Refactoring, Bug Fixing, and New Development Effect on Technical Debt: An Industrial Case Study

Ehsan Zabardast, Javier Gonzalez-Huerta, and Darja Šmite
Software Engineering Research Lab SERL Sweden
Blekinge Institute of Technology
Karlskrona, Sweden
$\{$*ehsan.zabardast, javier.gonzalez.huerta, darja.smite*$\}$*@bth.se*

***Abstract*—Code evolution, whether related to the development of new features, bug fixing, or refactoring, inevitably changes the quality of the code. One particular type of such change is the accumulation of Technical Debt (TD) resulting from sub-optimal design decisions. Traditionally, refactoring is one of the means that has been acknowledged to help to keep TD under control. Developers refactor their code to improve its maintainability and to repay TD (e.g., by removing existing code smells and anti-patterns in the source code). While the accumulation of the TD and the effect of refactoring on TD have been studied before, there is a lack of empirical evidence from industrial projects on how the different types of code changes affect the TD and whether specific refactoring operations are more effective for repaying TD. To fill this gap, we conducted an empirical study on an industrial project and investigated how Refactoring, Bug Fixing, and New Development affect the TD. We have analyzed $2,286$ commits in total to identify which activities reduced, kept the same, or even increased the TD, further delving into specific refactoring operations to assess their impact. Our results suggest that TD in the studied project is mainly introduced in the development of new features (estimated in 72.8 hours). Counterintuitively, from the commits tagged as refactoring, only 22.90% repay TD (estimated to repay 8.30 hours of the TD). Moreover, while some types of refactoring operations (e.g., Extract Method), help repaying TD, other refactoring operations (e.g., Move Class) are highly prone to introduce more TD.**

***Keywords*-Technical Debt, Empirical Study, Industrial Study, Case Study, Refactoring, Bug Fixing, New Development**

I. Introduction

Technical Debt (TD) is a metaphor used to discuss the long-term consequence of sub-optimal design decisions taken when short-term goals are prioritized [1], and motivate the importance of refactoring for primarily nontechnical stakeholders [2]. TD inevitably accumulates and evolves as the software develops, and as it is maintained [3]. TD is infamous for its negative impacts on software maintainability and evolvability [4], [5]; and has, therefore, become an important research topic in modern software engineering.

Technical Debt has been approached by researchers from many different angles. On the one hand, one research direction in the area of technical debt has focused on the "rhetorical discussions" about the use of the metaphor [2], ways for identifying and measuring the debt, causes, and effects of the technical debt. On the other hand, several other research works focus on the understanding of how to deal with the TD, in particular, TD repayment, with refactoring being one of the most common topics [6].

Refactoring is "the process of changing a software system in such a way that does not alter the external behavior of the code yet improves its internal structure" [7]. Certain refactoring operations aim at improving the maintainability of the code, while others are aiming at improving the understandability or having more "clean code" (as defined in [8]), or remediating the TD by removing a TD item [5], [9], [10]. TD items (TDI) are "single elements of TD," something that can be identified in the code [5], and have been introduced to be able to quantify or visualize the TD. In their turn, refactoring operations are presented in refactoring catalogs, such as the ones presented by Fowler [7], [11]. The catalogs provide the motivation behind each refactoring operation and the circumstances in which it should be used.

Yet, refactoring is not the only way to address the TD. As software evolves, developers perform various manipulations to the code that can be categorized into three major types of activities: refactoring, bug fixing, and new feature development, all of which are likely to have some impact on the TD. Related studies show, for example, that developers spend, on average, 25% of the development time on managing the TD [12], while these activities are not always performed systematically but rather sporadically during the development process [12], [13]. Similarly, Palomba et al. [14] and Kim et al. [15] found that refactoring operations are mostly performed when new features are implemented and not as a result of dedicated code maintenance. At the same time, any manipulation of the code, whether related to the development of new features, bug fixing, or refactoring, inevitably changes the quality of the code and often results in the accumulation of TD. However, to the best of our knowledge, the research comparing the evolution of TD linked to the different types of activities (refactoring, bug fixing, and new development) is scarce.

In this article, we investigate the effects that refactoring, bug fixing, and new development have on the accumulation of the TD. In addition, we further delve into specific refactoring operations to understand how these refactoring operations, that are expected to improve the internal quality of the source code [7], [11], impact on the accumulation of repayment of TD. Therefore, we aim at answering the following research questions:

- **RQ1.** *To what extent do activities marked as Refactoring, Bug Fixing, and New Development affect the accumulation of Technical Debt in the project?*
- **RQ2.** *How each specific type of refactoring operations affects the Technical Debt in the project?*

This is done by conducting an empirical study where we analyze a large-scale industrial project commit by commit, to assess the impact that each commit has on the accumulated TD. In this project, developers systematically tagged their commits to identify the activity addressed in the commit. We used two independent tools to i) calculate TD and ii) detect the refactoring operations in each commit. We merge the results to evaluate how TD is affected by the activities. We analyzed $2,286$ commits in total and investigated whether the total TD was reduced, remained the same as before, or increased for each activity, and later by the specific refactoring operation.

The rest of this article is structured as follows. Section II summarizes the related work. Section III, research methodology, describes how the data was collected and analyzed. The results are presented in Section IV, and the implications of the results are discussed in Section V. Section VI discusses the threats to validity. Lastly, Section VII concludes the paper.

II. Related Work

Managing TD is essential, and companies have different approaches on how to address this problem. In particular, tracking TD and how it impacts the development is of particular interest for industry and academia. In [12], the authors investigate the state-of-practice in managing TD and aim to understand how companies track TD, what tools they use, and what is the cost of managing TD. They found that, on average, 25% of development time is spent on managing TD. In [13], the authors investigate the waste of development time with regards to TD management. Their results suggest that developers waste 23% of their time on managing TD (i.e., mainly through refactoring), and developers frequently introduce new TD.

Refactoring the code is one of the strategies to deal with TD, and it has been investigated before [10], [16]–[24]. In [18], the authors investigate the relationship between code quality and refactoring operations. They conclude that there is no clear relationship between the refactoring operations and code quality because the refactoring operations mostly target the code components that quality metrics do not consider as in-need-of-improvement. Palomba et al. [17] investigate the perception of developers on code smells. They summarize their findings in four lessons: not all the code smells are considered as design flaws; the "intensity" of the problem is an indication of it being a code smell or not; the complex or long source code are generally an important sign of code smells; and, the experience of developers is a key when identifying a CS.

The impact of refactoring operations, in general, have mostly been studied on code smells, which is only one type of TDI. Santos et al. in [19] vestigate the impact of code smells on software development. In [25], Fujiwara et al. propose a method to assess the benefits of refactoring instances in maintainability. They use three metrics; namely *refactoring frequency*, *defect density*, and *fix frequency*. They conclude that after a term with higher refactoring frequency, defect introduction decreases. In their paper, Tufano et al. [22] studied "when the code smells are introduced" and "what is the survivability of the code smells." They conducted a study over the change history of 200 open source projects from Apache, Android, and Eclipse ecosystems. They focused on five different code smell types, namely *Blob Class*, *Class Data Should be Private*, *Complex Class*, *Functional Decomposition*, and *Spaghetti Code*. They concluded that "most of the smell instances are introduced when an artifact is created and not as a result of its evolution." They also found that 80% of the code smells remain in the system. Finally, from the remaining 20% of the code smells removed, refactoring operations only remove 9% of the code smells. In a similar study, Yoshida et al. [20] analyzed the refactoring data and code smells detected in Apache Ant, ArgoUML, and Xerces-J. They investigate the effectiveness of refactoring patterns applied to code smells to answer whether the refactoring operation helped to remove the code smells. The results of their investigation concluded that "... refactoring rarely removes the code smell because the corresponding pattern is rarely applied to a code smell." In a similar article, Palomba et al. [14] investigate the relationship between refactoring operations and code changes. Their results indicate that most of the refactoring operations are done to remove the duplicated code or "previously introduced self-admitted technical debt."

The articles that study TD management approaches have an overview of what activities impact TD [6]. Digkas et al. [23] studied fifty-seven open-source projects to investigate how TD accumulates during the maintenance process. They found out that a small proportion of issues (types of TDIs in SonarQube terminology) are responsible for the repayment of the large percentage of TD. They also studied the evolution of TD, considering other activities than refactorings. Silva et al. [21], similar to the previous study, investigated the motivation behind applying specific refactoring operations. They concluded that refactoring operations are mostly performed when new requirements are presented rather than to remove existing code smells.

Our study aims to fill in the gaps that are not covered in previous studies by combining their strengths and different perspectives. While previous studies mostly focus on specific type of activities (primarily refactoring) and their impact on specific types of TDI (code smells), our study compares the impact of different activities (refactoring, bug fixing, and new development) and their impact on TD. In contrast to many studies of open source projects and sole reliance on automatically detected refactoring operations, we have a better certainty over the nature of activities by using the tags that developers systematically introduced in their commits, and thus their intention. Finally, while previous studies use open-source data, we use industrial data to have insights from the state-of-the-practice.

III. Research Methodology

To address the research questions, we designed an empirical study where we analyzed data gathered through archival analysis. We selected a large-scale (approximately 1.5 million LOC) industrial project from a company that chose to stay anonymous. The product provides financial services via mobile phones and the internet (FinTech global product). The software in the project was written in JAVA and has evolved for more than ten years. The project has followed core Agile practices (e.g., continuous integration) with frequent releases. We chose this project based on convenience (availability and access) and because we could extract developers' intention for each specific commit activity through the systematically documented commit tags. The analysis is circumscribed to a period of one year, where the project was under heavy development process. We expected to have more activity on the code during this period, which encompasses more development and refactoring. It is important to highlight that this project was developed with some specific development practices such as "clean-code", test-focused development, high emphasis on refactoring, and having a reliable regression test suite in place.

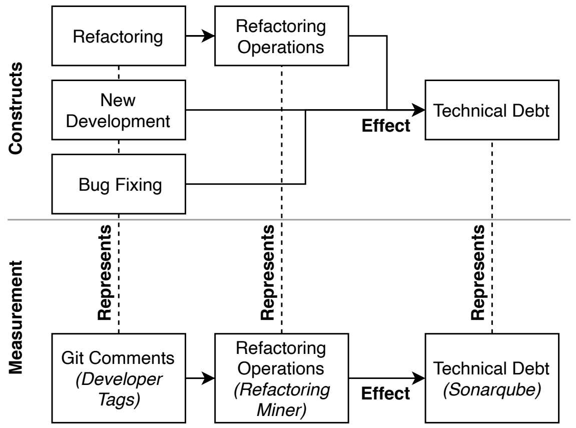

Figure 1. The Study Constructs and Measurements.

In the following, we describe the main constructs and measurements used in our study (summarized in Fig. 1). We analyzed i) the type of commit activity based on the tag provided by developers for each commit; ii) the amount of TD in a given commit; iii) refactoring operations in each commit. We used two tools to collect the data: i) SonarQube[1] for calculating the TD in the project, and ii) RefactoringMiner[2] to detect the refactoring operations (ROs). Both tools have been previously used in similar studies of refactorings [16] and TD [23]. The data was collected by the tools separately in two steps and then combined for the analysis using the git hash identifier for each commit. The details of how the data was collected in each step are described further in Section III-A.

A. Data Collection

1) Detecting Types of Code Change Activities: As described in the case description, the comment field in each commit was systematically tagged by the developers, which helped us determine the intention of that particular commit. We classified the activities on the commits in six categories, namely *Bug Fixing*, *Build*, *Commit Merge*, *New Development*, *Refactoring*, and *Other*.

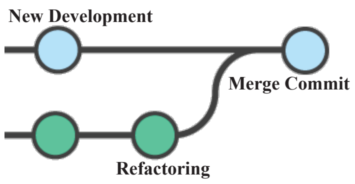

Figure 2. Merge Commit Investigation Example.

We discarded the commits tagged with *Build* and *Other* since we are interested in commits that i) we can extract developers' intent; and ii) are related to the activities in the scope of this paper (i.e., refactoring, bug fixing, and new development). In the case of *Merge Commits*, we looked into the branches before the merge commits to investigate them instead. As an example, in Fig. 2, instead of investigating TD in the *Merge Commit*, we looked back at the commits of the two branches, i.e., *Refactoring* and *New Development*, to analyze the activities and TD in those commits. The details of the collected data are presented in Table I.

2) Detecting Technical Debt: SonarQube is an open-source tool used for code quality inspection for a software project, which analyzes the source code in order to detect bugs, code smells, and security vulnerabilities (what in SonarQube terminology is referred to as issues). Additionally, SonarQube provides the effort in time, which is calculated based on the remediation effort function. Therefore, the technical debt of a project is the summation of the estimated time needed to solve all the issues. We used the default profile in SonarQube for calculating the remediation time.

[1]https://www.sonarqube.org (version 6.7.4)
[2]https://github.com/tsantalis/RefactoringMiner

Table I
DEMOGRAPHICS OF THE ACTIVITIES.

Name	# of Instances	Percentage
Bug Fixing	226	9.88%
Build	199	8.70%
Commit Merge	578	25.28%
New Development	801	35.03%
Refactoring	476	20.82%
Other	6	0.26%
Total	2286	100%

3) Detecting Refactoring and Refactoring Operations: We used RefactoringMiner to extract the detected refactoring operations performed on commits classified as refactoring. RefactoringMiner is a library developed by Tsantalis et al. [26] that detects various types of refactoring operations in the history of a JAVA project. The latest version of RefactoringMiner can detect 40 different types of ROs with the precision of 98% and recall of 87% [26]. We use RefactoringMiner to first detect the ROs in the history of the project in all the investigated commits. Later, we filtered and investigated only the ROs that are tagged as a "refactoring" commit by the developers.

B. Data Analysis

To analyze the effect of Refactoring, Bug Fixing, and New Development, we calculated $\Delta TD_j = \sum e_j - \sum e_{j-1}$ (e: effort in minutes)—i.e., the TD introduced or removed by commit *j*. ΔTD_j is the difference in TD between commit *j*, in which the activity has happened, and the TD of the previous commit TD (i.e., commit *j-1*). The sign of ΔTD_j determines how TD is affected. If the sign is positive, it means that the project accumulates more TD with that particular commit (i.e., TD increases). If the sign is negative, it means that the TD was paid back with that particular commit (i.e., TD decreases). If ΔTD_j is *zero*, it means that TD has not changed with that particular commit.

To analyze how individual Refactoring Operations affect TD, we use the same concept of ΔTD_j. However, we analyze the data with a different granularity level.

We utilize the data collected by the RefactoringMiner tool to extract the ROs that happened on specific files of the commits tagged as refactoring. The developer expressed that the primary purpose of the commit was performing refactoring. We look at the issues (if any) that were introduced or removed by that particular commit in the files in which RefactoringMiner detected ROs. With this data, we can trace whether particular ROs happening in a file (or pairs of files) have an effect on ΔTD_j for those particular files (or pairs of files). Fig. 3 illustrates an example of how ΔTD_j is calculated when an RO removes an issue.

IV. RESULTS

In this section, we present our results from studying the impact of refactoring (also focusing on specific refactoring

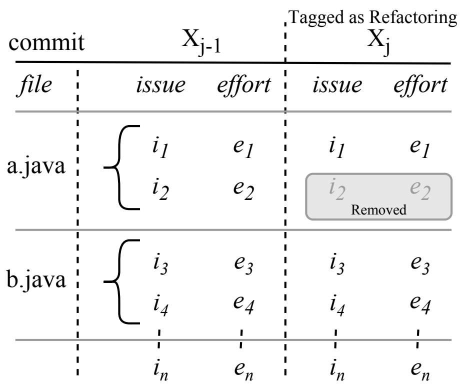

Figure 3. An Example of How ΔTD is Calculated Where an Issue is Removed.

operations), bug fixing, and new development on Technical Debt. Table II summarizes the descriptive statistics related to the three types of activities and their impact on the accumulation and repayment of TD. As we can see, Refactoring and Bug Fixing have on average negative impact on TD (overall mean of -1.04 and -0.2 minutes respectively), meaning that refactoring and bug fixing, at large, helps in repaying TD. The mean for commits tagged as New Development is 5.45 minutes meaning that, overall, it contributes to the accumulation of TD.

Table II
STATISTICS FOR THE COLLECTED DATA ON REFACTORING (R), BUG FIXING (BF), AND NEW DEVELOPMENT (ND).

	N	Mean	SD	Effect on TD	Cases Adding TD	Cases Repaying TD
R	476	$-1.04'$	$29.30'$	$-8.3h$	106	109
BF	226	$-0.2'$	$20.89'$	$-0.77h$	40	44
ND	801	$5.45'$	$38.87'$	$72.8h$	250	158

Fig. 4 illustrates the accumulation of TD in hours in the project during the period under analysis (i.e., prior to the removal of commits tagged as *Build*, *Commit Merge*, and *Other*). Commit 1 in this picture represents the initial commit on the time-span of our analysis we investigated. The total amount of TD increased by $67.72h$ during the analyzed time-span, although, as we can observe in Fig. 4, it fluctuates significantly during that period. Our results further suggest that, in most cases, TD is introduced during the *New Development*, while *Refactoring* and *Bug Fixing* were found to, on average, remove TD. There are four major changes in TD highlighted in Fig. 4 with red vertical solid lines for major increases and green vertical dashed lines for major decreases:

1: A sharp increase of TD tagged as new development.

2: A significant decrease of TD tagged as refactoring (tagged as *"Removed legacy value"*, *"Clean up"*, and *"Removed dead code"*).

3: A sharp increase of TD tagged as new development.

4: A significant decrease of TD tagged as refactoring and bug fixing (tagged as *"Fixing failing test cases"*).

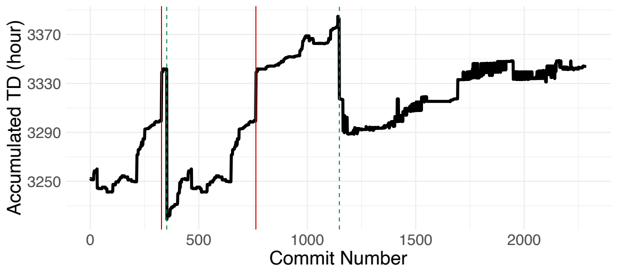

Figure 4. The Evolution of Accumulated Technical Debt (hours) in the Project.

Fig. 5 illustrates the impact of each activity on the accumulation of TD in the project. The commits illustrated in this figure are sequential but not consecutive because, as described in Section VII, we have removed the commits tagged as *Build*, *Commit Merge*, and *Other*. The red bars in the figure show the total amount of the TD introduced by a given commit, whereas the green bars show the total amount of the TD removed in a given commit.

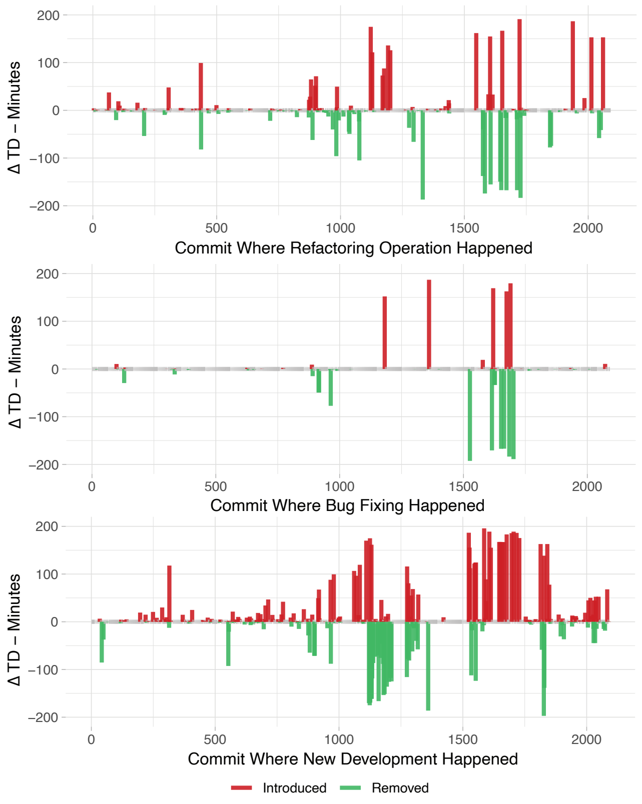

Figure 5. The Illustration of the Impact of Activities on ΔTD During the Evolution of the Code.

As it could have been expected, our results indicate that *New Development* is the primary source of accumulation of TD. The 801 commits tagged as *New Development* contributed to the accumulation of TD by $72.8h$ hours in total. As shown in Fig. 6, 31.21% (250 cases) of the commits marked as *New Development* contributes to the accumulation of TD, while 19.73% (158 cases) contributed to its repayment. This can be owing to the fact that the commits tagged as the development of new features might also include refactorings to introduce design or architectural changes. Previous research suggests that refactoring most likely occurs during new development or bug fixing (e.g., [14], [15]).

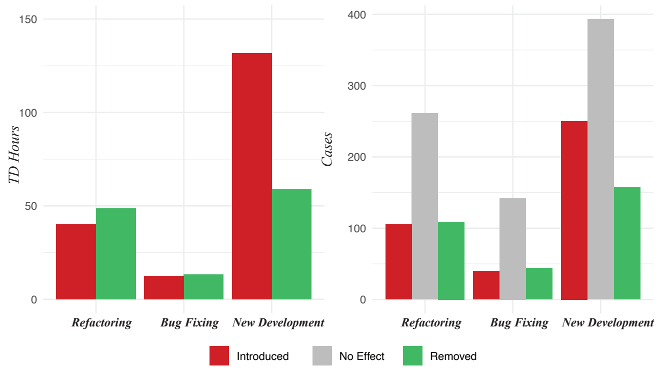

Figure 6. The Total Number of Cases and TD, in hours, each Activity Introduces or Removes from the Project.

In the analyzed period, the commits tagged as *Bug Fixing* contributed to the repayment of TD. The 226 commits tagged as bug fixing contributed to the repayment of TD by $0.77h$ (i.e., 46 minutes) in total. While 16.54% (44 cases) of the commits repaid TD, 15.03% (40 cases) contributed to the accumulation of TD, and 62.83% had no impact on the TD in the project.

Refactoring is the main activity that contributes to the repayment of TD. The 476 commits tagged as refactoring contributed to the repayment of TD by 8.30 hours in total. While 22.90% (109 cases) of the commits repaid TD, 22.27% (106 cases) contributed to the accumulation the TD, and 54.83% had no impact on the TD in the project.

To further detect the individual Refactoring Operations that had happened in the commits tagged by the developers as "Refactoring," we used RefactoringMiner tool. Our analysis of the specific ROs suggests that out of 40 different types of ROs that the tool detects, there are only 22 types present within the analyzed commits. The impact of these ROs on the TD is illustrated in Fig. 7.

Out of 22 ROs detected, 5 ROs, namely *Replace Attribute*, *Pull Up Method*, *Parametrize Variable*, *Move Attribute*, and *Extract Superclass* helped to remove the TD in all the cases. Three ROs, namely *Push Down Method*, *Pull Up Attribute*, and *Extract Subclass* were found to have no effect on the TD, while *Extract and Move Method* were found to accumulate

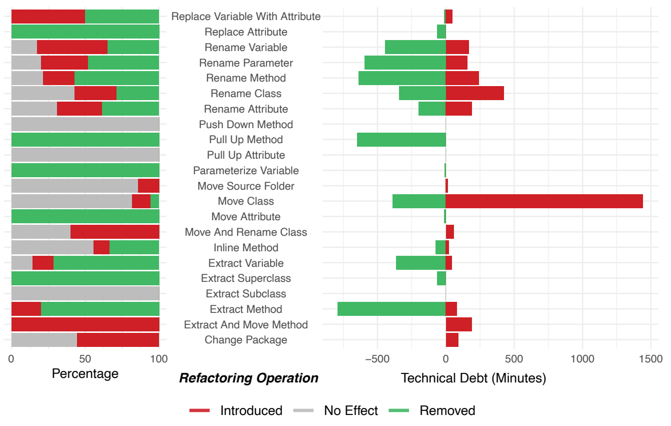

Figure 7. Refactoring Operations Types Detected within the Analyzed Commits in the Project.

TD in all cases. *Replace Attribute*, *Rename Variable*, *Rename Parameter*, *Rename Method*, *Rename Class*, *Rename Attribute*, *Pull Up Method*, *Extract Variable*, and *Extract Method* can be said to be the most effective in removing TD. At the same time, *Rename Class*, *Move Class*, *Move and Rename Class*, *Extract and Move Method*, and *Change Package* contribute to the accumulation of TD, although in some cases can also help repaying TD. Fig. 7 (on the left side) together with Table III illustrate and summarize the ratio of observations for the RO types.

V. Discussion

In this section, we discuss our results with regards to our research questions, followed by the implications of the major findings for future research and practice.

A. RQ1: The effect of Refactoring, Bug Fixing, and New Development on TD

As already highlighted in Section IV, Refactoring, Bug Fixing, and New Development affect the accumulation of TD in very different ways. We have observed that commits tagged as *Refactoring* are, in general, contributing to the repayment of TD, but also that some commits tagged as *New Development* are contributing to its repayment. However, in the majority of the cases commits have no global impact on the accumulated TD (i.e., grey bars in Fig. 6). This might be owing to the fact that, although the product under study has strict rules to tag commits, developers can be amalgamating changes in a single commit. For example a commit tagged as *New Development* might also contain many refactoring operations to prepare the architecture for the new code added, and even some bug fixes, as suggested in previous research in the area [15]. Refactorings can also remove TD on a given code entity but at the same time introduce the same amount of TD elsewhere, and the total might have 0 balance (as if there has been no effect). In addition, sometimes refactoring operations might contribute to improving the quality of the code (by for example reducing the size of a class without achieving the required length to remove the Large Class code smell).

When considering the whole project, it might seem that refactoring might not have a big impact on TD, but as illustrated in Fig. 4, without the major refactoring events that usually happen after new development, the accumulated TD will grow very quickly and out of control.

Commits tagged as New Development contribute to the accumulation of TD in 31.21% of the cases. However, New Development also helps repaying TD in 19.73% of the cases. Refactoring, on the other hand, is expected to have substantial effect on the TD accumulated in the project. However Refactoring is only responsible for the removal of TD in 22.90% of the cases; in 54.83% cases, do not change the amount of TD; and, counterintuitively, it introduces additional TD in 22.27% of the cases. While Refactoring is more effective in removing TD than Bug Fixing and New Development, given its purpose, it's not living up to the expectations. Refactoring, by definition, is performed to increase the code quality. Even though Refactoring slightly out performs the other activities in removing TD, it still introduces substantial amount of TD in a high proportion of cases, and in half of the cases, does not introduce observable changes in the overall amount of TD accumulated in the project. These results are similar to the previous studies (e.g., [13], [16], [27]) that suggest that developers might waste a significant proportion of their time dealing with TD in an inefficient way [13] and that refactoring operations were not found to be effective in removing TD items as Code Smells [27], or even were responsible for introducing bugs [16].

Bug Fixing activities, by definition, are not aimed to deal with TD. Bugs are not Technical Debt Items *per-se*; however, they can be the consequence of TD. The TD removed by bug fixing activities might be accidental, e.g., due to the deletion of parts of the code, or refactorings being embedded in a bug-fixing-tagged commit.

This overall lack of effectiveness of refactoring to repay TD might be owing to the fact that developers do not refactor the code only to remove TD, but also to make design or architectural changes to enable other modification in the code base. Other explanations might be: i) certain refactoring operations are not specifically designed to mitigate TD, and ii) that certain operation can have non-trivial side effects that introduce TD in a bigger amount than the TD that helps removing. And lastly, not all the TD can be resolved by simply refactoring the code.

B. RQ2: The effect of Refactoring Operations on TD

In response to our second research question, we observe that some types of Refactoring Operations are more effective in removing TD such as *Extract Method* and *Pull Up*

Table III
REFACTORING OPERATIONS SUMMARY OF RESULTS - THE TABLE SUMMARIZES THE TOTAL NUMBER OF DETECTED CASES, THE DIVISION OF *Removed*, *Introduced*, AND *No Changes* CASES WITH THEIR RESPECTIVE RATIOS, AND THE AMOUNT OF INTRODUCED AND REMOVED TD IN MINUTES.

	Refactoring Operation	**# Total**	**# Removed**	**TD Minutes Removed**	**# Introduced**	**TD Minutes Introduced**	**# No Change**	**% Removed**	**% Introduced**	**% No Change**
1	Replace Variable with Attribute	2	1	-14	1	49	0	50	50	1
2	Replace Attribute	1	1	-65	0	0	0	100	0	0
3	Rename Variable	23	8	-443	11	168	4	34.78	47.83	17.39
4	Rename Parameter	25	12	-596	8	156	5	48	32	20
5	Rename Method	28	16	-640	6	243	6	57.14	21.43	21.43
6	Rename Class	21	6	-343	6	424	9	28.57	28.57	42.86
7	Rename Attribute	13	5	-201	4	192	4	38.46	30.77	30.77
8	Push Down Method	3	0	0	0	0	3	0	0	100
9	Pull Up Method	10	10	- 650	0	0	0	100	0	0
10	Pull Up Attribute	2	0	0	0	0	2	0	0	100
11	Parametrize Variable	1	1	-10	0	0	0	100	0	0
12	Move Source Folder	7	0	0	1	15	6	0	14.29	85.71
13	Move Class	682	39	-390	86	1442	557	5.72	12.61	81.67
14	Move Attribute	1	1	-14	0	0	0	100	0	0
15	Move and Rename Class	5	0	0	3	57	2	0	60	40
16	Inline Method	9	3	-76	1	22	5	33.33	11.11	55.56
17	Extract Variable	7	5	-363	1	45	1	71.43	14.29	14.29
18	Extract Superclass	1	1	-65	0	0	0	100	0	0
19	Extract Subclass	1	0	0	0	0	1	0	0	100
20	Extract Method	15	12	-791	3	82	0	80	20	0
21	Extract and Move Method	1	0	0	1	190	0	0	100	0
22	Change Package	9	0	0	5	91	4	0	55.56	44.44
	Total	**867**	**121**	**-4661**	**137**	**3176**	**609**	**-**	**-**	**-**

Method. The results suggest that the refactoring operations which has to do with more than one file, i.e., changing more than one file in the same refactoring (e.g., *Extract and Move Method*, *Move Class*, and *Move and Rename Class*) tend to increase the total amount of TD. Our results are aligned with the findings of [16] that suggest the refactoring operations involving hierarchies are prone to introduce faults. Therefore these refactoring operations should be used cautiously with more accurate code inspection and testing activities.

C. Implications for Research and Practice

- **Implications for researchers**: There is still room for further research in this area. The findings of this study can be used as a guideline to further investigate where the refactoring operations should be used to be more effective, i.e. whether the refactorings are happening in the hotspots (the file with frequent changes) follow the same pattern. Further analysis is required to investigate whether there exists a correlation between the total number of refactorings and the total amount of Technical Debt. These results can lead to a better understanding of how to utilize refactorings and thus improve TD management.
- **Implications for practitioners**: As mentioned before, some specific types of refactoring operations seem to yield better results when applied to the code while others seem to exacerbate the code quality. Practitioners can utilize these results when prioritizing the refactoring operations they use while maintaining the code. Lastly, the refactoring types that contribute to increasing the total amount of TD should be used cautiously.

Generally, the mapping between theoretical constructs and their representation are essential when building tools and maybe of interest to raise as a challenge and focus of the projects when building decision support tools.

VI. THREATS TO VALIDITY

The results of this study are subject to threats to *construct validity*, *internal validity*, and *external validity*.

Construct validity refers to the relationship between the theory and the measurements of the observations. It is the most critical threat for this study which concerns the collection of the data, and it is related to the limited scope of the analysis. We only analyze a limited number of commits. These commits are tagged by developers to identify the type of activities performed on that commit. We rely on a tool to detect refactoring operations. More specifically, the threats to the construct validity of the study are:

- *Imprecise identification of the refactoring operations*: We only studied the refactoring operations that have happened in the commits tagged by the developers as *Refactoring*. Further, to detect the individual refactoring operations in refactored commits, we have used RefactoringMiner. We have tried to mitigate this threat by taking the developers' intent into account and by selecting a state-of-the-art tool whose accuracy has been analyzed.
- *Imprecise calculation of Technical Debt*: We have tried to mitigate this threat by using SonarQube which is

broadly used tool measuring TD and has been also employed in similar research studies (e.g., [23], [28]). We have used the default remediation that SonarQube associates to TD items, since we believe the results can be more repeatable, and also because practitioners might be reluctant to customize static code analysis tools [29].

- *Developers incurring in unintentional TD*: TD per definition refers to taking the shortcut intentionally, but this is the most rare case (as discussed in [5] pp. 153-154). We cannot be certain about whether the TD was introduced intentionally, and we only rely on the results provided by SonarQube. This might threaten our results, and we will investigate this issue in future empirical studies.
- *TD being introduced not in the files affected by refactoring operations*: When analyzing RQ2 we have minimized this threat by circumscribing the analysis to the files affected by each refactoring operation.

Threats to internal validity refer to confounding factors that might affect the results. The first threat to the internal validity comes from the association between the main activity tagged by developers in the commit and the different activities that can contain in reality. We make the analysis relying on the information tagged by the developer, but a given commit might of course comprise several different activities. However we are analyzing the main *intent* expressed by the developer. In the case of a refactoring commit, it can contain not only refactoring operations, but the goal of the commit is improving the quality of the code, therefore one can expect a positive impact on the global TD of the project. Our results might have been affected by this fact and we plan to deeper investigate this phenomena in further empirical studies.

Threats to external validity refer the generalizability of the results. In this study, we have analyzed a large scale industrial project which is mainly developed in Java. We understand that the generalizability of the results is limited, and we can only claim that our results are applicable to the analyzed context. We plan to replicate this study not only in other industrial but also in Open-Source projects.

VII. Conclusion

In this paper we present an empirical study for investigating the impact of refactoring, bug fixing, and new development on technical debt, further delving into specific refactoring operations to assess their impact. We have analyzed $2,286$ commits from a large scale industrial project, commit by commit in file level. Our results, within the studied project, show that overall Refactoring help mitigating TD. However we have also found that the majority of the cases Refactoring has no effect on TD, and it can even contributes to the accumulation of TD, which is in line with previous results (e.g., [27]). The TD is mainly introduced during the development of new features, although we have also observed that commits tagged as new development can help repaying TD, which might be owing to the fact that commits tagged as new development might contain refactoring operations embedded, as found in previous research [15]. Finally, Bug fixing was found to contribute to the repayment of TD in the studied period.

Further, we have investigated the impact of specific refactoring operations (ROs) on TD. The results suggest that some ROs namely Extract Method, Pull Up Method, Rename Method, Rename Parameter, and Rename Variable help to remove TD while the ROs that deal with more than one file namely, Extract and Move Method, Move Class, and Move and Rename Class increases the total amount of TD. These results are aligned with previous studies (e.g., [16]), that although addressing similar research questions, focused only on code smells and performed a more coarse-grained analysis on open source projects.

This study supports, within the limits of the threats of validity, that even though ROs are thought as a means to mitigating TD, in some cases, they might contribute to the accumulation of TD if not applied with care. This does not necessarily mean that refactoring operations lower the code quality, but what our results in the analyzed context suggest is that certain operations might tend to introduce new problems in the code (i.e., TD) while might not be effective on solving the problem the developer had in mind. Further, from a managerial point of view, our findings about the impact of refactoring operations on TD can, at the same time, help reduce the waste of effort by developers.

Acknowledgment

This research was supported by the KKS foundation through the SHADE KKS Hög project with ref: 20170176 and through the KKS S.E.R.T. Research Profile with ref. 2018010 project at Blekinge Institute of Technology, SERL Sweden.

References

[1] W. Cunningham, "The wycash portfolio management system," *ACM SIGPLAN OOPS Messenger*, vol. 4, no. 2, pp. 29–30, 1993.

[2] P. Kruchten, R. L. Nord, and I. Ozkaya, "Technical debt: From metaphor to theory and practice," *IEEE Software*, vol. 29, no. 6, pp. 18–21, 2012.

[3] N. Rios, M. G. de Mendonça Neto, and R. O. Spínola, "A tertiary study on technical debt: Types, management strategies, research trends, and base information for practitioners," *Information and Software Technology*, vol. 102, pp. 117–145, 2018.

[4] P. Avgeriou, P. Kruchten, I. Ozkaya, and C. Seaman, "Managing technical debt in software engineering (dagstuhl seminar 16162)," in *Dagstuhl Reports*, vol. 6, no. 4, 2016.

[5] P. Kruchten, R. Nord, and I. Ozkaya, *Managing Technical Debt: Reducing Friction in Software Development*. Pearson, 2019.

[6] Z. Li, P. Avgeriou, and P. Liang, "A systematic mapping study on technical debt and its management," *Journal of Systems and Software*, vol. 101, pp. 193–220, mar 2015.

[7] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 2018.

[8] R. C. Martin, *Clean Code*. Prentice Hall, 2008.

[9] A. Ampatzoglou, A. Ampatzoglou, P. Avgeriou, and A. Chatzigeorgiou, "A Financial Approach for Managing Interest in Technical Debtb," in *International Symposium on Business Modeling and Software Design*, 2016, pp. 117–133.

[10] G. Suryanarayana, G. Samarthyam, and T. Sharma, *Refactoring for Software Design Smells: Managing Technical Debt*. Elsevier Science, 2014, vol. 11.

[11] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.

[12] A. Martini, T. Besker, and J. Bosch, "Technical debt tracking: Current state of practice: A survey and multiple case study in 15 large organizations," *Science of Computer Programming*, vol. 163, pp. 42–61, 2018.

[13] T. Besker, A. Martini, and J. Bosch, "Software developer productivity loss due to technical debt—a replication and extension study examining developers' development work," *Journal of Systems and Software*, vol. 156, pp. 41–61, 2019.

[14] F. Palomba, A. Zaidman, R. Oliveto, and A. De Lucia, "An Exploratory Study on the Relationship between Changes and Refactoring," in *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*. IEEE, may 2017, pp. 176–185.

[15] M. Kim, T. Zimmermann, and N. Nagappan, "A field study of refactoring challenges and benefits," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. ACM, 2012, p. 50.

[16] G. Bavota, B. De Carluccio, A. De Lucia, M. Di Penta, R. Oliveto, and O. Strollo, "When does a refactoring induce bugs? an empirical study," in *2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation*. IEEE, 2012, pp. 104–113.

[17] F. Palomba, G. Bavota, M. D. Penta, R. Oliveto, and A. D. Lucia, "Do They Really Smell Bad? A Study on Developers' Perception of Bad Code Smells," in *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE, sep 2014, pp. 101–110.

[18] G. Bavota, A. De Lucia, M. Di Penta, R. Oliveto, and F. Palomba, "An experimental investigation on the innate relationship between quality and refactoring," *Journal of Systems and Software*, vol. 107, pp. 1–14, 2015.

[19] J. A. M. Santos, J. B. Rocha-Junior, L. C. L. Prates, R. S. do Nascimento, M. F. Freitas, and M. G. de Mendonça, "A systematic review on the code smell effect," *Journal of Systems and Software*, vol. 144, no. July, pp. 450–477, oct 2018.

[20] N. Yoshida, T. Saika, E. Choi, A. Ouni, and K. Inoue, "Revisiting the relationship between code smells and refactoring," in *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, vol. 2016-July. IEEE, may 2016, pp. 1–4.

[21] D. Silva, N. Tsantalis, and M. T. Valente, "Why we refactor? confessions of GitHub contributors," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2016*. New York, New York, USA: ACM Press, 2016, pp. 858–870.

[22] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. D. Penta, A. De Lucia, and D. Poshyvanyk, "When and Why Your Code Starts to Smell Bad (and Whether the Smells Go Away)," *IEEE Transactions on Software Engineering*, vol. 43, no. 11, pp. 1063–1088, nov 2017.

[23] G. Digkas, M. Lungu, P. Avgeriou, A. Chatzigeorgiou, and A. Ampatzoglou, "How do developers fix issues and pay back technical debt in the Apache ecosystem?" in *IEEE 25th International Conference on Software Analysis, Evolution and Reengineering*, 2018, pp. 153–163.

[24] V. Lenarduzzi, N. Saarimaki, and D. Taibi, "On the diffuseness of code technical debt in open source projects," in *Proceedings - 2019 IEEE/ACM International Conference on Technical Debt*, 2019.

[25] K. Fujiwara, K. Fushida, N. Yoshida, and H. Iida, "Assessing refactoring instances and the maintainability benefits of them from version archives," in *International Conference on Product Focused Software Process Improvement*. Springer, 2013, pp. 313–323.

[26] N. Tsantalis, M. Mansouri, L. M. Eshkevari, D. Mazinanian, and D. Dig, "Accurate and efficient refactoring detection in commit history," in *Proceedings of the 40th International Conference on SoftwarTe Engineering*. ACM, 2018, pp. 483–494.

[27] G. Bavota, A. De Lucia, M. Di Penta, R. Oliveto, and F. Palomba, "An experimental investigation on the innate relationship between quality and refactoring," *Journal of Systems and Software*, vol. 107, pp. 1–14, 2015.

[28] N. Saarimaki, M. T. Baldassarre, V. Lenarduzzi, and S. Romano, "On the accuracy of sonarqube technical debt remediation time," in *2019 45th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE, 2019, pp. 317–324.

[29] C. Vassallo, S. Panichella, F. Palomba, S. Proksch, A. Zaidman, and H. C. Gall, "Context is king: The developer perspective on the usage of static analysis tools," in *25th IEEE International Conference on Software Analysis, Evolution and Reengineering*, 2018.