Summer 6-2020

On the Relationship Between Developer Experience and Refactoring: An Exploratory Study and Preliminary Results

Eman Abdullah AlOmar
Rochester Institute of Technology

Anthony Peruma
Rochester Institute of Technology

Christian D. Newman
Rochester Institute of Technology

Mohamed Wiem Mkaouer
Rochester Institute of Technology

Ali Ouni
University of Quebec at Montreal

Recommended Citation

Eman Abdullah AlOmar, Anthony Peruma, Christian D. Newman, Mohamed Wiem Mkaouer, and Ali Ouni. 2020. On the Relationship Between Developer Experience and Refactoring: An Exploratory Study and Preliminary Results. In Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops (ICSEW'20). Association for Computing Machinery, New York, NY, USA, 342–349. https://doi.org/10.1145/3387940.3392193

On the Relationship Between Developer Experience and Refactoring: An Exploratory Study and Preliminary Results

Eman Abdullah AlOmar
eman.alomar@mail.rit.edu
Rochester Institute of Technology
Rochester, New York, USA

Anthony Peruma
anthony.peruma@mail.rit.edu
Rochester Institute of Technology
Rochester, New York, USA

Christian D. Newman
cnewman@se.rit.edu
Rochester Institute of Technology
Rochester, New York, USA

Mohamed Wiem Mkaouer
mwmvse@rit.edu
Rochester Institute of Technology
Rochester, New York, USA

Ali Ouni
ali.ouni@etsmtl.ca
ETS Montreal, University of Quebec
Montreal, Quebec, Canada

ABSTRACT

Refactoring is one of the means of managing technical debt and maintaining a healthy software structure through enforcing best design practices, or coping with design defects. Previous refactoring surveys have shown that these code restructurings are mainly executed by developers who have sufficient knowledge of the system's design, and disposing of leadership roles in their development teams. However, these surveys were mainly limited to specific projects and companies. In this paper, we explore the generalizability of the previous results though analyzing 800 open-source projects. We mine their refactoring activities, and we identify their corresponding contributors. Then, we associate an expertise score to each contributor in order to test the hypothesis of whether developers with higher scores tend to perform a higher number of refactoring operations. We found that (1) although refactoring is not restricted to a subset of developers, those with higher experiences score tend to perform more refactorings than others; (2) our qualitative analysis of three randomly sampled projects show that the developers who are responsible for the majority of refactoring activities are typically on advanced positions in their development teams, demonstrating their extensive knowledge of the design of the systems they contribute to.

CCS CONCEPTS

• **Software and its engineering → Software evolution**; **Maintaining software**.

KEYWORDS

Software maintenance and evolution, Mining software repositories, Software refactoring, Developer experience, Quality

1 INTRODUCTION

Refactoring has been considered, along with code reviews, as the main quality safeguard and the backbone of managing technical debt [8]. Therefore, understanding the best refactoring practices is highly important. The spectrum of research exploring the practice of refactoring covers a wide variety of dimensions, such as the identification of refactoring opportunities [11, 23, 24], recommendation of adequate refactoring operations [7, 9, 16–19, 30, 35], detection of applied refactorings [12, 32, 40], studying the impact of refactoring on quality [6, 29, 41, 42], the reasons as to why developers refactor their code [25, 26, 31], etc. However, little is known about how the level of experience influences developer refactoring activities. Nevertheless, developer experience directly impacts their ability to estimate software quality, and therefore, their ability to determine the appropriate refactoring strategy that needs to be deployed. Moreover, developers' knowledge of the system's structure and sub-components varies, and so is their privilege to access and modify them. This paper aims to start the discussion around the importance of considering the developer's experience as part of proposing solutions related to refactoring, since their applicability depends on the perception and privilege of the developers in charge.

A couple of refactoring studies have pointed out that refactoring is typically performed by experienced developers: Tsantalis et al. [36] performed a multidimensional empirical study on refactoring activities that included: the proportion of refactoring operations performed on production and test code, the most active refactoring contributors, the relationship between refactorings with releases and testing activity, and the purpose of the applied refactorings. With regard to developer experience, the authors found that the top refactoring contributors had a management role within the project. In another study, Kim et al. [13] surveyed 328 professional software engineers at Microsoft to investigate when and how they do refactoring. They found that developers with different expertise levels experienced five risk factors involved in refactoring, namely, regression bugs, code churns, merge conflicts, time taken from other tasks, the difficulty of performing code reviews after refactoring, and the testing cost. They also investigated the relationship between the refactoring effort and reduction of the number of inter-module dependencies and after release defects. They reported that other factors, such as developer experience, need to be examined as the changes to the number of module dependencies and post-release defects might be caused by such factors other than refactoring. The findings of these studies [13, 22, 36] indicate that experience plays a significant role in the execution of refactoring, yet they were both limited to developer surveys without any concrete evidence from the source code, and they were also limited to a few projects.

In this paper, we explore the hypothesis of whether developers with more experience are most likely to be responsible for a higher number of refactoring activities. We aim to challenge the generalizability of the previous findings indicating that only a subset of developers performs major refactoring activities. Our study is driven by the two following research questions:

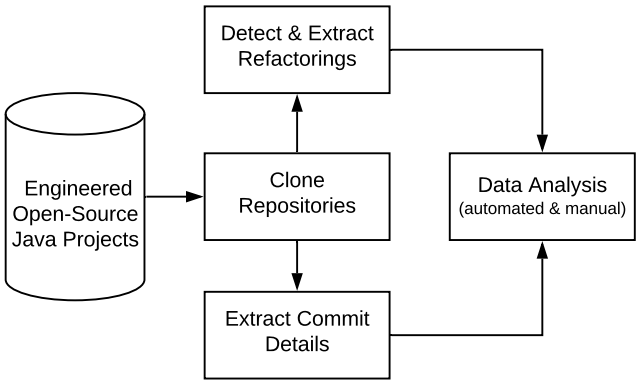


Figure 1: Overview of our methodology

- **RQ1.** What is the distribution of experience among developers that perform refactorings?
  To answer this research question, we start with mining refactorings from 800 well-engineered projects. We identify the subset of authors who were involved in these refactoring activities along with all project contributors. We estimate their experience in the project by measuring their developer commit ratio score. We compare the scores of developers whose commits witnessed refactorings with the scores of developers whose commits had no refactorings.
- **RQ2.** Would higher experience indicate higher refactoring activities?
  The rationale behind this question is investigating whether refactoring activities tend to be performed by a subset of developers. To answer this question, we split developers, based on their experience score, into two sets, where the first set contains the top 5% of developers with high experience scores while the second set gathers the remaining contributors. Then we compared the count of the refactorings performed by each set. We further randomly sampled three projects, and we extracted their top contributors with respect to refactoring both production and test code.

The remainder of this paper is organized as follows. Section 2 outlines our experimental methodology in collecting the necessary refactoring data for the experiments that are discussed afterward in Section 3. Section 4 gathers potential limitations to the validity of our empirical analysis before concluding and describing our future work directions in Section 5.

2 METHODOLOGY

Our research methodology consists of three main phases - Data Collection, Detection & Extraction, and Data Analysis. Figure 1 provides an overview of our methodology. Described below are details of the methodology activities.

To conduct our exploratory study, we used a dataset of well-engineered open-source Java projects. The authors of this dataset [20] curated a set of open-source projects, proven to follow software engineering practices such as documentation, testing, issue and bug tracking, and project management. We chose this dataset as it was also analyzed in previous studies [3, 10, 27] that have been mining refactoring operations, just like our study. In total, our dataset is composed of 800 projects hosted on GitHub. Each project was cloned in order to extract the data needed for our experiments. This data included, but not limited to, each commit author, source files impacted by each committed change, and timestamps etc. The projects in our dataset were cloned in early 2019, and 74.6% of the projects had their most recent commit within the last three years.

Next, we utilized RefactoringMiner [37] to identify refactoring operations occurring in the projects. RefactoringMiner iterates over the commit history of a repository in chronological and compares the changes made to Java source code files in order to detect refactorings. Of the available state-of-the-art set of refactoring detection tools, RefactoringMiner has the highest performance, more specifically, a precision of 98% and a recall of 87% [31, 37]. Running refactoringMiner on the projects under study, resulted scanning a total of 748,001 commits, from which, 111,884 commits contained at least one refactoring operation, and we collected a total of 711,495 refactoring operations. On average, each project contains 732 refactoring commits authored by 19 developers.

Finally, we analyzed the output generated from our detection and extraction activities to answer our research questions. Since our research questions are both quantitative and qualitative, we used tools/scripts along with manual activities to arrive at our findings.

For replication purposes, our dataset and other artifacts are available on our project website [1].

3 RESULTS & DISCUSSION

In this section, we report and discuss our findings for analyzing the identified refactoring-related patterns to answer our two research questions *RQ1 and RQ2*.

3.1 What is the distribution of experience among developers that perform refactorings?

For our experiment on developer experience, we looked at the project contributions made by the developer. In other words, we utilized the volume of commits made to Java source code files by a developer as a proxy for experience. Introduced by [14], this approach calculates the Developer's Commit Ratio (DCR) for each developer in the project. This ratio measures the number of individual commits made by the developer against all project commits. Formally, this ratio is defined as:

$$DCR = \frac{IndividualContributorCommits}{TotalProjectCommits} \quad (1)$$

In our experiment, we only consider the author of a commit as its developer. We followed the same approach as in Peruma et al. [27], where the authors looked at the DCR distribution of developers that perform rename refactorings. As shown in Figure 2, we looked at two types of distributions - developers that only performed non-refactoring operations (depicted as 'Non-Refactoring Commits' in the chart), and developers that performed a mix of refactoring and non-refactoring operations on the source code (depicted as 'All

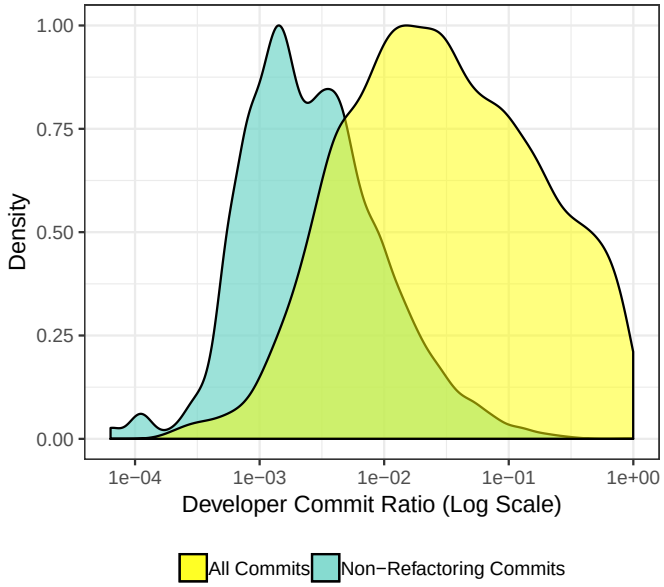


Figure 2: Distribution of DCR values for developers based on the type of commit performed in their project

Table 1: Statistical summary of DCR scores based on the type of commit performed in the project

Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
Non-Refactoring Commits					
0.0001	0.0010	0.0019	0.0031	0.0043	0.0130
All Commits					
0.0002	0.0065	0.0197	0.0456	0.0604	0.2632

Commits' in the chart). Not surprisingly, our dataset had a large proportion of developers that performed a mix of refactoring and non-refactoring operations. These developers also had a higher DCR score.

A clear observation in Figure 2 shows that developers who are involved in the implementation and maintenance of a system have more experience than developers that are exclusively focused on implementing new features. Even though we see an overlap in the density plot, the majority of non-refactoring developers are more concentrated on the lower end of the DCR scale. Furthermore, looking at the statistical summary of DCR scores in Table 1, we see that the average DCR score of a developer performing only non-refactoring commits is 0.0031 while the average DCR score of a developer performing all types of commit operations is 0.0456. Similar to [27] we performed a nonparametric Mann-Whitney-Wilcoxon test on the DCR values for developers that do not perform any refactoring operations and those that did. We obtained a statistically significant p-value (< 0.05) when the DCR values of these two groups of developers were compared. Hence, this shows that developers working exclusively on new features to a system are more likely to have less experience in the project than developers whose work also includes performing refactoring activities. Our findings also confirm the studies carried out by [13, 36] that developer experience is an essential factor when it comes to software refactoring. However, unlike [13, 36], the approach we took relied on a metric (DCR) and was performed automatically over a much larger sample.

> *Summary* Using an alternate approach (i.e., developer contributions), we confirm findings from prior research that more experienced developers are typically involved in refactoring activities in systems. As our approach utilizes existing repository data, and is automated, it provides a non-subjective and scalable approach to estimate the most experienced developers in a project and thereby help to identify developers that are suitable for specific project tasks.

3.2 Would higher experience indicate higher refactoring activities?

In this research question, we investigate whether specific developers are significantly contributing to the overall refactoring of the system, or if it is randomly distributed among all developers. We approach this research question from two fronts - quantitative and qualitative. In the quantitative approach, we perform an empirical and automated study on our dataset. In the qualitative approach, we perform a manual, case study like investigation on a select set of projects.

Quantitative

This part of the research question utilizes the DCR values associated with each developer in the project, along with the total number of refactoring and non-refactoring commits made by the

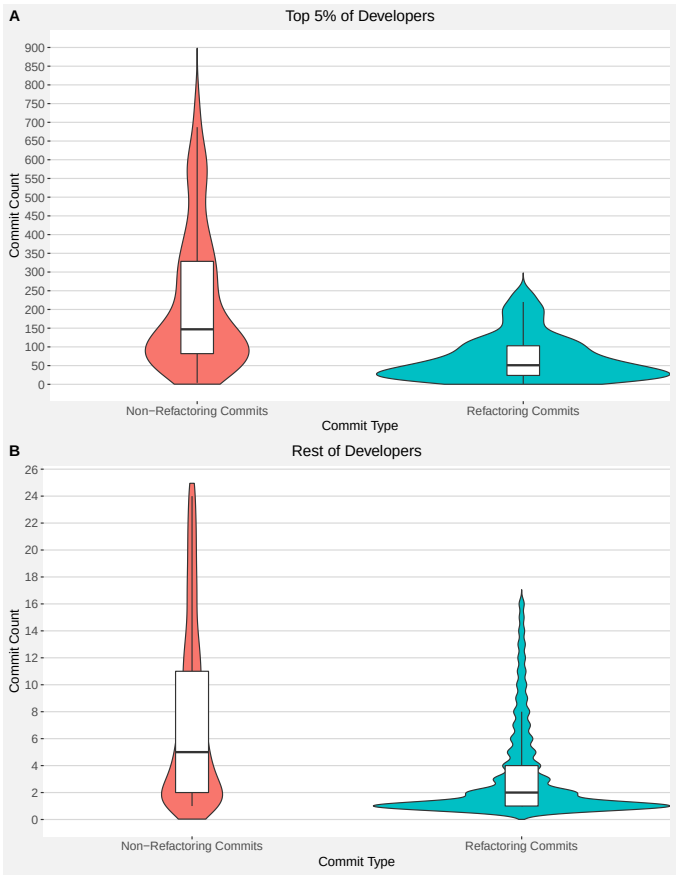


Figure 3: Comparative counts of refactoring and non-refactoring commits for developers. Chart 'A' is for the top 5% of developers, while chart 'B' is for the remaining developers

developer for only Java source files. To perform the comparison, we split the developers into two sets. The first set consisted of developers that fell into the top 5% (labeled as *TOP-5*) of DCR scores while the second set contained the remaining (i.e., 95%) developers. The *TOP-5* of developers equated to approximately a 95% confidence level and confidence interval of 5. Represented by the *TOP-5* are 372 developers, while the remaining developers amount to 7,066. For developers in each of the two sets, we obtained the count of refactoring and non-refactoring commits made by the developer. Figure 3 shows a violin plot of this dataset. Chart 'A' shows the refactoring and non-refactoring commits of the *TOP-5* of developers, while chart 'B' shows the same counts for the remaining developers. A violin plot provides an ideal mechanism to represent our findings as they are useful in providing a visual comparison of multiple distributions. For better interpretation and visualization, we removed outliers from the data via the Tukey's fences approach [39].

Looking at Figure 3, the first clear observation we see is the volume of commit counts made by the two sets of developers. A majority of the *TOP-5* developers contribute significantly more to

Table 2: Statistical summary of the volume of refactoring operations performed by the top 5% and the remaining set of developers

Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
		Top 5%			
1.00	2.00	6.00	11.14	15.00	55.00
		Rest			
1.00	1.00	2.00	3.57	5.00	16.00

the project in terms of refactoring and non-refactoring commits. On average, a *TOP-5* developer makes 70.24 and 223.7 refactoring and non-refactoring commits, respectively. On the other hand, the rest of the developers average around 3.21 and 15.69 refactoring and non-refactoring commits, respectively. Furthermore, the *TOP-5* violin plot shows a high frequency of developers performing, approximately, 15 to 75 refactoring commits. The same does not hold for non-refactoring commits, where we see a higher density within the range of 75 to 125 commits. Additionally, we observed that our dataset contains some developers that perform at most around 300 refactoring commits while non-refactoring commits go up to around 800. Hence, non-refactoring commit counts have a higher variation than refactoring commits. The refactoring box plot is more condensed than the non-refactoring boxplot; this indicates that the data varies less and hence is more consistent.

Finally, we looked at the number of refactoring operations performed by the two groups of developers. It should be noted that a single refactoring commit can contain one or more refactoring operations. A statistical summary of our findings is presented in Table 2, while a comparative histogram is available in Figure 4.Even though the histogram shows a higher volume of refactoring operations by inexperienced developers, it should be noted that this is the cumulative count across all projects in the dataset. If we were to look at the individual developer contributions, we could see that experienced developers apply refactorings more often than inexperienced developers.

Qualitative

To better understand the key role of the *TOP-5* contributors in the development team, we extract refactorings from a select set of projects - Hadoop[1], OrientDB[2], and Camel[3]. These three systems were chosen based on the criteria used in [15] (i.e., had more than 100 stars, had more than 60 forks, had size over 2 MB, these repositories are active and well-used). Next, we cluster production and test files of these projects, by developer ID. Finally, we carefully examine the top contributor's professional profiles to identify their role in the organization hosting the software project. Our findings are detailed below.

Figure 5 portrays the distribution of the refactoring activities on production code and test code performed by project contributors for each software system we examined. The Hadoop project has a total of 114 developers. Among them are 73 (64%) refactoring

[1]https://github.com/apache/hadoop
[2]https://github.com/orientechnologies/orientdb
[3]https://github.com/apache/camel

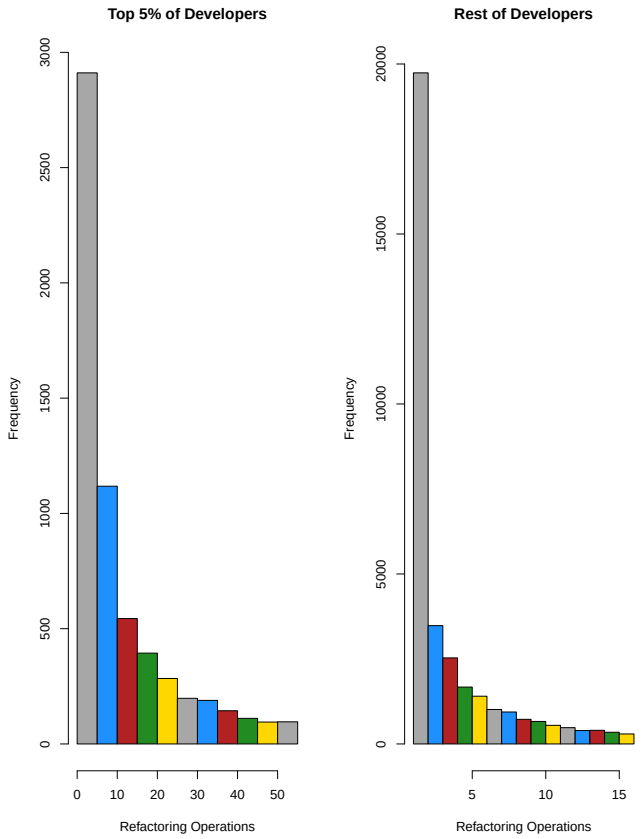


Figure 4: Histogram of refactoring operations performed by the top 5% and the remaining set of developers

contributors. As we observe in Figures 5a and 5b , not all of the developers are major refactoring contributors. The main refactoring contributor has a refactoring ratio of 25% on production code and 10% on test code. Figure 5c and 5d present the percentage of the refactorings for the OrientDB production code and test code. Out of the total 113 developers, 35 (31%) were involved refactoring. The top contributor has a refactoring ratio of 57% and 44% on production and test code respectively. For Camel, in Figures 5e and 5f , 73 (20%) developers were on the refactoring list out of 368 total committers. The most active refactoring contributor has high ratios of 51% and 48% respectively in production and test code. We also note that very few developers applied refactorings exclusively on either production code or test code for the three projects under study.

The manual analysis aligns with the findings of the previous section in distinguishing a subset of developers that monopolize the refactoring activity across the three projects. To identify their key role in the development of the project, we searched, using their GitHub IDs, their professional profiles on Linked-In[4]. We were successful in locating the role of the top contributors for the 3 projects, and we found, through their public affiliation to the project, that they were either development leads or senior developers.

[4]Used in previous studies as a source to identify developers skills and experience.

Our findings show that refactoring activities are mainly performed by a subset of developers who have a management role in the company. Senior developers care more about refactoring the source code to ensure high-quality software and make the software easier for future development. These subsets of developers may perform certain practices when applying code refactoring (e.g., refactoring before and after adding new features, testing frequently to avoid any bugs that may introduce and affect the functionality of the software, and documenting and automating the application of refactoring). One of the reasons that seems not to encourage the other subsets of developers to significantly refactor the code is the technical constraints such as inadequate tool supports or lack of trust of automated support for composite refactorings. A discussion about various barriers to refactoring has been highlighted in Murphy et al. [21].

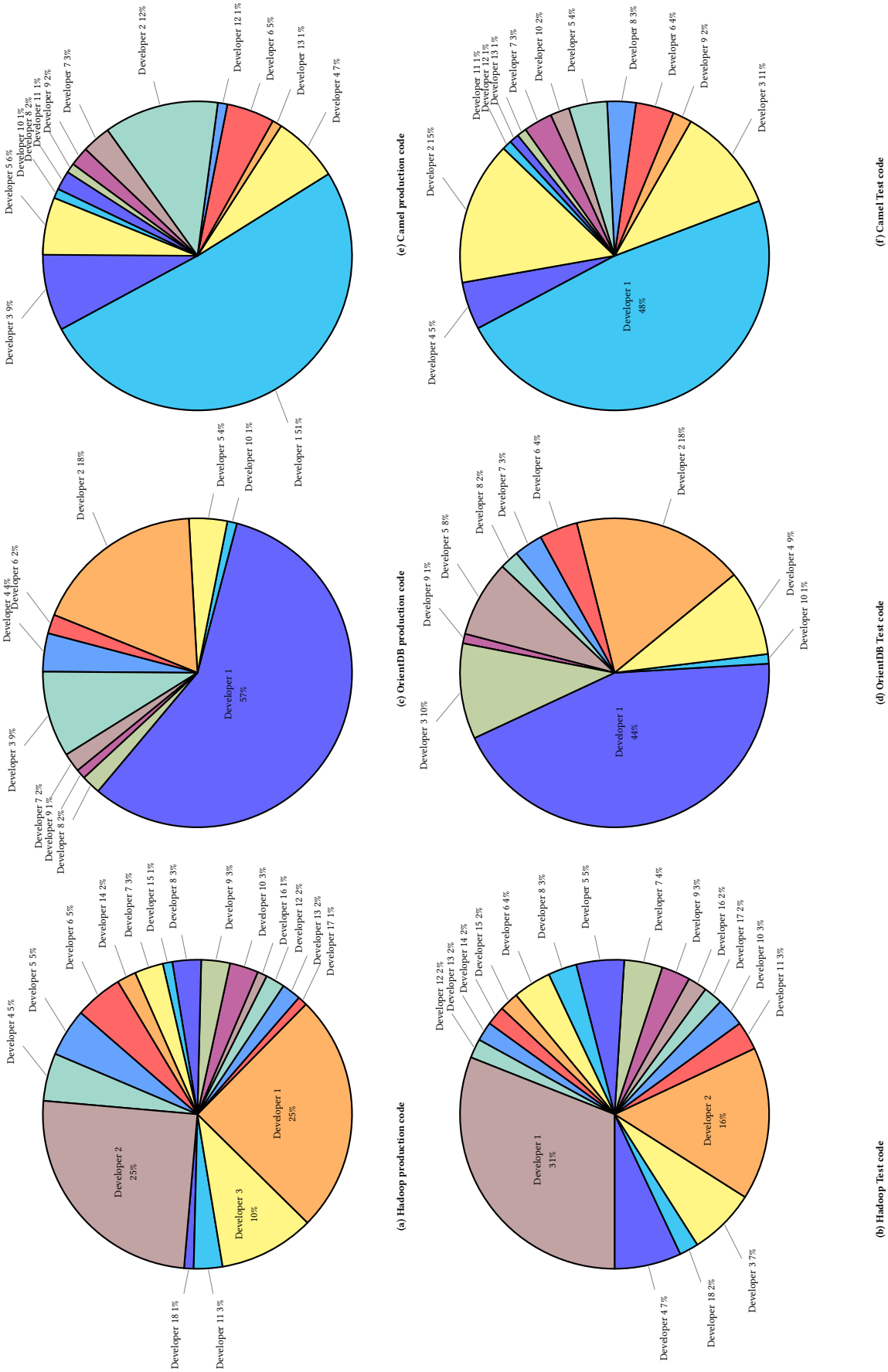
> *Summary.* While refactorings are applied by various developers, only a reduced set of developers are responsible for performing the majority of these activities, in both production and test files. This set of developers take over refactoring activities without necessarily being dominant in other programming activities. As we examine the top contributor's publicly accessible professional profiles, we identify their positions to be advanced in the development team; hence, demonstrating their extensive knowledge of the design of the systems they contribute to.

3.3 Research implications and future directions

While refactoring is being applied by various developers, it would be interesting to evaluate their refactoring practices. We want to capture and better understand the code refactoring best practices and learn from experienced developers so that we can recommend them for other developers. Also, it would be interesting to investigate the difference between the experienced and inexperienced developers in terms of distributions of refactoring operations, i.e., we aim to see if any specific refactoring types are highly solicited by one group compared to the other. As previous studies have already shown, some refactoring operations tend to be more complex than others [21], and so it is interesting for us to validate it in practice.

AlOmar et al. [2] performed an exploratory study on how developers document their refactoring activities in commit messages; this activity is called Self-Affirmed Refactoring (SAR). They found that developers tend to use a variety of textual patterns to document their refactoring activities, such as *refactor*, *move* and *extract*. In follow-up work, AlOmar et al. [3] identified which quality models are more in-line with the developer's vision of quality optimization when they explicitly mention in the commit messages that they refactor to improve these quality attributes. Since we noticed that various developers are responsible for performing refactorings, one potential research direction is to investigate which developers are responsible for the introduction of SARs in order to examine whether or not experience plays a role in the introduction of SARs. Another potential research direction is to study if developer experience is one of the factors that might contribute to the significant

Figure 5: Refactoring contributors in production and test files in Hadoop, OrientDB and Camel



improvement of the quality metrics that are aligned with developer perception tagged in the commit messages. In other words, we would like to evaluate the top contributors refactoring practice against all the rest of refactoring contributors by assessing their contributions on the main internal quality attributes improvement (e.g., cohesion, coupling, and complexity). Furthermore, previous studies analyzed the impact of refactorings on structural metrics and quality attributes [6, 29, 41, 42]. It would be interesting to revisit such analysis while taking into account the degree of expertise of the refactoring contributors. As developers with larger experience and managerial roles have better exposure to the system's design, it is expected that their restructurings are of better quality, and this can be empirically demonstrated.

Furthermore, with regards to the analysis of refactoring and design quality, previous studies investigated how refactorings can be responsible for introducing code smells, and so hindering the design quality [28, 38]. It would be interesting to verify whether such unexpected results can correlate with the developer's experience. Along with hindering design quality, the misuse of refactoring can also be responsible for bugs [4, 5], and various studies have proposed testing strategies to make refactoring safer [33, 34]. One of our future directions is to also correlate the bug-proneness of refactorings with the degree of expertise of the contributors. It is assumed that the lack of functional knowledge may facilitate the introduction of bugs, but this is subject to empirical validation as well.

4 THREATS TO VALIDITY

The first threat is that our analysis is restricted to only open-source, Java, Git-based repositories. However, we were still able to analyze projects that are highly varied in size, contributors, number of commits, and refactorings. Additionally, the representativeness of the dataset can be considered as a threat to this study. However, we mitigate this threat by utilizing 800 engineered projects that have also been part of a prior study on refactoring [27]. Furthermore, the projects are of varying sizes, contributors, and refactoring operations.

The accuracy of the refactoring detection tool also poses a threat to our study. However, previous studies [31, 37] report on high precision and recall scores for RefactoringMiner. However, a drawback to using RefactoringMiner is that the study is limited to Java projects. Our future work includes the use of refactoring mining tools that support other programming languages, such as RefDiff [32], to expand the representativeness of our dataset.

A major threat to validity is related to the calculation of experience. Obtaining the experience of each and every developer is a challenge for our study, given the volume of data in our dataset and also that experience can be subjective. Hence, we adopted a mechanism (i.e., DCR), used by prior research [14, 27], where we utilized project contributions as a proxy for experience. The reasoning behind the measurement assumes that the longer a developer is involved in a project, and the more they contribute to it, the more experienced they become. Such an assumption may not hold for some specific scenarios; however, since the projects in our dataset are heterogeneous in nature, our assumption holds.

5 CONCLUSION

We present an exploratory study of the level of experience of developers that apply refactorings. Prior studies use smaller samples to study similar questions; however, in our study, we have examined a more extensive and representative set of systems by comparison. Since we can confirm results from prior, manual studies automatically, we have identified a way to obtain similar results automatically. This means that it is possible to now study the impact of developer experience on a larger scale.

In future work, we plan to leverage the results from this study to determine specific types of refactorings made by developers at different experience levels. We would also like to explore ways to leverage this data to help suggest/recommend refactorings or suggest/recommend refactoring methodology based on the developers level of experience.

ACKNOWLEDGMENTS

We would like to thank the authors of Refactoring Miner for publicly providing it.

REFERENCES

[1] [n.d.]. Project Website. https://sites.google.com/g.rit.edu/refactoring/.
[2] Eman Abdullah AlOmar, Mohamed Wiem Mkaouer, and Ali Ouni. 2019. Can Refactoring Be Self-Affirmed? An Exploratory Study on How Developers Document Their Refactoring Activities in Commit Messages. In *Proceedings of the 3rd International Workshop on Refactoring* (Montreal, Quebec, Canada) *(IWOR '19)*. 8. https://doi.org/10.1109/IWoR.2019.00017
[3] Eman Abdullah AlOmar, Mohamed Wiem Mkaouer, Ali Ouni, and Marouane Kessentini. 2019. On the impact of refactoring on the relationship between quality attributes and design metrics. In *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 1–11.
[4] Everton LG Alves, Myoungkyu Song, and Miryung Kim. 2014. RefDistiller: a refactoring aware code review tool for inspecting manual refactoring edits. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 751–754.
[5] Gabriele Bavota, Bernardino De Carluccio, Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, and Orazio Strollo. 2012. When does a refactoring induce bugs? an empirical study. In *IEEE 12th International Working Conference on Source Code Analysis and Manipulation*. 104–113.
[6] Gabriele Bavota, Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, and Fabio Palomba. 2015. An experimental investigation on the innate relationship between quality and refactoring. *Journal of Systems and Software* 107 (2015).
[7] Gabriele Bavota, Andrea De Lucia, Andrian Marcus, and Rocco Oliveto. 2014. Recommending refactoring operations in large software systems. In *Recommendation Systems in Software Engineering*. Springer, 387–419.
[8] Zadia Codabux and Byron Williams. 2013. Managing technical debt: An industrial case study. In *2013 4th International Workshop on Managing Technical Debt (MTD)*. IEEE, 8–15.
[9] Marcos César de Oliveira, Davi Freitas, Rodrigo Bonifácio, Gustavo Pinto, and David Lo. 2019. Finding needles in a haystack: Leveraging co-change dependencies to recommend refactorings. *Journal of Systems and Software* 158 (2019), 110420.
[10] Sarah Fakhoury, Devjeet Roy, Adnan Hassan, and Vernera Arnaoudova. 2019. Improving source code readability: theory and practice. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. IEEE, 2–12.
[11] Francesca Arcelli Fontana, Pietro Braione, and Marco Zanoni. 2012. Automatic detection of bad smells in code: An experimental assessment. *Journal of Object Technology* 11, 2 (2012), 5–1.
[12] Miryung Kim, Matthew Gee, Alex Loh, and Napol Rachatasumrit. 2010. RefFinder: a refactoring reconstruction tool based on logic query templates. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*. ACM, 371–372.
[13] Miryung Kim, Thomas Zimmermann, and Nachiappan Nagappan. 2014. An empirical study of refactoringchallenges and benefits at microsoft. *IEEE Transactions on Software Engineering* 40, 7 (2014), 633–649.
[14] D. E. Krutz, N. Munaiah, A. Peruma, and M. Wiem Mkaouer. 2017. Who Added That Permission to My App? An Analysis of Developer Permission Changes in Open Source Android Apps. In *2017 IEEE/ACM 4th International Conference*

on Mobile Software Engineering and Systems (MOBILESoft). 165–169. https://doi.org/10.1109/MOBILESoft.2017.5

[15] Stanislav Levin and Amiram Yehudai. 2017. Boosting Automatic Commit Classification Into Maintenance Activities By Utilizing Source Code Changes. In *Proceedings of the 13th International Conference on Predictive Models and Data Analytics in Software Engineering* (Toronto, Canada) *(PROMISE)*. ACM, New York, NY, USA, 97–106. https://doi.org/10.1145/3127005.3127016

[16] Mohamed Wiem Mkaouer, Marouane Kessentini, Slim Bechikh, Kalyanmoy Deb, and Mel Ó Cinnéide. 2014. High dimensional search-based software engineering: finding tradeoffs among 15 objectives for automating software refactoring using NSGA-III. In *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation*. 1263–1270.

[17] Mohamed Wiem Mkaouer, Marouane Kessentini, Slim Bechikh, Mel Ó'Cinnéide, and Kalyanmoy Deb. 2014. Software refactoring under uncertainty: a robust multi-objective approach. In *Proceedings of the Companion Publication of the 2014 Annual Conference on Genetic and Evolutionary Computation*. 187–188.

[18] Mohamed Wiem Mkaouer, Marouane Kessentini, Mel Ó Cinnéide, Shinpei Hayashi, and Kalyanmoy Deb. 2017. A robust multi-objective approach to balance severity and importance of refactoring opportunities. *Empirical Software Engineering* 22, 2 (2017), 894–927.

[19] Wiem Mkaouer, Marouane Kessentini, Adnan Shaout, Patrice Koligheu, Slim Bechikh, Kalyanmoy Deb, and Ali Ouni. 2015. Many-objective software remodularization using NSGA-III. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 24, 3 (2015), 1–45.

[20] Nuthan Munaiah, Steven Kroh, Craig Cabrey, and Meiyappan Nagappan. 2017. Curating GitHub for engineered software projects. *Empirical Software Engineering* 22, 6 (2017), 3219–3253.

[21] Emerson Murphy-Hill and Andrew P. Black. 2008. Breaking the Barriers to Successful Refactoring: Observations and Tools for Extract Method. In *Proceedings of the 30th International Conference on Software Engineering* (Leipzig, Germany) *(ICSE '08)*. Association for Computing Machinery, New York, NY, USA, 421–430. https://doi.org/10.1145/1368088.1368146

[22] Christian D Newman, Mohamed Wiem Mkaouer, Michael L Collard, and Jonathan I Maletic. 2018. A study on developer perception of transformation languages for refactoring. In *Proceedings of the 2nd International Workshop on Refactoring*. 34–41.

[23] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, and Denys Poshyvanyk. 2013. Detecting bad smells in source code using change history information. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 268–278.

[24] Fabio Palomba, Annibale Panichella, Andrea De Lucia, Rocco Oliveto, and Andy Zaidman. 2016. A textual-based technique for smell detection. In *2016 IEEE 24th international conference on program comprehension (ICPC)*. IEEE, 1–10.

[25] A. Peruma. 2019. A Preliminary Study of Android Refactorings. In *2019 IEEE/ACM 6th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. 148–149. https://doi.org/10.1109/MOBILESoft.2019.00030

[26] Anthony Peruma, Mohamed Wiem Mkaouer, Michael J Decker, and Christian D Newman. 2018. An empirical investigation of how and why developers rename identifiers. In *Proceedings of the 2nd International Workshop on Refactoring*. ACM, 26–33.

[27] A. Peruma, M. W. Mkaouer, M. J. Decker, and C. D. Newman. 2019. Contextualizing Rename Decisions using Refactorings and Commit Messages. In *2019 19th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 74–85. https://doi.org/10.1109/SCAM.2019.00017

[28] Anthony Peruma, Christian D. Newman, Mohamed Wiem Mkaouer, Ali. Ouni, and Fabio Palomba. 2020. An Exploratory Study on the Refactoring of Unit Test Files in Android Applications. In *Proceedings of the 4th International Workshop on Refactoring* (Seoul, South Korea) *(IWoR 2020)*. Association for Computing Machinery, New York, NY, USA.

[29] Gustavo H Pinto and Fernando Kamei. 2013. What programmers say about refactoring tools? an empirical investigation of stack overflow. In *Proceedings of the 2013 ACM workshop on Workshop on refactoring tools*. 33–36.

[30] Luca Rizzi, Francesca Arcelli Fontana, and Riccardo Roveda. 2018. Support for architectural smell refactoring. In *Proceedings of the 2nd International Workshop on Refactoring*. 7–10.

[31] Danilo Silva, Nikolaos Tsantalis, and Marco Tulio Valente. 2016. Why we refactor? confessions of github contributors. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 858–870.

[32] Danilo Silva and Marco Tulio Valente. 2017. RefDiff: detecting refactorings in version histories. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 269–279.

[33] Gustavo Soares, Diego Cavalcanti, Rohit Gheyi, Tiago Massoni, Dalton Serey, and Márcio Cornélio. 2009. Saferefactor-tool for checking refactoring safety. (01 2009).

[34] Gustavo Soares, Rohit Gheyi, Dalton Serey, and Tiago Massoni. 2010. Making program refactoring safer. *IEEE software* 27, 4 (2010), 52–57.

[35] Ricardo Terra, Marco Tulio Valente, Sergio Miranda, and Vitor Sales. 2018. JMove: A novel heuristic and tool to detect move method refactoring opportunities. *Journal of Systems and Software* 138 (2018), 19–36.

[36] Nikolaos Tsantalis, Victor Guana, Eleni Stroulia, and Abram Hindle. 2013. A multidimensional empirical study on refactoring activity. In *Proceedings of the 2013 Conference of the Center for Advanced Studies on Collaborative Research*. IBM Corp., 132–146.

[37] Nikolaos Tsantalis, Matin Mansouri, Laleh M Eshkevari, Davood Mazinanian, and Danny Dig. 2018. Accurate and efficient refactoring detection in commit history. In *Proceedings of the 40th International Conference on Software Engineering*. ACM.

[38] Michele Tufano, Fabio Palomba, Gabriele Bavota, Rocco Oliveto, Massimiliano Di Penta, Andrea De Lucia, and Denys Poshyvanyk. 2015. When and why your code starts to smell bad. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 403–414.

[39] John W Tukey. 1977. *Exploratory data analysis*. Vol. 2. Reading, Mass.

[40] Zhenchang Xing and Eleni Stroulia. 2006. Refactoring detection based on umldiff change-facts queries. In *2006 13th Working Conference on Reverse Engineering*. IEEE, 263–274.

[41] Zhenchang Xing and Eleni Stroulia. 2006. 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, 458–468.

[42] Norihiro Yoshida, Tsubasa Saika, Eunjong Choi, Ali Ouni, and Katsuro Inoue. 2016. Revisiting the relationship between code smells and refactoring. In *IEEE 24th International Conference on Program Comprehension (ICPC)*. 1–4.