A Large-Scale Empirical Study on Refactoring Usage on GitHub

Anonymous Author(s)

ABSTRACT

Code refactoring is a significant activity for software development and evolution. Despite its importance, we understand surprisingly little about refactoring and its impact on software development and evolution. Previous studies have focused on various aspects of refactoring: developing refactoring detection algorithms; investigating the motivations and responsibilities behind refactoring operations; and assessing the relations between refactoring activities and software quality. To fill this gap, this work makes three contributions. First, it presents a large-scale refactoring dataset containing over 2 Million refactoring operations from 11,933 open source Java projects. Second, it integrates this refactoring dataset with the *Boa* infrastructure so that it could be utilized by the MSR community. Third, it investigates the impact of refactoring activities on the software evolution and quality of four project categories based on two dimensions: whether they have refactorings; and whether they are developed by GitHub organization or users. Our key findings are: i) the number of refactoring operations is positively correlated to the number of commits in the software repositories; ii) more refactoring activities are performed systematically in *Organization* project with more coupling between objects (CBO) rather than *User* project; iii) refactoring operations are less likely to be applied in the classes with high hierarchy complexity; iv) compared with method-level refactorings, class-level refactoring operations are less often performed in classes with high complexity; v) refactoring activities are typically performed by file major contributors.

ACM Reference Format:
Anonymous Author(s). 2020. A Large-Scale Empirical Study on Refactoring Usage on GitHub. In *MSR '20: Mining Software Repositories, May 25–26, 2020, Yongsan-gu, Seoul, Korea.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Applying refactoring operations in software development and evolution is commonly used practice to developers. However, the impact of refactoring activity on software development and evolution is still an open question to researchers. In previous refactoring studies, researchers worked on developing refactoring detection algorithms to compute a list of refactoring operations performed between two versions of the source code [4, 13, 22, 31, 41]. Some researchers investigated the motivation behind refactoring operations [35, 40]. Other researchers focused on investigating the relations between refactoring activities and software evolution in the scope of code quality/smells [8, 28], code review [3, 16, 36], code types [40], code changes [30], bug fixes [7, 20] and version releases [19, 20, 32]. However, no extant studies have explored the impact of refactoring activities on software development with a large-scale dataset consisting of different project categories. What kind of projects need code refactoring? What are the impacts on the projects with refactorings versus those without refactorings? How do refactorings affect team-based projects versus individual projects?

Inspired by these questions, we conducted an empirical study of refactorings based on a large-scale dataset generated by *Boa* [5]. This dataset is composed of 11,933 open-source Java repositories and more than 2 million refactoring operations in 15 different refactoring types as detected by *RMINER* [41]. We classified projects into four categories based on two dimensions, whether they have refactoring activities (i.e. *Refactoring* and *NoRefactoring*) and whether they are owned by a GitHub organization (i.e. *User* and *Organization*). To analyze the dataset in these four categories, we formulated the following research questions:

RQ1 Are *Refactoring* projects larger than *NoRefactoring* projects?
RQ2 Do *Refactoring* projects have higher code complexity than *NoRefactoring* projects?
RQ3 Are refactoring activities mostly applied in classes with high complexity?
RQ4 Do *Organization* projects contain more refactoring operations than *User* projects?
RQ5 Do major contributors apply more refactoring operations in *User* projects or *Organization* projects? Do developers perform refactoring operations in their own files?

The key findings of this study are: **1)** *Refactoring* projects have larger size than *NoRefactoring* projects (and *Organization* larger than *User*) in terms of the size metrics: the number of commits, AST nodes, contributors, and project lifetime. **2)** The detected number of refactorings in a project is strongly positively correlated to the number of commits of the project, and other size metrics. **3)** *Refactoring* projects are significantly more complex than *NoRefactoring* projects with respect to both the quantity and the ratio of highly coupled classes. **4)** The detected refactoring types are likely applied in classes with smaller inheritance hierarchies and low lack cohesion of methods. **5)** Compared with method-level refactorings, class-level refactoring operations are less often performed in classes with high complexity in C&K metrics. **6)** *Organization* projects usually contain more refactoring operations and higher refactoring density. **7)** Most detected refactoring operations in files are performed by file owners and major contributors, especially for GitHub user repositories.

This paper makes the following additional contributions:

- Generated a large *Boa* refactoring dataset containing 11,933 open source Java repositories and 2 million+ detected refactoring operations based on 15 different refactoring types.
- Proposed *Relative Complexity*, an approach to estimate project-level quality based on class-level quality metric and a collection of software repositories.

MSR '20, May 25–26, 2020, Yongsan-gu, Seoul, Korea
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

2 METHODOLOGY

2.1 Study Design

The goal of this paper is to study the impacts of code refactoring. This study was conducted in two phases. In the first phase, we created a large-scale refactoring dataset containing 11,933 Java projects (including project metadata, entire commit history, and all file contents) and 2,372,924 refactoring descriptions associated to its relevant commit. Second, we analyzed the extracted data based on the four classified project categories. An overview is presented in Figure 1 and described below.

2.2 *Boa* Refactoring Dataset

2.2.1 Boa Java Dataset. *Boa* consists of a computational and data infrastructure and a domain-specific language (DSL) designed to ease mining large-scale software repositories [14]. We chose this mining tool for dataset generation, refactoring detection, and data analysis to take advantage of its scalability. We selected a large collection of open source Java projects recommended by Allamanis and Sutton [1]. This set was selected not only because it is a curated corpus of open source Java projects with above-average quality from GitHub, but also because it covers a wide variety of domains. Using this set of projects, we generated a *Boa* dataset containing 11,933 Java projects from GitHub with 22,621,474 Java files. The generated *Boa* dataset contains most of the repository data which includes project metadata, commit information of all branches, and the corresponding file contents under each commit. To analyze this dataset, users can write a query in *Boa*'s domain-specific language. Another feature of *Boa* is its extensibility in both the data infrastructure and the domain-specific language which facilitated our task of creating a comprehensive refactoring dataset.

2.2.2 Refactoring Detection Tool. To detect all the refactoring operations along with the entire commit history of each project, we used *RMINER* by Tsantalis et al. [41]. *RMINER* extracts a list of performed refactoring operations (refactoring descriptions) between two versions of a Java source file. Furthermore, *RMINER* is reported as a high precision and recall refactoring detection tool in comparison to the state-of-the-art [41]. Another reason for selecting this tool was that *RMINER* can be utilized through its external API. Therefore, we can extend the domain-specific language in *Boa* with *RMINER* as a plugin in order to detect refactoring activities in parallel. It is important to note that the current *RMINER*[1] can detect 40 refactoring types. For this study we only consider the 15 types (shown in Table 1) from the original paper [41].

2.2.3 Boa Refactoring Dataset. The generated *Boa* Java dataset mentioned in Section 2.2.1 keeps the source code content for each file. This new storage technique moves the source code parsing phase to the runtime of queries. This approach helps extend the DSL with other programming analysis tools that require source code as input, such as *RMINER*. With this new mechanism, we extended *Boa*'s DSL with *RMINER* as a plugin to detect refactoring operations applied in the entire commit history of each project in the dataset. Note that we don't consider merge commits in the commit history. This is because merge commits may involve refactoring operations that have already been applied in a feature branch. After we collected the detected refactoring descriptions in the dataset, we extended the dataset by associating the detected refactoring descriptions to their corresponding commits. We store all commit and associated refactoring descriptions pairs. Finally, the dataset is completed by combining the project data and the refactoring data. Table 2 shows the statistics of the refactoring dataset. In previous work [8, 19], there was no comprehensive refactoring dataset including complete information of commit history and file contents associated to refactoring operations.

[1]https://github.com/tsantalis/RefactoringMiner

Table 1: Refactoring Types

AST Node	Refactoring Type
Package	*Change Package*
Type	*Move Class, Rename Class, Move And Rename Class, Extract Superclass, Extract Interface*
Method	*Move Method, Extract Method, Inline Method, Pull Up Method, Extract And Move Method, Rename Method*
Field	*Move Attribute, Pull Up Attribute, Push Down Attribute*

Table 2: Refactoring Dataset

Dataset Metric	Count
Projects	11,933
Commits	4,648,490
Java Files	22,621,474
AST Nodes	12,673,911,435
Refactoring Commits	461,234
Detected Refactorings	2,372,924
Refactoring Types	15

2.3 Analysis Method

2.3.1 Project Classification and Selection. To answer our research questions, we classified the dataset into four categories based on two dimensions: whether they have refactoring (i.e. *Refactoring* and *NoRefactoring*) and whether they are developed by GitHub organization (i.e. *User* and *Organization*). The *User* and *Organization* projects are classified according to the `type` label, indicating the repository ownership, in the project metadata provided by GitHub. After that, we selected projects with a repository lifetime longer than one month to filter out toy projects. As a result, the projects selected for further analysis contain 9,200 projects and 2,123,238 refactoring descriptions. A statistical overview of the analyzed projects is presented in Table 3, where each metric except `Projects` is the count of the entities through the entire commit histories.

Table 3: Statistics of Analyzed Projects

Dataset Metrics	*NoRefactoring*		*Refactoring*		TOTAL
	User	*Organization*	*User*	*Organization*	
# Projects	1,164	422	4,813	2,801	9,200
# Commits	44,716	71,975	1,214,676	3,257,297	4,588,664
# Java Files	127,383	325,735	5,221,038	16,670,372	22,344,528
# Classes	284,896	398,818	7,521,390	24,677,012	32,882,116
# Fields	3,592,202	3,092,425	38,545,405	118,103,090	163,333,122
# Methods	2,916,596	9,927,562	65,417,301	237,323,983	315,585,442
# AST Nodes	213,060,776	532,869,264	2,772,445,264	9,040,716,293	12,559,091,597
# Refactoring Commits	0	0	127,832	285,035	412,867
# Detected Refactorings	0	0	613,435	1,509,803	2,123,238

2.3.2 Data Extraction. We wrote *Boa* queries to extract refactoring descriptions and a set of metrics to answer the research questions based on the four classified project categories. For RQ1 and RQ4, we measured the general project metrics in terms of the number of commits, AST nodes, contributors, and project lifetime (months)

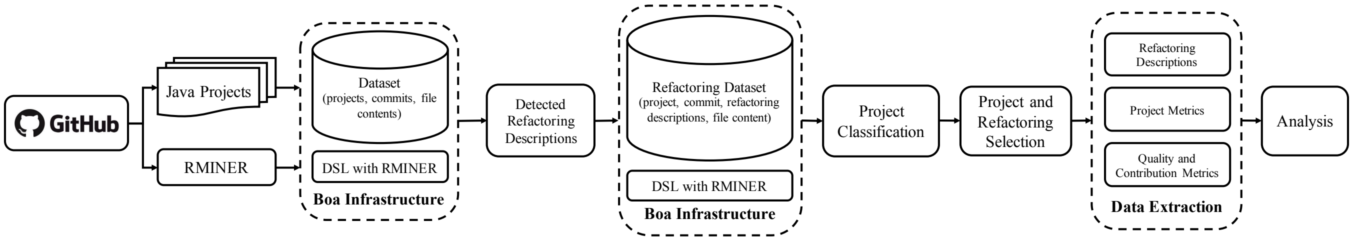

GitHub
Java Projects
RMINER
Dataset (projects, commits, file contents)
DSL with RMINER
Boa Infrastructure
Detected Refactoring Descriptions
Refactoring Dataset (project, commit, refactoring descriptions, file content)
DSL with RMINER
Boa Infrastructure
Project Classification
Project and Refactoring Selection
Refactoring Descriptions
Project Metrics
Quality and Contribution Metrics
Data Extraction
Analysis

Figure 1: Overview of Study Methodolgy

as project size indicator. To answer RQ2 and RQ3, we estimated *Relative Complexity* (Section 2.3.3), based on class-level quality metrics, of each analyzed project to measure the project-level software quality. We choose Chidamber & Kemerer metrics as the class-level quality indicator [12]. To associate the file contributions to refactoring activity for RQ5, we mine contribution metrics reported by Bird et al. [9]. The definitions of all used quality and contribution metrics are provided in Table 4.

Table 4: Quality and Contribution Metrics

Quality Metrics	Definition
Weighted Methods per Class (WMC)	The sum of cyclomatic complexity for the methods in the class.
Depth of Inheritance Tree (DIT)	The the length of the longest path in the inheritance tree rooted by the class.
Number Of Children (NOC)	The number of immediate descendants of the class.
Response for a Class (RFC)	The sum of distinct invoked functions and constructors in the class.
Coupling Between Object (CBO)	The number of classes coupled to the class.
Lack of Cohesion of Methods (LCOM)	The cohesiveness of the methods in the class.
Contribution Metrics	**Definition**
File Owner	Developer who contributes most commits to a file.
File Minor Contributor	Contribution is less than 5% to the file.
File Major Contributor	Contribution is larger than 5% to the file.

2.3.3 Project Relative Complexity. The C&K metrics are considered important indicators for measuring code quality, but the metrics are used to directly measure class quality – not for an entire project. Since we need to compare the code quality between projects, we need another mechanism for measuring project-level quality. Another problem is that the concept of code quality is usually subjective. To solve these problems, we introduce *Relative Complexity* based on class-level quality metrics and a collection of projects. The key idea is that the concept of code complexity should be more objective and the threshold for determining a project as complex should be relative and referential to the other projects in the analyzed dataset.

Relative Complexity consists of two indicators: **ComplexClassNum** in Equation 1 and **ComplexClassRatio** in Equation 5. Let I be the indicator function such that $I(e)$ equals to 1 if event e is true else 0.

$$\texttt{ComplexClassNum}(p, m) = \sum_{c \in C} \texttt{ComplexClass}(c, m) \qquad (1)$$

ComplexClassNum calculates the number of relative complex classes in project p. Larger values of `ComplexClassNum` for project p indicate higher project-level complexity in the metric m.

$$\texttt{ComplexClass}(c, m) = I(f_m(c) > T_m) \qquad (2)$$

ComplexClass estimates whether a class c is relatively complex in class-level metric m (1 is complex and 0 is non-complex), given threshold T_m. f_m computes metric m for a class.

$$T_m = \frac{\sum_{p \in P} \texttt{AvgMetric}(p, m)}{|P|} \qquad (3)$$

$\boldsymbol{T_m}$, used for determining the relative complex class in metric m, is estimated based on the average of the `AvgMetric` of each project p in the collection P.

$$\texttt{AvgMetric}(p, m) = \frac{\sum_{c \in C} f_m(c) \cdot I(f_m(c) > 0)}{\sum_{c \in C} I(f_m(c) > 0)} \qquad (4)$$

AvgMetric calculates the average non-zero metric value of all the classes in the set C of project p. In this study, we select the latest snapshot as the set of classes C for each project.

$$\texttt{ComplexClassRatio}(p, m) = \frac{\texttt{ComplexClassNum}(p, m)}{|C|} \cdot 100\% \qquad (5)$$

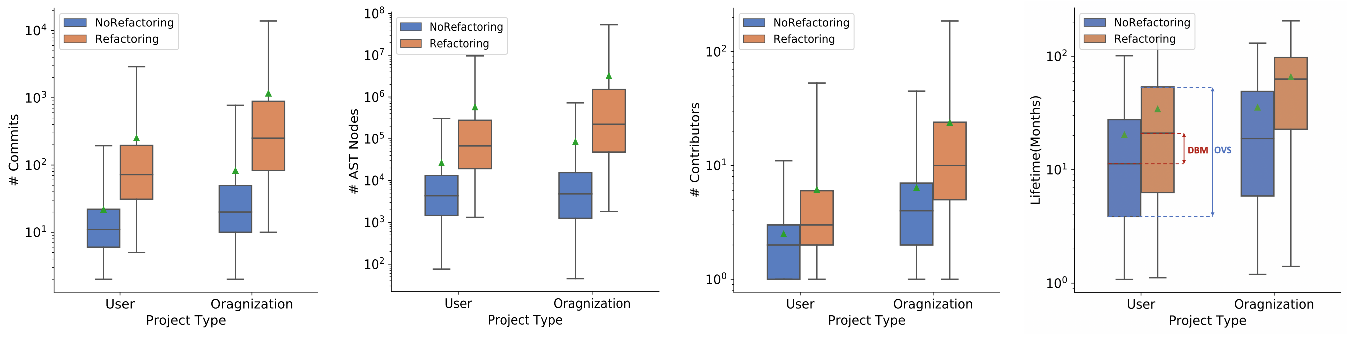
ComplexClassRatio estimates the proportion of the relative complex classes in project p. Similar to `ComplexClassNum`, the higher the `ComplexClassRatio` is, the more complex the project p is in metric m.

3 ANALYSIS OF THE RESULTS

In this section, we discuss the findings of our analysis for each research question. Additionally, we present observations of the results and suggest possible explanations.

RQ1: Are *Refactoring* projects typically larger than *NoRefactoring* projects?

It is widely believed that refactoring activity plays a significant role in software maintenance by improving code structure and readability. Intuitively, a software repository with a large number of commits and AST nodes could contain more refactoring operations due to an increased maintenance workload. Additionally, more readable code could help improve effective team working for software projects. Therefore, the number of contributors involved in the project may also affect the frequency of refactoring activities. Finally, sustainability directly influences a project's lifetime, but refactoring might impact a project's lifetime. To verify our assumptions, we analyzed project size metrics. Figure 2 provides the box



(a) # Commits **(b)** # AST Nodes **(c)** # Contributors **(d)** Lifetime (months)

Figure 2: Project size metrics in the *Refactoring*, *NoRefactoring*, *User*, and *Organization* projects. The x-axis and legend indicate the four categories of software repository, and the y-axis presents the corresponding metric value.

plots (omitting outliers for readability) of the size metrics including the number of commits, AST nodes, contributors, and lifetime.

Refactoring VS NoRefactoring. The first thing you may notice is that *Refactoring* projects have a dominating advantage in quantity based on the size metrics. In other words, the **Interquartile Range** (IQR) of the *Refactoring* groups is higher than the *NoRefactoring* groups for both *User* and *Organization*. IQR is the distance between the 25th and 75th percentile that covers the middle 50% of the data. Especially in Figures 2a and 2b, there are no overlapped IQRs between the *Refactoring* and *NoRefactoring* groups in both *User* and *Organization* projects. The results indicate that *Refactoring* projects have larger size than *NoRefactoring* projects, with respect to the number of the commits and AST nodes. For the *Organization* group, in Figures 2c and Figure 2d, the boxes overlap without medians. According to Wild et al.'s guidelines of call and estimate effect sizes [43], we can conclude that for *Organization* groups, *Refactoring* projects tend to have more contributors and a longer lifetime than *NoRefactoring* projects. If two boxes overlap with both medians such as the *User* projects in Figures 2c and 2d, then the **Distance Between Medians** (DBM) as a percentage of **Overall Visible Spread** (OVS) is required to indicate the difference between the two groups [43]. The DBM% can be evaluated with equation 6, where the DBM is the exact distance between two medians in two distributions and the OVS is the overall distance of both IQRs.

$$DBM\% = \frac{DBM}{OVS} \times 100\% \tag{6}$$

The calculated DBM%'s for Figure 2 are provided in the Table 5. The percentages under each column present the DBM% between the two subgroups in the category. For example, the values in the *User* column of the table indicate the DBM percentages between the two subgroups (i.e. *Refactoring* and *NoRefactoring*) in the *User* projects for all size metrics. If the DBM% is larger than 10% for group *A* and *B* with both sample sizes over 1,000, there tends to be a difference between *A* and *B*. Since the sample size of *Organization NoRefactoring* projects is 422 (shown in Table 3), 20% is the threshold. Therefore, only the DBM%'s larger than 20% can indicate a difference between two boxes that overlap with medians in the columns *Organization* and *NoRefactoring*. According to Table 5,

Table 5: DBM%'s for Figure 2. Values in parentheses are thresholds, where if DBM% is larger than the threshold, there exists a significant difference between two boxes (shown as ✓). X indicates there is no significant difference between two subgroups. A (—) indicates no overlap between two boxes.

Metric	***Refactoring* VS *NoRefactoring***		***User* VS *Organization***	
	User (10%)	***Organization*** (20%)	***NoRefactoring*** (20%)	***Refactoring*** (10%)
# Commits	—	—	21% ✓	21% ✓
# AST Nodes	—	—	5% X	10% ✓
# Contributors	20% ✓	27% ✓	33% ✓	32% ✓
Project Lifetime	20% ✓	48% ✓	17% X	46% ✓

there exists significant differences between the two subgroups in the *User* projects in Figures 2c and 2d.

Finding 1: *Refactoring* projects have larger size than *NoRefactoring* projects w.r.t. to the following size metrics: number of commits, AST nodes, contributors, and project lifetime.

The implication for this finding is that refactoring recommending tools can suggest refactoring activity in large projects.

User VS Organization. Since most pairs of boxes overlap with medians, we need to check their DBM%'s. All the box differences in the *Refactoring* projects are critical since all the DBM%'s are larger than 10%. However, this is not the case for *NoRefactoring* projects since the DBM%'s are less than 20% in the metrics Contributors and Lifetime. Without the classification of *Refactoring* or *NoRefactoring*, *Organization* projects typically have larger size metrics than *User* projects, with DBM% over 10% in our dataset, similar to the results reported by Munaiah et al. [26].

Finding 2: *Organization* projects are typically larger than *User* projects, especially for those containing refactoring activities.

The implication for this finding is that team-based projects need more refactoring recommendations.

These results lead us to further analysis of the correlations between the refactoring activity and the project size metrics, which are shown in Figure 3. First, we immediately notice that for both *User* and *Organization* the number of commits is strongly positively correlated with the number of refactorings. Second, *Organization* projects typically have higher positive correlations between the number of refactorings and size metrics, except AST node numbers. Finally, all project size metrics show positive correlations with the number of refactorings.

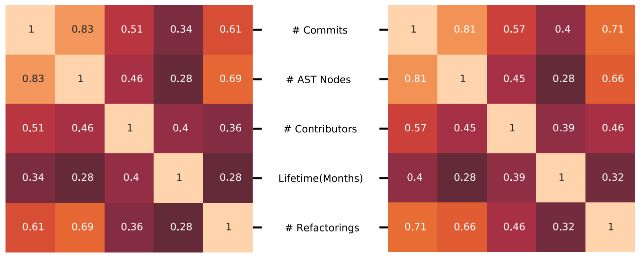


Figure 3: *User* (left) VS *Organization* (right)

Finding 3: Detected number of refactorings in a project is strongly positively correlated to number of commits in the project.

The implication for this finding is that the number of commits in a project could be used as a feature for predicting refactoring activities.

RQ2: Do *Refactoring* projects have higher code complexity than *NoRefactoring* projects?

Software developers usually need to spend more effort to maintain a software program due to its high code complexity. Refactoring operations are introduced to improve code design and lower its complexity. We thus assume that software with higher code complexity should contain more refactoring operations to ensure its code quality. To confirm this, we conduct an experiment to investigate this research question. We introduced *Relative Complexity* for measuring project code quality based on a collection of projects. Since we are using C&K metrics as class-level quality indicators, we calculated the *Relative Complexity* for each metric (*WMC*, *DIT*, *NOC*, *RFC*, *CBO*, and, *LCOM*), where the results are shown in Figure 4.

ComplexClassNum - Refactoring vs NoRefactoring. Since most pairs of boxes overlap with medians in Figure 4a, the DBM%'s are provided in Table 6a to identify the significant differences. *WMC* is used to measure class complexity by counting the number of independent control flows through all the methods belonging to it. The larger the number of all possible control flows in the method, the more complex it could be since developers must analyze more paths. Similarly, the more methods with a large number of control flows in a class, the more difficult for programmers to identify the class's purpose. In other words, higher values of *WMC* indicate a class is more complex to developers. In Table 6a, the DBM%'s in the *User* and *Organization* columns are 22% and 21% respectively for the metric *WMC*. Thus we can conclude that *Refactoring* projects usually contain more relative complex classes than *NoRefactoring* projects with respect to *WMC*.

DIT is evaluated by measuring the length of the longest path in the inheritance tree rooted by a class. In this case, the deeper an inheritance tree under a class, the more methods and fields of it could be reused by its descendants, which increases the reusability of the system. However, it also increases maintenance complexity since those inherited methods are associated with more classes. Moreover, the deeper a class is in the tree, the more difficult it is for developers to identify its purpose since it inherits many methods and fields. In other words, the higher the *DIT* value for a class, the more complex the class could be. For *DIT*, there exists a critical difference between *Refactoring* and *NoRefactoring* in the *User* projects but not in the *Organization* projects. This result is a little surprising to us. This is because both *Extract Interface* and *Extract Superclass* refactoring types are detected in both *User* projects and *Organization* projects. Since both refactoring types should increase the depth of inheritance tree in the system, we excepted *Refactoring* projects to have more classes with higher depth. The possible reasons for no critical difference presented in the *Organization* projects could be: 1) The latest snapshots of each analyzed project may contain a large number of classes that could dilute the difference. 2) These two types of refactorings are performed relatively rarely, as shown later in Table 7.

Table 6: DBM%'s for Figures 4a/4b. Values in parentheses are thresholds, where if DBM% is larger than the threshold, there exists a significant difference between two boxes (shown as ✓). X indicates there is no significant difference between two subgroups.

(a) DBM%'s for `ComplexClassNum` distribution in Figure 4a.

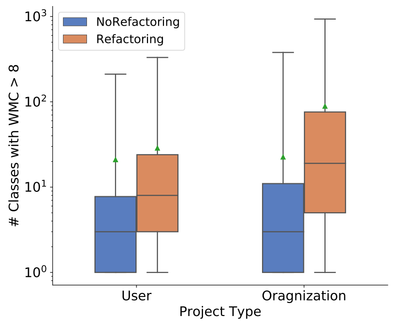
Metric	*Refactoring* **VS** *NoRefactoring*		*User* **VS** *Organization*	
	User (10%)	*Organization* (20%)	*NoRefactoring* (20%)	*Refactoring* (10%)
WMC	22% ✓	21% ✓	0% X	15% ✓
DIT	14% ✓	12% X	13% X	13% ✓
NOC	0% X	0% X	22% ✓	15% ✓
RFC	24% ✓	23% ✓	0% X	16% ✓
CBO	16% ✓	18% X	3% X	14% ✓
LCOM	27% ✓	22% ✓	10% X	15% ✓

(b) DBM%'s for `ComplexClassRatio` Distribution in Figure 4b.

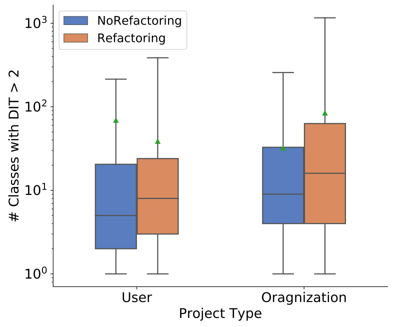
Metric	*Refactoring* **VS** *NoRefactoring*		*User* **VS** *Organization*	
	User (10%)	*Organization* (20%)	*NoRefactoring* (20%)	*Refactoring* (10%)
WMC	14% ✓	16% X	0% X	1% X
DIT	4% X	21% ✓	12% X	5% X
NOC	13% ✓	38% ✓	13% X	15% ✓
RFC	7% X	16% X	9% X	0% X
CBO	19% ✓	34% ✓	5% X	14% ✓
LCOM	13% ✓	23% ✓	11% X	1% X

NOC counts the number of direct subclasses under a class. Similar to *DIT*, the more children a class has, the more reusable the methods in it. However, a class with more children can incur more maintenance effort from developers. In the field of program analysis, both *DIT* and *NOC* are in debate among researchers since a higher value could be both good and bad to a system [11, 12, 17, 23, 34, 38, 44]. But, for this study, we consider a class with higher *DIT* or *NOC* as a more relatively complex class in a project. The corresponding DBM%'s shown in Table 6a are both 0% in *User* and *Organization* projects. In this case, we can say that *Refactoring* and *NoRefactoring* projects have no critical difference with respect to *NOC*.

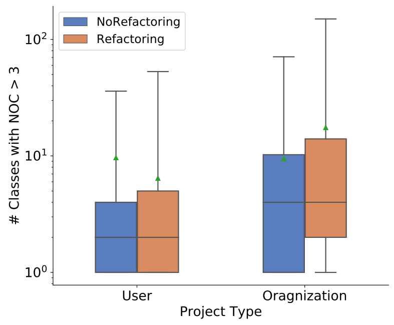
RFC is evaluated by counting invoked functions and constructors in a class. A larger number of invocations in a class usually indicates that developers need to spend more time debugging and testing to ensure correctness. In other words, the higher the *RFC* of a class, the more complex the class could be. The metric *CBO* measures the interaction between a class and other classes. Given a class *C*, it can be calculated as the number of classes that import the class *C* and the number of classes imported by the class *C*. A class with high *CBO* indicates that it is more difficult to be modified as a change might propagate to other classes. The metric *LCOM* estimates the cohesion of the methods in a class. If a class has lower cohesion of methods, it is difficult for developers to figure out the responsibilities of the class since the methods in it may be serving several needs. In other words, the higher the *LCOM* for a class, the lower cohesion of methods are in the class. In this study, we use LCOM5 [18]. For DBM%'s of *RFC*, *CBO*, and *LCOM*, most of them are larger than the thresholds (except *CBO* in *Organization*).



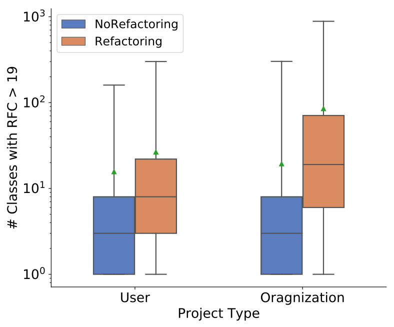
(a1) WMC



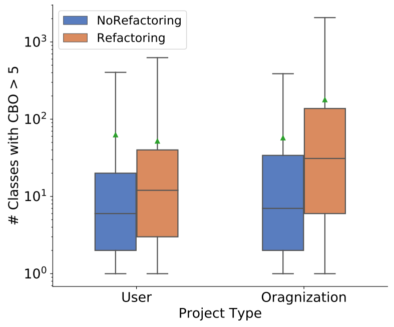
(a2) DIT



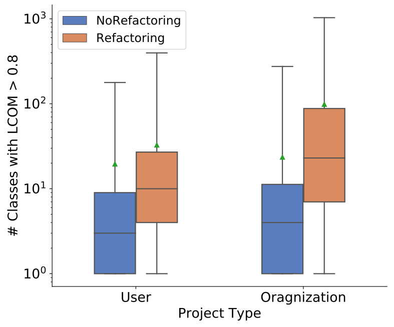
(a3) NOC



(a4) RFC

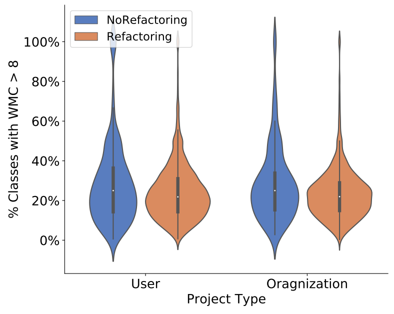


(a5) CBO

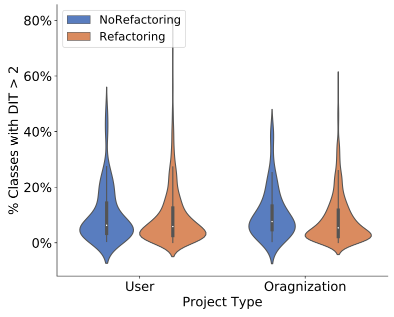


(a6) LCOM

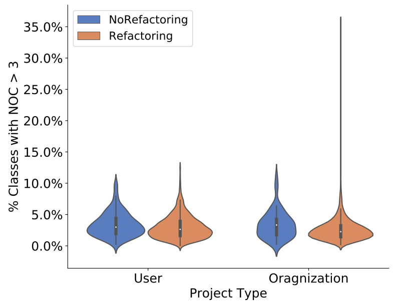
(a) `ComplexClassNum` Distribution



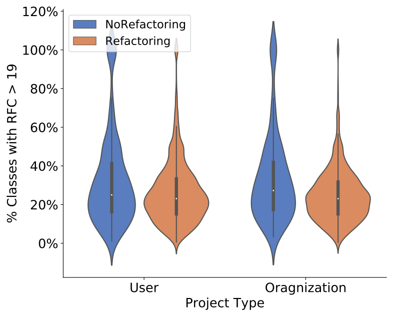
(b1) WMC



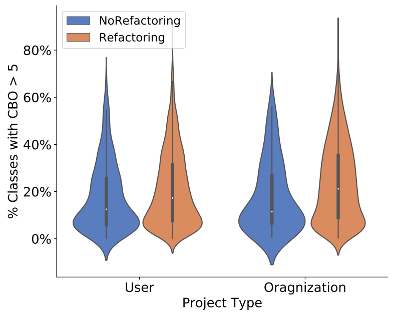
(b2) DIT



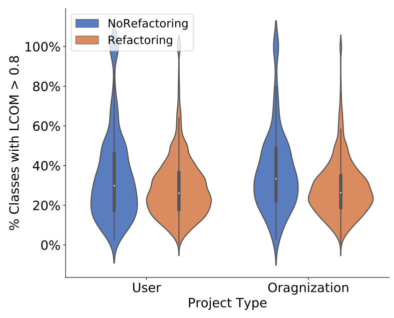
(b3) NOC



(b4) RFC



(b5) CBO



(b6) LCOM

(b) `ComplexClassRatio` Distribution

Figure 4: *Relative Complexity* of analyzed projects. The x-axis and the legend indicate the four project categories, and the y-axis shows the value of `ComplexClassNum`(`ComplexClassRatio`) calculated by Equation 1(5). The thresholds for each metric is calculated in Equation 3.

Finding 4: *Refactoring* projects typically have larger number of relatively complex classes than *NoRefactoring* projects, for metrics *WMC*, *RFC*, and *LCOM*.

The implication for this finding is that refactoring recommending tools can suggest refactoring activity in the projects with a large number of complex classes.

ComplexClassNum - User vs Organization. The first thing we find is that there is no critical difference between the *User* and *Organization* groups for *NoRefactoring* projects for most C&K metrics (except *NOC*). The results indicate if projects do not contain refactoring activities, then they usually have a similar number of relatively complex classes. Conversely, critical differences exist in the *Refactoring* projects for all metrics.

Finding 5: *Organization Refactoring* projects usually have more relative complex classes than *User Refactoring* projects.

The implication for this finding is that team-based projects containing refactoring activities need more maintenance effort.

ComplexClassRatio - Refactoring vs NoRefactoring. The fraction of relatively complex classes in each project should also be considered for analyzing project-level complexity. The results are shown in Figure 4b and its DBM%'s are shown in Table 6b. For *NOC*, *CBO*, and *LCOM* there exist critical differences between *Refactoring* and *NoRefactoring* in both *User* and *Organization* projects. Note that DBM% only indicates the difference between the two boxes, but it cannot present which one is higher. In Figure 4b, *Refactoring* projects have lower `ComplexClassRatio` than *NoRefactoring* projects in metrics *NOC* and *LCOM* but not for *CBO*. It seems that *Refactoring* projects typically have higher *CBO* classes, not only more classes total but also in higher proportion.

ComplexClassRatio - User vs Organization. The results show there are almost no significant differences between *User* and *Organization* for both *Refactoring* and *NoRefactoring* projects, except *NOC* and *LCOM* in *Refactoring* projects.

Finding 6: *Refactoring* projects are significantly more complex than *NoRefactoring* projects, with respect to both the quantity and the ratio of complex classes in metric *CBO*.

The implication for this finding is that code changes should be carefully applied in the projects containing refactoring activities.

RQ3: Are refactoring activities mostly applied in classes with high complexity?

As seen in the previous section, *Refactoring* projects usually contain more relative complex classes. Here we investigate whether refactoring operations are applied in the relatively complex classes. We estimate the class qualities with C&K metrics in the **Classes Before Refactoring** (CBR) along the commit history of the analyzed projects. After that we use the metric thresholds provided in Figure 4 to determine whether an estimated class is complex or non-complex in each metric. Since developers and researchers believe that refactoring operations would benefit software maintainability, we expect more refactoring operations would be applied in the relatively complex classes. However, it is not the case according to our results shown in Figure 5.

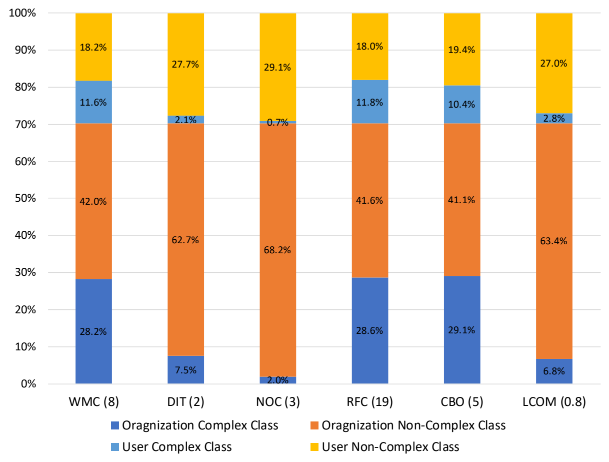


Figure 5: *User* VS *Organization* with class complexity distribution in the CBR for each metric, where the complex/non-complex class is defined by Equation 2. The values in the parentheses are the thresholds for each metric.

For *WMC*, most refactoring operations (18.2% + 42.0% = 60.2%) are applied to classes with lower relative complexity. Similarly, 59.6% and 60.5% of CBR are not relatively complex for metrics *RFC* and *CBO*. In addition, for the metrics *DIT*, *NOC*, and *LCOM*, over 90% of CBR are non-complex. It seems that developers do not prefer to apply refactorings in relatively complex classes with respect to *DIT*, *NOC*, and *LCOM*. Note that around 70% of CBR are from *Organization* projects while only 30% of them are from *User* projects. However, both categories follow the general trend that refactoring operations are less likely to be performed in relatively complex classes. Especially for large inheritance hierarchies and high lack of cohesion of methods.

Do all the detected refactoring types applied in the complex and non-complex classes follow the same trend? To answer this question we calculate the proportions of refactoring operations applied in the complex/non-complex classes for each refactoring type and each metric. The result is shown in Table 7. If we go over the metrics *DIT*, *NOC*, and *LCOM* for all refactoring types, we see that all of them are less likely (from 1.4% to 15.2%) to be performed in the complex classes. Thus developers seem to apply refactoring operations in classes that have simple hierarchies and that is more cohesive, possibly to avoid introducing faults. Bavota et al. [7] found the applied refactorings involving hierarchy could induce faults, which also supports our finding.

Finding 7: Detected refactoring types are mostly applied in non-complex classes for metrics *DIT*, *NOC*, and *LCOM*.

The implication for this finding is that recommendation tools should rarely suggest refactoring operations in the classes with high complexity hierarchies.

When we examine the refactoring types, we find that all class-level refactoring operations are more likely performed in non-complex classes for all metrics. Note that *Move Class*, *Rename Class*, and *Move And Rename Class* are the top 3 most applied refactoring types at the class level. Compared with class-level, method-level and field-level refactorings are actually more likely performed in complex classes for the metrics *WMC* and *RFC*.

Table 7: Class complexity distribution in the CBR for each refactoring type and each C&K metric. **NC**: The non-complex class with the metric value less or equal to the threshold. **C**: The complex class with the metric value larger than the threshold. The sum of two percentages in **NC** and **C** is 100% for each refactoring type and each metric. The numbers in the parentheses under metric columns are the thresholds for each metric. The percentages in the parentheses under refactoring type column are the proportions of the type/category in all detected refactoring operations.

Refactoring Type		WMC		DIT		NOC		RFC		CBO		LCOM	
		NC (≤ 8)	C (> 8)	NC (≤ 2)	C (> 2)	NC (≤ 3)	C (> 3)	NC (≤ 19)	C (> 19)	NC (≤ 5)	C (> 5)	NC (≤ 0.8)	C (> 0.8)
Class Level (31.9%)	**Move And Rename Class (1.9%)**	🟥 80.6%	🟥 19.4%	🟥 93.5%	🟥 6.5%	🟥 98.6%	🟥 1.4%	🟥 78.9%	🟥 21.1%	🟨 72.9%	🟨 27.1%	🟥 92.3%	🟥 7.7%
	Rename Class (5.4%)	🟥 77.0%	🟥 23.0%	🟥 92.0%	🟥 8.0%	🟥 98.2%	🟥 1.8%	🟥 75.1%	🟥 24.9%	🟨 70.9%	🟨 29.1%	🟥 91.2%	🟥 8.8%
	Move Class (23.0%)	🟥 76.7%	🟥 23.3%	🟥 91.3%	🟥 8.7%	🟥 97.5%	🟥 2.5%	🟥 77.6%	🟥 22.4%	🟨 71.9%	🟨 28.1%	🟥 90.2%	🟥 9.8%
	Extract Superclass (1.2%)	🟨 68.6%	🟨 31.4%	🟥 87.9%	🟥 12.1%	🟥 98.2%	🟥 1.8%	🟨 66.6%	🟨 33.4%	🟨 63.3%	🟨 36.7%	🟥 88.4%	🟥 11.6%
	Extract Interface (0.4%)	🟨 57.8%	🟨 42.2%	🟥 85.8%	🟥 14.2%	🟥 95.0%	🟥 5.0%	🟨 71.8%	🟨 28.2%	🟨 51.2%	🟨 48.8%	🟥 91.4%	🟥 8.6%
Method Level (54.6%)	**Inline Method (1.9%)**	🟩 19.8%	🟩 80.2%	🟥 90.5%	🟥 9.5%	🟥 96.9%	🟥 3.1%	🟩 17.8%	🟩 82.2%	🟦 40.4%	🟦 59.6%	🟥 92.8%	🟥 7.2%
	Move Method (8.9%)	🟩 20.7%	🟩 79.3%	🟥 90.3%	🟥 9.7%	🟥 97.2%	🟥 2.8%	🟦 27.5%	🟦 72.5%	🟦 43.9%	🟦 56.1%	🟥 92.1%	🟥 7.9%
	Pull Up Method (6.4%)	🟦 25.2%	🟦 74.8%	🟥 85.8%	🟥 14.2%	🟥 96.5%	🟥 3.5%	🟦 39.5%	🟦 60.5%	🟦 46.3%	🟦 53.7%	🟥 93.8%	🟥 6.2%
	Extract Method (12.1%)	🟦 27.1%	🟦 72.9%	🟥 91.2%	🟥 8.8%	🟥 96.5%	🟥 3.5%	🟩 19.3%	🟩 80.7%	🟦 42.5%	🟦 57.5%	🟥 91.9%	🟥 8.1%
	Rename Method (21.1%)	🟦 30.6%	🟦 69.4%	🟥 90.3%	🟥 9.7%	🟥 97.1%	🟥 2.9%	🟦 44.4%	🟦 55.6%	🟦 54.7%	🟦 45.3%	🟥 89.7%	🟥 10.3%
	Extract And Move Method (4.2%)	🟦 32.8%	🟦 67.2%	🟥 84.8%	🟥 15.2%	🟥 96.6%	🟥 3.4%	🟩 24.3%	🟩 75.7%	🟦 47.1%	🟦 52.9%	🟥 92.5%	🟥 7.5%
Field Level (12.9%)	**Push Down Attribute (0.9%)**	🟦 30.3%	🟦 69.7%	🟥 92.9%	🟥 7.1%	🟥 90.0%	🟥 10.0%	🟦 42.0%	🟦 58.0%	🟦 42.8%	🟦 57.2%	🟥 90.6%	🟥 9.4%
	Pull Up Attribute (3.3%)	🟦 34.6%	🟦 65.4%	🟥 89.3%	🟥 10.7%	🟥 97.5%	🟥 2.5%	🟦 43.0%	🟦 57.0%	🟦 49.1%	🟦 50.9%	🟥 86.0%	🟥 14.0%
	Move Attribute (8.7%)	🟦 36.3%	🟦 63.7%	🟥 93.2%	🟥 6.8%	🟥 97.6%	🟥 2.4%	🟦 34.4%	🟦 65.6%	🟨 51.7%	🟨 48.3%	🟥 84.8%	🟥 15.2%

🟥 locates where the **NC** > 75%. 🟩 locates where the **C** > 75%. 🟨 locates where the **C** < **NC** < 75%. 🟦 locates where the **NC** < **C** < 75%.

Finding 8: Class-level refactoring operations are less likely performed in classes with high complexity in C&K metrics.

The implication for this finding is that recommendation tools should rarely suggest class-level refactoring operations in complex classes.

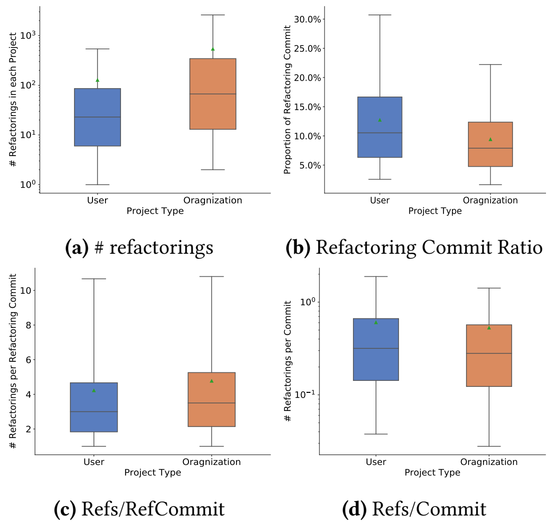
For the method-level refactorings, *Inline Method*, *Move Method*, *Extract Method*, and *Extract And Move Method* are mostly applied (over 75%) in the complex classes for metrics *WMC* and *RFC*. If a class has high *WMC*, then the class usually has more methods or the methods in it contain more paths. *Inline Method*, *Move Method*, and *Pull Up Method* can reduce the number of methods in a given class to lower the *WMC* value. *RFC* is calculated by counting the number of distinct methods and constructors invoked in the class. In this case, *Inline Method* could be used to reduce method calls. Note that in the detected refactoring operations, the top 3 most applied refactoring types are: *Move Class*, *Rename Method*, *Extract Method*.

Finding 9: Method-level refactorings are more actively applied in classes with high *WMC* and *RFC*, especially for *Extract Method*, *Inline Method*, and *Move Method*.

The implication for this finding is that refactoring recommending tools should suggest more method-level refactorings in the highly weighted classes.

RQ4: Do *Organization* projects contain more refactoring operations than *User* projects?

In general, *Organization* projects have larger sizes in the scope of code content and contributors and also have higher code complexity than *User* repositories. It is possible that *Organization* projects have more refactoring activities in their commit history than *User* projects. To further analyze refactoring activity between these two project categories, we conducted an experiment to investigate the applied refactoring activities between the *User* and *Organization* projects. First, we present the number of detected refactoring operations in each project, shown in Figure 6a. The DBM% of the two boxes is 13%, which indicates a critical difference. Thus, we can say that *Organization* projects usually have more refactoring operations than *User* projects. Next, we analyze the refactoring activities between the two groups with respect to the refactoring commit proportion and refactoring densities.



(a) # refactorings **(b)** Refactoring Commit Ratio

(c) Refs/RefCommit **(d)** Refs/Commit

Figure 6: *User* VS *Organization* in refactorings

According to Figure 6b, we can find that the *User* group has a higher ratio of refactoring commits in each project than the *Organization* group. Since the DBM% is 22%, we can say *User* projects generally have higher ratios of refactoring commits than *Organization* projects. This result is quite surprising, as we expected the *Organization* projects would contain a higher proportion of refactoring activities. One of the differences between *Organization* and *User* projects is that *Organization* projects typically have more contributors. To keep software quality and prevent defects, code review is commonly employed in software development [24], especially for those projects involved with more contributors. In this case, the program in the project should be interpretable to most of the collaborators which could significantly benefit code review in practice Bacchelli and Bird [6]. Therefore, code refactoring would play a significant role to improve code readability in software development with a large number of developers.

However, developers from *User* projects tend to spend more effort on refactoring activities rather than *Organization* projects. For refactoring density, we calculate the number of refactorings per refactoring commit for each project. From Figure 6c, the results show that developers of *Organization* projects perform more refactoring operations per refactoring commit than *User* projects with DBM% 15%. What about the number of refactorings per commit? The result is shown in Figure 6d. However, with DBM% 7% there is no difference between the two boxes. As long as *Organization* projects have more commits than *User* projects, which is proven in Figure 2a, then *Organization* generally contain more refactoring operations in the commit history.

Finding 10: *Organization* projects contain more refactoring operations and a higher refactoring density.

The implication for this finding is that recommendation tools should suggest more refactoring operations in team-based projects rather than the individual projects.

RQ5: Do major contributors apply more refactoring operations in *User* projects or *Organization* projects?
It is common sense that human factors can directly impact software development. Previous studies have assessed the relations between developer contribution behavior and software quality [9, 33]. Since the contributor number is positively correlated to refactoring number in Figure 3, we believe there exist relations between developer contribution and refactoring activity in software development. We consider three levels of expertise defined in Table 4 for each contributor committed to code files in the entire commit history of the project. The result of refactoring responsibility are shown in Figure 7 for all different refactoring types except *Change Package*. This is because *Change Package* involves multiple code files such that it is difficult to determine the responsibility based on file contribution.

In the figure, the left bars are from *User* projects and the right ones are from *Organization* projects. It is not hard to see that most of the applied refactoring operations in a file are performed by the file owner (who has the most contribution to the file), for all refactoring types. However, around 80% of refactoring operations are applied by the owner in the *User* group and only around 70% in the *Organization* group. This is because the *Organization* projects typically have more contributors and more commits than the *User* projects. Note that a file owner is also a major contributor (who contributed at least 5% of a file's commits). This seems to indicate developers tend to refactor code they are most familiar with.

Finding 11: Most detected refactoring operations applied in files are performed by the owners and major contributors especially for GitHub user repositories.

The implication for this finding is that file contribution could be considered as a feature for predicting refactoring activities.

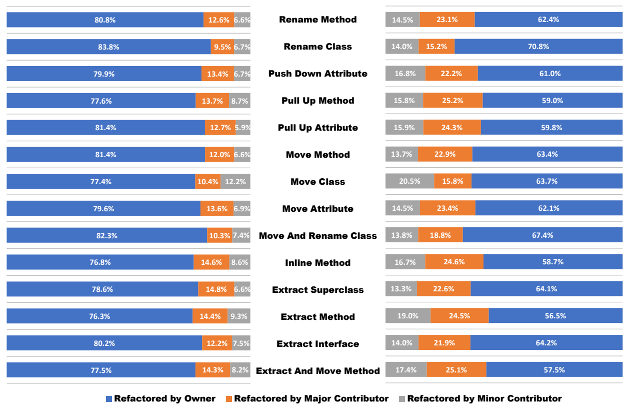


Figure 7: *User* (left) vs *Organization* (right)

4 THREATS TO VALIDITY

Here we discuss potential threats to the validity of our analysis.

Construct Validity. The most important threat to construct validity is how we assess each project's code quality. Specifically, we chose to use general project metrics (number of commits, classes, AST nodes, eg.) and C&K metrics. There are other metrics that capture software quality, for example metrics computed by means of dynamic analysis, which may yield different results. Additionally, we introduced the new notion of project *Relative Complexity* as explained in Section 2 based on these C&K metrics and our dataset.

Internal Validity. The limited number of detected refactoring types (15) and the accuracy of the detection tool used, *RMINER*, is our main internal threat. The 15 refactoring types studied in this paper do not cover all refactoring types listed by Fowler [15]. However, Murphy-Hill et al. [27] showed that most of them are commonly applied by programmers during software development. Moreover, a large scale manual validation over the 2 million refactorings was not performed in this study to assess the precision and recall of the refactoring detection tool. However, *RMINER* is reported to have high accuracy by comparing to other state-of-the-art refactoring detection tools [35, 41]. Another threat is that the refactoring dataset may be missing refactoring operations applied in the source code written in Java 9 or higher. This is because the *Boa* infrastructure currently only supports up to Java 8.

External Validity. First, the analysis was restricted to only open-source, Java-based, and Git-based repositories. The *User* and *Organization* projects are classified according to the GitHub account type. Another threat is that our refactoring dataset only contains the project whose bare Git repository size less than 160MB. This is because of the memory limitation of each process in the cluster used by the *Boa* infrastructure. However, the 11,933 projects in our refactoring dataset are recommended due to its high-quality projects in various domains [1]. These results may not generalize for other datasets, languages, and code repositories.

5 RELATED WORK

In previous studies, the refactoring community mostly focused on these areas: refactoring detection algorithms for building an accurate detection tool, the motivation behind refactoring activity in order to reason about refactoring behavior, and the relations between refactoring operations applied and software quality. In this section, we summarize the related work for investigating why developers perform refactor operations and analyzing how refactoring activity impacts software quality.

Silva et al. [35] detected refactoring operations performed by developers in a set of Java projects, and asked the refactoring performers for the motivation(s) behind such activities. Their results show that refactoring behavior is often driven by changes in new features and bug fix requests, but not for fixing code smells. Kim et al. [21] sent a survey to Microsoft engineers to find the refactoring definition in practice and identify the benefits and risks of refactoring behavior. They found that i) the motivation to push developers to perform refactoring is due to low readability, ii) the main benefit of refactoring activity is to improve readability and maintainability, and iii) the main risk for refactoring operations is due to bug introduction. Similarly, Wang [42] interviewed 10 professional software engineers to investigate motivations behind refactoring activities. They identified 12 reasons for refactoring activities to be performed by developers and classified them into two categories: intrinsic motivators and external motivators. Intrinsic motivators indicate that programmers apply refactoring operations without external rewards. For example, *Responsibility with Code Authorship* indicates developers intend to write and maintain their own code with high quality. In contrast, one external motivator is *Refactoring as Assigned Tasks* where developers apply refactoring operations because of assignments from managers.

Palomba and Zaidman [29] conducted a study to investigate the relations between test flakiness and code smells in 18 software systems. Their key finding is that refactorings applied in test smells can fix the flakiness. Cedrim et al. [10] analyzed how refactoring types impact code smells in the commit history of 23 projects. Their results show only ten percent of refactoring operations could fix code smells, while around one-third of them may actually induce new ones. Similarly, Bavota et al. [8] mined the commit histories of 3 Java software programs to investigate the relations between refactorings and metrics/code smells. They found that there is no clear relation between quality metrics and refactorings. However, our Table 7 shows that refactorings are likely applied in classes with smaller inheritance hierarchies. Compared with method-level refactorings, class-level refactoring operations are less often performed in classes with high complexity C&K metrics.

Szóke et al. [39] analyzed source code with software quality along with the commit history and tried to associate it with refactoring activities. They found that only a block of applied refactorings would increase code quality. Alshayeb [2] analyzed the refactoring activities with five properties (understandability, maintainability, adaptability, testability, and reusability). They found that refactoring activities do not always improve those properties. Moser et al. [25] also conducted a study to investigate the impact of refactoring activities in the software development of industries. Their result shows that refactoring can improve both software quality and productivity. Stroggylos and Spinellis [37] studied the impact of refactoring activities with eight quality metrics. They reported that refactoring activity does not always improve software quality. Bavota et al. [7] also proposed a study to investigate the relations between refactorings and faults. They found that the applied refactorings, involving hierarchies, could induce faults in software programs. That could be a possible reason for our finding that most refactorings are applied in classes with smaller inheritance hierarchies.

Kádár et al. [19] provided a validated code refactoring dataset of 7 open-source projects, measured source code metrics, and assessed maintainability. The differences between our work and the previous papers are that i) we provided a large-scale integrated refactoring dataset containing the complete repository data (project metadata, commit metadata, and file contents) and the refactoring descriptions detected in the whole commit history and ii) we analyzed the impacts of refactoring operations and the software quality with *Relative Complexity* in the four categories of projects.

6 CONCLUSION

It is commonly believed that refactoring can improve the design of a system and increase software maintainability. In previous studies, researchers focused on various aspects of refactoring, eg. refactoring detection algorithms, the motivation behind refactoring operation, and relations between refactoring operation and software evolution and quality. While these existing studies are based on the limited number of projects or a single category of project, in this work we proposed a large-scale refactoring study on over 11k Java open-source projects with four project categories based on two dimensions, whether they have refactorings and whether they are developed by a GitHub organization or users.

Our results demonstrate that: **1)** *Refactoring* projects have larger size than *NoRefactoring* projects (and *Organization* larger than *User*) in terms of the size metrics: the number of commits, AST nodes, contributors, and project lifetime. **2)** The detected number of refactorings in a project is strongly positively correlated to the number of commits of the project, and other size metrics. **3)** *Refactoring* projects are significantly more complex than *NoRefactoring* projects with respect to both the quantity and the ratio of highly coupled classes. **4)** The detected refactoring types are likely applied in classes with smaller inheritance hierarchies and low lack cohesion of methods. **5)** Compared with method-level refactorings, class-level refactoring operations are less often performed in classes with high complexity in C&K metrics. **6)** *Organization* projects usually contain more refactoring operations and higher refactoring density. **7)** Most detected refactoring operations in files are performed by file owners and major contributors, especially for GitHub user repositories. In the future, we intend to conduct a deeper analysis to identify the benefits of refactoring activities in the software development lifecycle and propose a predictive model for predicting refactoring operations as recommendations to developers.

REFERENCES

[1] Miltiadis Allamanis and Charles Sutton. 2013. Mining source code repositories at massive scale using language modeling. In *Proceedings of the 10th Working Conference on Mining Software Repositories*. IEEE Press, 207–216.

[2] Mohammad Alshayeb. 2009. Empirical investigation of refactoring effect on software quality. *Information and software technology* 51, 9 (2009), 1319–1326.

[3] 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*. ACM, 751–754.

[4] Giuliano Antoniol, Massimiliano Di Penta, and Ettore Merlo. 2004. An automatic approach to identify class evolution discontinuities. In *Proceedings. 7th International Workshop on Principles of Software Evolution, 2004.* IEEE, 31–40.

[5] Authors blinded for review. [n.d.]. Paper blinded for review. In *Proceedings of the blinded conference*.

[6] Alberto Bacchelli and Christian Bird. 2013. Expectations, outcomes, and challenges of modern code review. In *Proceedings of the 2013 international conference on software engineering*. IEEE Press, 712–721.

[7] 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 *2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation*. IEEE, 104–113.

[8] 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), 1–14.

[9] Christian Bird, Nachiappan Nagappan, Brendan Murphy, Harald Gall, and Premkumar Devanbu. 2011. Don't touch my code!: examining the effects of ownership on software quality. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM, 4–14.

[10] Diego Cedrim, Alessandro Garcia, Melina Mongiovi, Rohit Gheyi, Leonardo Sousa, Rafael de Mello, Baldoino Fonseca, Márcio Ribeiro, and Alexander Chávez. 2017. 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*. ACM, 465–475.

[11] Shyam R Chidamber, David P Darcy, and Chris F Kemerer. 1998. Managerial use of metrics for object-oriented software: An exploratory analysis. *IEEE Transactions on software Engineering* 24, 8 (1998), 629–639.

[12] Shyam R Chidamber and Chris F Kemerer. 1994. A metrics suite for object oriented design. *IEEE Transactions on software engineering* 20, 6 (1994), 476–493.

[13] Danny Dig, Can Comertoglu, Darko Marinov, and Ralph Johnson. 2006. Automated detection of refactorings in evolving components. In *European Conference on Object-Oriented Programming*. Springer, 404–428.

[14] Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N Nguyen. 2013. Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 422–431.

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

[16] Xi Ge, Saurabh Sarkar, and Emerson Murphy-Hill. 2014. Towards refactoring-aware code review. In *Proceedings of the 7th International Workshop on Cooperative and Human Aspects of Software Engineering*. ACM, 99–102.

[17] Rachel Harrison, Steve J Counsell, and Reuben V Nithi. 1998. An investigation into the applicability and validity of object-oriented design metrics. *Empirical Software Engineering* 3, 3 (1998), 255–273.

[18] Brian Henderson-Sellers. 1995. *Object-oriented metrics: measures of complexity*. Prentice-Hall, Inc.

[19] István Kádár, Péter Hegedus, Rudolf Ferenc, and Tibor Gyimóthy. 2016. A code refactoring dataset and its assessment regarding software maintainability. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 1. IEEE, 599–603.

[20] Miryung Kim, Dongxiang Cai, and Sunghun Kim. 2011. An empirical investigation into the role of API-level refactorings during software evolution. In *Proceedings of the 33rd International Conference on Software Engineering*. ACM, 151–160.

[21] Miryung Kim, Thomas Zimmermann, and Nachiappan Nagappan. 2014. An empirical study of refactoring challenges and benefits at Microsoft. *IEEE Transactions on Software Engineering* 40, 7 (2014), 633–649. https://doi.org/10.1109/TSE.2014.2318734

[22] Sunghun Kim, Thomas Zimmermann, Kai Pan, E James Jr, et al. 2006. Automatic identification of bug-introducing changes. In *21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*. IEEE, 81–90.

[23] Umesh L Kulkarni, YR Kalshetty, and Vrushali G Arde. 2010. Validation of ck metrics for object oriented design measurement. In *2010 3rd International Conference on Emerging Trends in Engineering and Technology*. IEEE, 646–651.

[24] Shane McIntosh, Yasutaka Kamei, Bram Adams, and Ahmed E Hassan. 2014. The impact of code review coverage and code review participation on software quality: A case study of the qt, vtk, and itk projects. In *Proceedings of the 11th Working Conference on Mining Software Repositories*. ACM, 192–201.

[25] Raimund Moser, Pekka Abrahamsson, Witold Pedrycz, Alberto Sillitti, and Giancarlo Succi. 2007. A case study on the impact of refactoring on quality and productivity in an agile team. In *IFIP Central and East European Conference on Software Engineering Techniques*. Springer, 252–266.

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

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

[28] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Denys Poshyvanyk, and Andrea De Lucia. 2014. Mining version histories for detecting code smells. *IEEE Transactions on Software Engineering* 41, 5 (2014), 462–489.

[29] Fabio Palomba and Andy Zaidman. 2017. Does refactoring of test smells induce fixing flaky tests?. In *2017 IEEE international conference on software maintenance and evolution (ICSME)*. IEEE, 1–12.

[30] Fabio Palomba, Andy Zaidman, Rocco Oliveto, and Andrea De Lucia. 2017. An exploratory study on the relationship between changes and refactoring. In *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*. IEEE, 176–185.

[31] Kyle Prete, Napol Rachatasumrit, Nikita Sudan, and Miryung Kim. 2010. Template-based reconstruction of complex refactorings. In *2010 IEEE International Conference on Software Maintenance*. IEEE, 1–10.

[32] Napol Rachatasumrit and Miryung Kim. 2012. An empirical investigation into the impact of refactoring on regression testing. In *2012 28th Ieee International Conference on Software Maintenance (Icsm)*. IEEE, 357–366.

[33] Foyzur Rahman and Premkumar Devanbu. 2011. Ownership, experience and defects: a fine-grained study of authorship. In *Proceedings of the 33rd International Conference on Software Engineering*. ACM, 491–500.

[34] Raed Shatnawi. 2015. Deriving metrics thresholds using log transformation. *Journal of Software: Evolution and Process* 27, 2 (2015), 95–113.

[35] 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.

[36] Danilo Silva and Marco Tulio Valente. 2017. RefDiff: Detecting Refactorings in Version Histories. (2017).

[37] Konstantinos Stroggylos and Diomidis Spinellis. 2007. Refactoring–Does It Improve Software Quality?. In *Fifth International Workshop on Software Quality (WoSQ'07: ICSE Workshops 2007)*. IEEE, 10–10.

[38] Ramanath Subramanyam and Mayuram S. Krishnan. 2003. Empirical analysis of ck metrics for object-oriented design complexity: Implications for software defects. *IEEE Transactions on software engineering* 29, 4 (2003), 297–310.

[39] Gábor Szóke, Gábor Antal, Csaba Nagy, Rudolf Ferenc, and Tibor Gyimóthy. 2014. Bulk fixing coding issues and its effects on software quality: Is it worth refactoring?. In *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*. IEEE, 95–104.

[40] 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.

[41] 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, 483–494.

[42] Yi Wang. 2009. What motivate software engineers to refactor source code? evidences from professional developers. In *2009 IEEE International Conference on Software Maintenance*. IEEE, 413–416.

[43] CJ Wild, Maxine Pfannkuch, Matthew Regan, and Nicholas J Horton. 2011. Towards more accessible conceptions of statistical inference. *Journal of the Royal Statistical Society: Series A (Statistics in Society)* 174, 2 (2011), 247–295.

[44] Ping Yu, Tarja Systa, and Hausi Muller. 2002. Predicting fault-proneness using OO metrics. An industrial case study. In *Proceedings of the Sixth European Conference on Software Maintenance and Reengineering*. IEEE, 99–107.