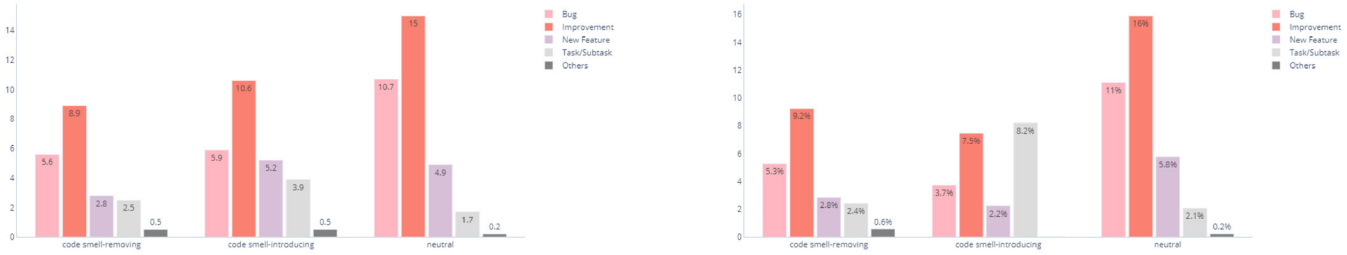
FIGURE 5 Categorization of the refactorings with respect to the change in the density of code smells (aggregated overall projects)



(A) including the files with no prior commit

(B) removing the files with no prior commit

FIGURE 6 Categorization of the refactorings with respect to the density of code smells aggregated overall projects (omitting outliers)



(A) including the files with no prior commit

(B) removing the files with no prior commit

FIGURE 7 Other activities performed with refactoring activities that touch smelly commits

percentage of commits on which other activities were performed together with the refactoring activities for two cases: (i) including the files with no prior commit in Figure 7A and (ii) removing the files with no prior commit in Figure 7B. Moreover, Figure 8 illustrates the overall distribution of other activities that are not limited to refactoring commits.

When 27 projects are overall analyzed, in Figure 7, we observe that the developers mostly implement improvement and bug-fixing activities, respectively, while performing refactoring actions. In the majority of the cases, commits also have other issues with refactoring activities. This might be due to the fact that developers can be combining changes in a single commit. For instance, a developer might perform refactoring activities to improve software maintenance and revise the architecture when improving the code[69] or fixing a bug.[70] Nevertheless, as seen in Figure 7A,B, the other activities have no effect on the change in the density of code smells according to the classification of refactoring activities (as removing, introducing, or neutral). The ratios of improvement, new feature, bug, task, and other activities are the same among the three groups in which refactorings introduce more smells, or remove existing smell, or do not affect the smells. However, as we can see in Figure 7B when the files with no prior commit are not included, task/subtask activities are performed more in commits where refactoring activities introduce new smells. In other words, when working on new tasks, performing refactoring activities causes new code smell introduction into the systems. Also, the ratios of improvement activities are the highest in all three groups of refactorings, whereas bug-fixing activities come the second.

On the other hand, when examining the overall distribution of "other activities" that are not limited to refactoring commits, Figure 8 shows that developers mostly (56.6%) implement bug-fixing activities. In other words, although developers generally tag commits as bug fixing, when they do refactoring activities, they also perform the improvements the most.

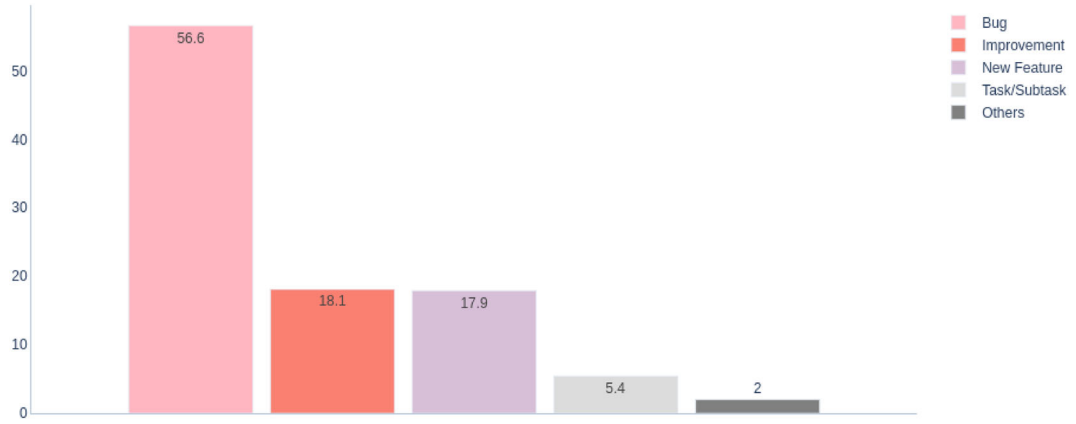


FIGURE 8 Distribution of issue types on all Jira issues

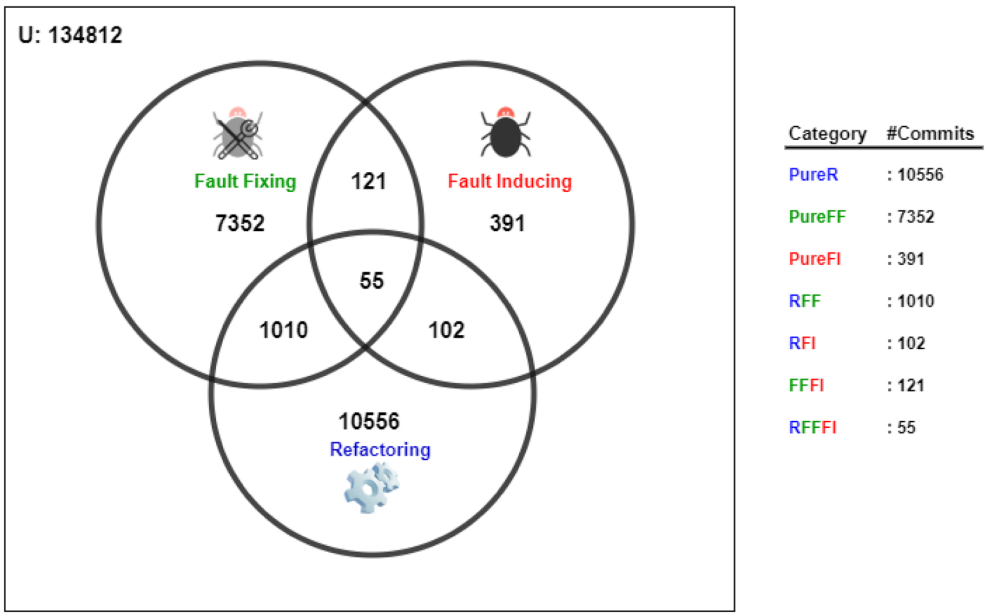


FIGURE 9 Representation of refactoring, fault-inducing, and fault-fixing activities (RQ3)

5.3 | How often do refactoring activities relate to fault-inducing/fault-fixing activities? (RQ3)

Figure 9 shows the number of commits belonging to seven categories defined to answer RQ3. Refactoring activities are usually desired across the software systems as the refactored code components are easier to understand, evolve, and maintain. On the other hand, fault-inducing changes in the software system are avoided as they may cause to produce an incorrect or unexpected result, and fixing them may cause maintenance costs for the systems. Therefore, the number of commits in which refactoring is performed is expected to be higher than the number of commits involving fault-inducing changes. Accordingly, Figure 9 shows that refactoring activities are performed in approximately 23% (ratio of $RFI + RFFFI = 157$ to $PureFI + FFFI + RFI + RFFFI = 669$) of the commits that induced at least one fault into the system. When we compute the ratio over the refactoring commits, only about 1.3% (ratio of $RFI + RFIFF = 157$ to $PureR + RFF + RFI + RFFFI = 11,723$) of the refactoring commits are also labeled as fault-inducing commits. Thus, we cannot say that refactoring activities are prone to induce faults into the system. On the other hand, if we look at the commits with no refactoring activity and the ratio of fault-inducing commits over those commits, we can see that 0.4% (ratio of $PureFI + FFFI = 512$ to $U - 11,723 = 123,089$) of the commits in which no refactoring activity is performed are fault inducing. In other words, the commits in which at least one refactoring is performed are approximately three times more related to fault-inducing changes than the commits in which no refactoring is performed. Although refactoring activities aim to reduce the cost of software development and maintenance, those activities might increase when compared with the commits on which no refactoring performed.

Refactoring, the principal goal of which is to improve the maintenance and evolution of the software systems by removing code smells, may also have merits on fault-fixing changes. While it is much difficult to inspect complex, poorly structured code during fault fixing, it is easier to identify and fix faults during the inspection. Thus, to effectively perform bug-fixing activities,[50] sometimes developers may need to refactor the code before fixing a fault in a poorly written and complex code. To observe this, we also analyzed the relationship between commits in which there is at least one fault-fixing change and commits in which refactoring activities occur. There are 1010 bug-fixing changes out of 11,763 (10%) refactoring commits. On the contrary, in 1010 out of 8538 (12%) fault-fixing commits, developers also performed refactoring activities. Prior studies indicate that refactoring changes should precede fault-fixing changes, that is, prior to a fault-fixing activity, necessary refactorings should be performed.[50,71] Our findings also show that these two actions are performed in the same commits in only 10% to 12% of the cases. In the rest of the cases, refactorings are performed separately from the fixing changes.

6 | THREATS TO VALIDITY

In this section, we discuss possible threats to the validity of our study including threats to external, construct, internal, and conclusion validity.

6.1 | External validity

Threats to external validity relate to the applicability of the findings in other environments such as other software projects and other programming languages. This study is limited to open-source Java projects belonging to the Apache ecosystem. Hence, we cannot prove the generalization of our study results to industrial projects or to projects implemented in other languages or other ecosystems. Nevertheless, the public dataset used in this study consist of 57,528 refactoring actions in thousands of commits on 33 projects with a diversity of age, size, and domain. Thus, we can safely argue that our research questions have so far been addressed on the largest and most diverse dataset in this field. Furthermore, to mitigate threats related to reliability, the study protocol is extensively described in Section 4 which explicitly reports the analysis steps and methods for each research question. Although the dataset was extensively created including many projects with various tool support, it still has some data inconsistencies or lack accurate links between certain software artifacts. For example, we noticed that Ptidej tool periodically performs code smell analysis on all files in a project, but there is no link between these files and the commits on which code smell(s) are matched. For this reason, the files on which code smell(s) are detected in a commit may not match the changed files in a commit. We assume that Ptidej might have been executed on a periodical fashion for all files, not just on the changed files. This requires certain preprocessing on the tool results to identify only the changed files and newly emerged code smells during those commits. As we use a publicly available dataset, provide the additional data package that we generated, and report our data processing methods in detail, other studies can also replicate or refute our findings on the relationships between refactoring activities and code debt indicators.

6.2 | Construct validity

Threats in this category regard to the relationship between the theory and the observation, the constructs defined in research questions for refactoring and code debt indicators. The dataset used in this study contains refactoring actions collected by RMiner tool, and hence, refactoring types that the tool does not identify are excluded. Nevertheless, it is worth noting that RMiner is able to support the majority of the most popular refactoring types[25] applied by developers. Moreover, we are aware that calculating precision and recall in a reliable manner is essential to correct and complete detection of refactorings. In this context, RMiner tool shows about 98% and 87% of precision and recall, respectively. However, it is worth noting that there is no golden set reporting the actual refactoring operations over all the history of the considered projects. Also, RMiner that is one of the most widely used tools for detecting refactoring activities is a state-of-the-art tool.[53,72] Furthermore, the code debt indicators are defined as code smells and faults, as the prior research and literature reviews on TD identify those two as the most commonly observed indicators. Code smells extracted with SonarQube are excluded in our study as its code smell calculation is not clearly defined and validated against the other tools in the literature. Also the amount of code smells identified by every SonarQube execution is quite higher than those by every Ptidej run, which might indicate high false positive rates. So we used the code smells identified by Ptidej in the public dataset, and hence, the study context is limited to the code smell rules and types identified by this tool only as discussed in Section 4.2.2. There are many code smells, most of which are described by Fowler,[40] and new ones have also been proposed in the literature. Also, different smell detection tools have been proposed, each of which is characterized by unique properties and rules[73,74] such as JDeodorant,[75] PMD,[76] and iPlasma.[77] The detection tools utilize various detection techniques that often rely on calculating standard object-oriented metrics or a set of combined metrics and determining threshold values. Setting a threshold for each metric value would require considering the software system size and the expertise of software engineers and developers.[15,74] The selected threshold values also have a potential impact on the number and type of code smells, that is, it may cause

detection of redundant code smells or missing many code smells. Therefore, the choice of detection tool definitely affects our experiments when identifying code smells. Note that Ptidej has a well-structured procedure through rule cards and imports related rule cards when each time detection of a specific code smell is requested. Also, the tool can detect several code smell types, and the rules it uses have extensive validation. A similar limitation exists regarding the software faults identified in the public dataset, but the authors of the Technical Debt Dataset utilized the state-of-the-art tools like SZZ and issue repositories to identify faults and link them to commits for fixing and inducing changes. The dataset gives all the details and can be analyzed to confirm the statistics provided in our study. We need to clarify that we did not use any of the SZZ, Ptidej, RMiner, and SonarQube tools as the dataset provides the extracted data through them. Therefore, this study's findings are limited in the context of this dataset, and any threat at the data collection stage would, unfortunately, impact our findings. Therefore, analyzing and validating the results with other datasets and other tools (e.g., JDeodorant and Designate) are among our future research directions.

6.3 | Internal validity

Threats in internal validity that might have influenced our observations are mainly related to the fact that constructs are measured in different granularities or that different items are compared with each other. For example, in the study presented by Bavota et al,[28] while the refactoring actions were analyzed at the release level, the faults were examined at the commit level. However, such analysis could affect the observations as it is not possible to know if the refactoring, detected between two releases, is implemented on the commits that induced at least one fault. We conduct commit-level refactoring analysis to avoid that problem. Furthermore, as we mentioned in RQ2, Cedrim et al[31] identified the consecutive commits according to the order of they were merged into the main branch while doing the code smell analysis. When analyzing the relationships between refactoring activities and code smells, we made sure that we examined the consecutive commits where the same files affected by refactoring were changed. Therefore, our study design ensures that the refactorings and code smells are analyzed on the same files. Moreover, in a software project, we examined other factors such as new feature, improvement, or task development that can affect the relationship between refactoring and code smells and faults. It is also possible that the refactoring activities in a file are affected by the decisions and experience of the developers. But it is beyond the scope of this post to investigate the causality between developers and the implementation of refactoring.

6.4 | Conclusion validity

Threats to conclusion validity relate the relationship between treatment and outcome. Analysis for each research question on the same commits is quite significant for us because we need to form a different subset of commits in each question for linking refactorings to projects, refactorings to smelly commits, and to fault-inducing and fault-fixing commits. Thus, we applied a different preprocessing and analysis method to address each RQ. We also applied statistical tests that support our conclusions, such as Scott–Knott to evaluate the differences between various refactoring types or projects. Moreover, we present appropriate visualizations such as pie charts, box plots, and bubble plot to show our findings in different perspectives. One can think that our RQ2 findings do not consider code smells moving from one file to another. In other words, when a refactoring activity creates a new file and moves code blocks to the file, existing code smells may be moved from one file to another. Then, we might misinterpret that the refactoring activity actually reduces the code smells, although code smells are simply moved from one file to another. Such cases might influence our results because we do not have the code blocks of every change in every file during a commit in the dataset, and hence, we cannot prove whether a code smell is really removed with a refactoring activity or it is simply moved to a new file.

Nevertheless, we examined newly added files for refactoring activities as sometimes our analysis can be biased where newly added files can bring existing code smells to new locations. According to our analysis, the number of newly added files is 69,381, and refactoring activities are performed in 11,099 of them. The commonly performed refactoring activities such as *Move Class*, *Extract Method*, and *Move Method* are also commonly detected on newly added files. Moreover, code smell(s) was detected in 1095 of the newly added files that were refactored. In more detail, when we compare all newly added files with new smelly files, new files can be more smelly for only *Extract Superclass*. It occurred in 6.2% of newly added files while occurred in 8.13% of new smelly and refactored files. Therefore, more detailed code analysis of newly added files can be an interesting and important topic as future research work.

7 | FINAL REMARKS AND FUTURE WORK

The TD metaphor is widely used as it explains to both technical and nontechnical stakeholders that managing the long-term effects of short-term remedies is critical. The major indicators of TD are represented by code smells and fault hazards due to poor design and implementation choices. On the other hand, refactoring, which aims to improve the internal structure of the software systems preserving their behavior and is the primary

factor of the introducing of the TD metaphor, is the most common strategy to the repayment of the TD by removing the code smells and preventing the occurrence of faults.

This paper reports a large exploratory study conducted on publicly available TD dataset which includes 57,528 refactoring activities, 37,553 code smells, 27,340 faults, and 134,812 commits of 33 Java open-source projects. Our study aims at understanding the distribution of 29 refactoring types among the different projects and their relation with code debt indicators (code smells and faults) on considered projects. The results of our study are highlighted as follows: (i) Although there are 29 refactoring types in the dataset, we observe that some refactoring types are more frequently applied by developers such as *Move Class*, *Extract Method*, *Rename Method/Variable*, *Pull Up Method* on both project basis, and over all projects. These refactoring types aim to remove code smells that are widely studied and have easy-to-use tool supports. (ii) Although refactoring is mostly inconclusive/neutral, unlike previous studies, it has been observed that refactoring activities are usually not harmful. Furthermore, this finding does not differ depending on the selected refactoring types. Therefore, we observe that the factors affecting the conclusions are data processing, proper matching between refactoring activities and code smells, and the inclusion of the files with a change history. We can conclude that the results over different studies conflict to each other due to those factors. Thus, indeed, an exact matching should be applied at a coarser granularity level when examining the relationship between refactoring and code smells. (iii) Various activities are performed in commits where refactoring actions are applied, and some of them might be associated with code smells. For instance, when refactoring is applied with a new task development, code smells are more likely to be introduced. (iv) When we analyzed the relationship between refactoring and faults, another code debt indicator, we discover that the commits in which at least one refactoring is performed are approximately three times more fault inducing than the commits in which no refactoring is performed.

As future work, we can examine the relationship between refactoring and faults at file level considering the evolution of files over change history, and support our results with statistical tests. Current analysis were conducted at commit level for the RQ3 as the refactorings and faults were matched with the commits in the Technical Debt Dataset. If we could like at file level to see the microgranular refactoring actions in files and their link to the locations where refactorings were also performed in files, the analysis results might be more interesting. Furthermore, we plan to analyze the components on which refactoring actions are performed over time. This way we may observe whether code smells in files where refactorings are classified as neutral or introducing later turn into faults or vice versa, that is, whether the elimination of code smells in files where refactorings are classified as removing prevents the occurrence of faults in future. This kind of analysis requires observation of file changes individually over change history and would give better insights on the relationship between TD management strategies and debt indicators.

DATA AVAILABILITY STATEMENT

The data that support the findings of this study are openly available in clowee (GitHub)[33] at https://github.com/clowee/The-Technical-Debt-Dataset and in Google Drive at https://drive.google.com/drive/folders/1It13L2ElPmgivcB0tFnhbDahAQPvF1O9.[55]

ORCID

Rusen Halepmollasi https://orcid.org/0000-0002-9941-2712

ENDNOTES

* Apache: https://apache.org.

† SonarQube: https://www.sonarqube.org.

‡ Open SZZ: https://github.com/clowee/OpenSZZ.

REFERENCES

1. Guo Y, Seaman C, Gomes R, Cavalcanti A, Tonin G, Da Silva FQB, Santos ALM, Siebra C. Tracking technical debt an exploratory case study. In: 2011 27th IEEE International Conference on Software Maintenance (ICSM) IEEE; 2011:528-531.
2. Cunningham W. The wycash portfolio management system. *ACM SIGPLAN OOPS Messenger*. 1993;4(2):29-30.
3. Tom E, Aurum A, Vidgen R. An exploration of technical debt. *J Syst Softw*. 2013;86(6):1498-1516.
4. McConnell S. *Managing Technical Debt*: Construx Software Builders, Inc; 2008;1-14.
5. Kruchten P, Nord RL, Ozkaya I. Technical debt: from metaphor to theory and practice. *IEEE Softw*. 2012;29(6):18-21.
6. Alves NSR, Mendes TS, de Mendonça MG, Spínola RO, Shull F, Seaman C. Identification and management of technical debt: a systematic mapping study. *Inf Softw Technol*. 2016;70:100-121.
7. Martini A, Bosch J, Chaudron M. Investigating architectural technical debt accumulation and refactoring over time: a multiple-case study. *Inf Softw Technol*. 2015;67:237-253.
8. Fowler M. Technical debt quadrant. 14–0; 2009.
9. Lenarduzzi V, Besker T, Taibi D, Martini A, Fontana FA. A systematic literature review on technical debt prioritization: strategies, processes, factors, and tools. *J Syst Softw*. 2020;171:110827.
10. Brown N, Cai Y, Guo Y, Kazman R, Kim M, Kruchten P, Lim E, MacCormack A, Nord R, Ozkaya I. Managing technical debt in software-reliant systems. In: The FSE/SDP Workshop on Future of Software Engineering Research ACM; 2010:47-52.
11. Li Z, Avgeriou P, Liang P. A systematic mapping study on technical debt and its management. *J Syst Softw*. 2015;101:193-220.

12. Behutiye WN, Rodríguez P, Oivo M, Tosun A. Analyzing the concept of technical debt in the context of agile software development: a systematic literature review. *Inf Softw Technol*. 2017;82:139-158.
13. Seaman C, Guo Y, Zazworka N, Shull F, Izurieta C, Cai Y, Vetrò A. Using technical debt data in decision making: potential decision approaches. In: 2012 Third International Workshop on Managing Technical Debt (MTD) IEEE; 2012:45-48.
14. Al Dallal J. Identifying refactoring opportunities in object-oriented code: a systematic literature review. *Inf Softw Technol*. 2015;58:231-249.
15. Lacerda G, Petrillo F, Pimenta M, Guéhéneuc YG. Code smells and refactoring: a tertiary systematic review of challenges and observations. *J Syst Softw*. 2020;167:110610.
16. Olbrich S, Cruzes DS, Basili V, Zazworka N. The evolution and impact of code smells: a case study of two open source systems. In: 2009 3rd International Symposium on Empirical Software Engineering and Measurement IEEE; 2009:390-400.
17. Chatzigeorgiou A, Manakos A. Investigating the evolution of bad smells in object-oriented code. In: 2010 Seventh International Conference on the Quality of Information and Communications Technology IEEE; 2010:106-115.
18. Khomh F, Di Penta M, Guéhéneuc Y-G, Antoniol G. An exploratory study of the impact of antipatterns on class change- and fault-proneness. *Empir Softw Eng*. 2012;17(3):243-275.
19. Palomba F, Bavota G, Di Penta M, Fasano F, Oliveto R, De Lucia A. On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation. *Empir Softw Eng*. 2018;23(3):1188-1221.
20. Tufano M, Palomba F, Bavota G, et al. When and why your code starts to smell bad (and whether the smells go away). *IEEE Trans Softw Eng*. 2017;43(11):1063-1088.
21. Demeyer S, Ducasse S, Nierstrasz O. Finding refactorings via change metrics. *ACM SIGPLAN Notices*. 2000;35(10):166-177.
22. Xing Z, Stroulia E. UMLDiff: an algorithm for object-oriented design differencing. In: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering; 2005:54-65.
23. Xing Z, Stroulia E. The JDEvAn tool suite in support of object-oriented evolutionary development. In: Companion of the 30th International Conference on Software Engineering; 2008:951-952.
24. Xing Z, Stroulia E. Refactoring practice: how it is and how it should be supported—an eclipse case study. In: 2006 22nd IEEE International Conference on Software Maintenance IEEE; 2006:458-468.
25. Murphy-Hill E, Black AP. Refactoring tools: fitness for purpose. *IEEE Softw*. 2008;25(5):38-44.
26. Murphy-Hill E, Parnin C, Black AP. How we refactor, and how we know it. *IEEE Trans Softw Eng*. 2011;38(1):5-18.
27. Weißgerber P, Diehl S. Are refactorings less error-prone than other changes? In: Proceedings of the 2006 International Workshop on Mining Software Repositories; 2006:112-118.
28. Bavota G, De Carluccio B, De Lucia A, Di Penta M, Oliveto R, Strollo O. When does a refactoring induce bugs? An empirical study. In: 2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation IEEE; 2012:104-113.
29. Gatrell M, Counsell S. The effect of refactoring on change and fault-proneness in commercial C# software. *Sci Comput Progr*. 2015;102:44-56.
30. Bavota G, De Lucia A, Di Penta M, Oliveto R, Palomba F. An experimental investigation on the innate relationship between quality and refactoring. *J Syst Softw*. 2015;107:1-14.
31. Cedrim D, Garcia A, Mongiovi M, et al. 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; 2017:465-475.
32. Prete K, Rachatasumrit N, Sudan N, Kim M. Template-based reconstruction of complex refactorings. In: 2010 IEEE International Conference on Software Maintenance IEEE; 2010:1-10.
33. Lenarduzzi V, Saarimäki N, Taibi D. The technical debt dataset. In: Proceedings of the Fifteenth International Conference on Predictive Models and Data Analytics in Software Engineering; 2019:2-11.
34. Codabux Z, Dutchyn C. Profiling developers through the lens of technical debt. In: Proceedings of the 14th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM); 2020:1-6.
35. Lenarduzzi V, Saarimäki N, Taibi D. Some SonarQube issues have a significant but small effect on faults and changes. A large-scale empirical study. *J Syst Softw*. 2020;170:110750.
36. Lenarduzzi V, Lomio F, Huttunen H, Taibi D. Are SonarQube rules inducing bugs? In: 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER) IEEE; 2020:501-511.
37. Tsoukalas D, Mathioudaki M, Siavvas M, Kehagias D, Chatzigeorgiou A. A clustering approach towards cross-project technical debt forecasting. *SN Comput Sci*. 2021;2(1):1-30.
38. Shaw M. Writing good software engineering research papers. In: 25th International Conference on Software Engineering, 2003. Proceedings IEEE; 2003:726-736.
39. Fowler M, Beck K, Brant J, Opdyke W, Roberts D. *Refactoring: Improving the Design of Existing Code*, Addison Wesley Object Technology Series: Addison-Wesley; 1999.
40. Fowler M. *Refactoring: Improving the Design of Existing Code*: Addison-Wesley Professional; 2018.
41. Lehman MM. Laws of software evolution revisited. In: European Workshop on Software Process Technology Springer; 1996:108-124.
42. Peters R, Zaidman A. Evaluating the lifespan of code smells using software repository mining. In: 2012 16th European Conference on Software Maintenance and Reengineering IEEE; 2012:411-416.
43. Chatzigeorgiou A, Manakos A. Investigating the evolution of code smells in object-oriented systems. *Innov Syst Softw Eng*. 2014;10(1):3-18.
44. Li W, Shatnawi R. An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution. *J Syst Softw*. 2007;80(7):1120-1128.
45. Khomh F, Di Penta M, Gueheneuc Y-G. An exploratory study of the impact of code smells on software change-proneness. In: 2009 16th Working Conference on Reverse Engineering IEEE; 2009:75-84.
46. D Ambros M, Bacchelli A, Lanza M. On the impact of design flaws on software defects. In: 2010 10th International Conference on Quality Software IEEE; 2010:23-31.
47. Opdyke WF. Refactoring: an aid in designing application frameworks and evolving object-oriented systems. In: Proceedings of SOOPPA'90: Symposium on Object-Oriented Programming Emphasizing Practical Applications; 1990.

48. Gorg C, Weißgerber P. Detecting and visualizing refactorings from software archives. In: 13th International Workshop on Program Comprehension (IWPC'05) IEEE; 2005:205-214.
49. Easterbrook S, Singer J, Storey M-A, Damian D. Selecting empirical methods for software engineering research. *Guide to Advanced Empirical Software Engineering*: Springer; 2008:285-311.
50. Palomba F, Zaidman A, Oliveto R, De Lucia A. An exploratory study on the relationship between changes and refactoring. In: 2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC) IEEE; 2017:176-185.
51. Spadini D, Aniche M, Bacchelli A. Pydriller: Python framework for mining software repositories. In: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering; 2018:908-911.
52. Guéhéneuc Y-G. Ptidej: A flexible reverse engineering tool suite. In: 2007 IEEE International Conference on Software Maintenance IEEE; 2007: 529-530.
53. Tsantalis N, Mansouri M, Eshkevari L, Mazinanian D, Dig D. Accurate and efficient refactoring detection in commit history. In: 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE) IEEE; 2018:483-494.
54. Lenarduzzi V, Palomba F, Taibi D & Tamburri DA OpenSZZ: a free, open-source, web-accessible implementation of the SZZ algorithm. In: Proceedings of the 28th International Conference on Program Comprehension ACM; 2020:446-450.
55. Halepmollasi R, Tosun A. Jsep_refactoring submission—Google Drive. (Accessed on 01/23/2021); 2021.
56. Biegel B, Soetens QD, Hornig W, Diehl S, Demeyer S. Comparison of similarity metrics for refactoring detection. In: Proceedings of the 8th Working Conference on Mining Software Repositories; 2011:53-62.
57. Scott AJ, Knott M. A cluster analysis method for grouping means in the analysis of variance. *Biometrics*. 1974;30:507-512.
58. Agrawal R, Imieliński T, Swami A. Mining association rules between sets of items in large databases. In: Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data; 1993:207-216.
59. Agrawal R, Srikant R, et al. Fast algorithms for mining association rules. In: Proceedings of 20th International Conference on Very Large Data Bases (VLDB), Vol. 1215; 1994:487-499.
60. Moha N, Guéhéneuc Y-G, Duchien L, Le Meur A-F. DECOR: a method for the specification and detection of code and design smells. *IEEE Trans Softw Eng*. 2009;36(1):20-36.
61. Śliwerski J, Zimmermann T, Zeller A. When do changes induce fixes? *ACM Sigsoft Softw Eng Notes*. 2005;30(4):1-5.
62. Sahal E, Tosun A. Identifying bug-inducing changes for code additions. In: Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement; 2018:1-2.
63. Vassallo C, Grano G, Palomba F, Gall HC, Bacchelli A. A large-scale empirical exploration on refactoring activities in open source software projects. *Sci Comput Progr*. 2019;180:1-15.
64. Du Bois B, Mens T. Describing the impact of refactoring on internal program quality. In: International Workshop on Evolution of Large-Scale Industrial Software Applications; 2003:37-48.
65. Silva D, Tsantalis N, Valente MT. Why we refactor? Confessions of GitHub contributors. In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering; 2016:858-870.
66. Kataoka Y, Imai T, Andou H, Fukaya T. A quantitative evaluation of maintainability enhancement by refactoring. In: 2002 Proceedings International Conference on Software Maintenance IEEE; 2002:576-585.
67. Counsell S, Hassoun Y, Loizou G, Najjar R. Common refactorings, a dependency graph and some code smells: an empirical study of Java OSS. In: Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering; 2006:288-296.
68. Al Dallal J. Predicting move method refactoring opportunities in object-oriented code. *Inf Softw Technol*. 2017;92:105-120.
69. Zabardast E, Gonzalez-Huerta J, Šmite D. Refactoring, bug fixing, and new development effect on technical debt: an industrial case study. In: 2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA) IEEE; 2020:376-384.
70. Kim M, Zimmermann T, Nagappan N. A field study of refactoring challenges and benefits. In: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering; 2012:1-11.
71. Palomba F, Zaidman A. Notice of retraction: does refactoring of test smells induce fixing flaky tests? In: 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME) IEEE; 2017:1-12.
72. Tsantalis N, Ketkar A, Dig D. RefactoringMiner 2.0. *IEEE Trans Softw Eng*. 2020;2020:1-21.
73. Fontana FA, Mariani E, Mornioli A, Sormani R, Tonello A. An experience report on using code smells detection tools. In: 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops IEEE; 2011:450-457.
74. Paiva T, Damasceno A, Figueiredo E, Sant-Anna C. On the evaluation of code smells and detection tools. *J Softw Eng Res Develop*. 2017;5(1):1-28.
75. Tsantalis N, Chaikalis T, Chatzigeorgiou A. JDeodorant: identification and removal of type-checking bad smells. In: 2008 12th European Conference on Software Maintenance and Reengineering IEEE; 2008:329-331.
76. PMD. http://pmd.sourceforge.net/
77. Marinescu R. Detection strategies: metrics-based rules for detecting design flaws. In: 20th IEEE International Conference on Software Maintenance IEEE; 2004:350-359.

How to cite this article: Halepmollasi R, Tosun A. Exploring the relationship between refactoring and code debt indicators. *J Softw Evol Proc*. 2024;36(1):e2447. doi:10.1002/smr.2447