A Large-Scale Empirical Exploration on Refactoring Activities in Open Source Software Projects

Carmine Vassallo, Giovanni Grano, Fabio Palomba, Harald C. Gall, Alberto Bacchelli

University of Zurich, Switzerland
vassallo@ifi.uzh.ch, grano@ifi.uzh.ch, palomba@ifi.uzh.ch, gall@ifi.uzh.ch, bacchelli@ifi.uzh.ch

Abstract

Refactoring is a well-established practice that aims at improving the internal structure of a software system without changing its external behavior. Existing literature provides evidence of how and why developers perform refactoring in practice. In this paper, we continue on this line of research by performing a large-scale empirical analysis of refactoring practices in 200 open source systems. Specifically, we analyze the change history of these systems at commit level to investigate: (i) whether developers perform refactoring operations and, if so, which are more diffused and (ii) when refactoring operations are applied, and (iii) which are the main developer-oriented factors leading to refactoring. Based on our results, future research can focus on enabling automatic support for less frequent refactorings and on recommending refactorings based on the developer's workload, project's maturity and developer's commitment to the project.

Keywords: Refactoring, Software Evolution, Software Maintenance

1. Introduction

Refactoring is the process of improving existing code without creating new functionalities [24]. The code improvements targeted through refactoring go in two main directions [28]: on the one hand, refactoring tasks aim at increasing the *code maintainability* by reducing large monolithic routines into a set of individually concise, well-named, single-purpose methods; on the other hand, refactoring tasks aim at enhancing *code extensibility* and flexibility, by introducing recognizable design patterns. Overall, the main goal of refactoring is to reduce *design debt* [57], which is the development work accumulated when quick and low-quality solutions are used, instead of high-quality ones, to implement functionalities.

Several researchers investigated the use of refactoring. For example, Kataoka *et al.* found that refactoring does lead to code metric improvements (*e.g.*, decreasing of code coupling and code complexity [27]), Moser *et al.* found that refactoring has a positive impact on developers' productivity when properly applied [36], and Wang *et al.* studied the motivations industrial developers have when refactor, finding that developers often reduce technical debt to attract peer recognition [64].

Despite its usefulness, refactoring requires substantial effort from developers. In fact, there is a lack of proper support for refactoring and developers almost always perform refactoring operations manually [30, 37]. Moreover, even when some sophisticated automated refactoring solutions are available [33], developers do not perceive them as trustworthy and manually verify the refactorings' correctness with the fear that they may introduce defects [30]. For this reason, previous work investigated other approaches aiming at only suggesting developers to refactoring opportunities without forcing their application. Simon *et al.* [52] proposed a metric-based visualization tool indicating code components that need refactoring, while Bodhuin *et al.* [14] conceived a refactoring decision support tool based on genetic algorithms. Furthermore, Tsantalis *et al.* developed *JDeodorant*, a tool that detects certain types of code smells (*i.e.*, *Feature Envy*, *Type Checking*, *Long Method* *God Class* and *Duplicate Code*) and suggests ad-hoc refactoring strategies to remove them. However, all the proposed approaches did not specifically investigate the actual developers' needs, such as what are the refactoring operations with which developers need more help and in which circumstances developers need support.

To fill this knowledge gap and to guide future research in building approaches for supporting developers during refactoring, we performed a large-scale empirical study aimed at investigating (i) the types of refactoring developers perform more frequently, (ii) when refactoring is performed across the project's history and (iii) the developer-oriented factors leading to refactoring.

Our study takes 200 open source projects belonging to three well-known ecosystem (namely Android, Apache and Eclipse) as subjects and our results show that developers adopt refactoring operations quite differently. Specifically, (1) the support provided by the current generation of development environment (*e.g.*, IDEs) has an impact on which refactorings are performed the most, (2) the age of a software component as well as the proximity to a software release guides the planning of refactoring activities during

the overall project maintenance, and (3) the current task developers are working on and the developers' commitment to the software project are strong indicators of developers' suitability for refactoring. Based on these results, future research can focus on enabling automatic support for less frequent refactoring and on recommending refactoring based on the developer's workload and project's maturity. And last but not least, our findings paves the way to refactoring tasks triaging.

2. Methodology

The *goal* of the study is to analyze the change history of software projects, with the *purpose* of investigating (i) which kind of refactoring operations developers perform more often, (ii) when refactoring is applied, and (iii) what are the developer-oriented factors leading to refactoring operations. The *perspective* is of both researchers and practitioners, interested in broadening the knowledge on refactoring practices in real software systems and conceiving new approaches to enhance the adoption of refactoring.

2.1. Research Questions

The first goal of our study is to analyze the primary use of different types of refactoring operations, with the purpose of understanding (i) the extent to which refactoring is applied in practice and (ii) on which types of refactoring operations the research community, project managers, and tool vendors should spend more effort to support practitioners. This leads to our first research question:

> **RQ1.** *Which types of refactoring are more frequently applied in software ecosystems such as Android, Apache, and Eclipse?*

Once assessed the refactoring distribution, we conduct a fine-grained analysis on when developers perform refactoring, so that we can delineate possible trends in the adoption of specific refactoring types. This may be useful for researchers when devising refactoring approaches that are closer to the developers' habits:

> **RQ2.** *When are refactoring operations performed?*

Finally, we investigate which kind of *developer-oriented* factors (*e.g.*, as the current task and workload, as well as the knowledge of the class) may induce developers to apply refactoring. The aim is to devise new methods for *"adaptive"* refactoring, *i.e.*, tools adapting their behavior to both the current task and the developer:

> **RQ3.** *What are the main developer-oriented factors related to refactoring?*

In the following subsections we detail the design decisions taken to answer these research questions.

2.2. Context Selection

The *context* of the study consists of software systems and refactoring operations. We consider the change history of 200 projects belonging to three well-known software ecosystems (*i.e.*, Android, Apache, and Eclipse). We select these three ecosystems and their projects to obtain substantial differences in terms of (i) project size (*e.g.*, Apache and Eclipse projects are generally larger with respect to Android apps), (ii) team size (*e.g.*, Apache projects involve hundreds of developers [7], while usually Android apps are developed by small teams), and (iii) application type (*e.g.*, we analyze Android mobile apps, Apache libraries, and plug-in based Eclipse projects). Table 1 provides an overview.

All the analyzed projects are hosted in Git repositories, associated to their own issue tracker. The Android ecosystem consists of 70 apps randomly selected from the F-Droid repository.[1] The Apache dataset includes 100 Java projects randomly selected from the list of all the Apache systems available on GitHub.[2] Finally, the Eclipse ecosystem is composed by 30 Java projects selected in a random manner from the list of GitHub repositories managed by the Eclipse Foundation.[3] Overall, we mined 579,671 commits and 4,803 issues.

As for the refactoring operations, we consider the 11 types in Table 2, which belong to the catalog defined by Fowler [24] and consider operations aiming at improving the design of the code from different perspectives (*e.g.*, by extracting parts of the source code to simplify a method or by renaming classes for understandability). As explained in the next section, the choice of these refactoring operations is driven by the availability of a tool able to effectively identify them by analyzing the change history of the studied projects.

2.3. Extraction of Refactoring Operations

Given the amount of commits analyzed in this study, a manual detection of refactoring operations performed by developers would have been too expensive. In literature, two main approaches to automatically identify refactoring operations at commit level have been defined, *i.e.*, RefactoringMiner [61] and RefDiff [51]. In this study, we opt for the former because its accuracy has been shown to be higher than RefDiff [61]. More specifically, RefactoringMiner identifies all the refactoring operations considered in our study through the application of a two-step approach. It firstly uses the UMLDiff [66] algorithm to infer which classes, methods, and fields have been added/removed/modified between two subsequent commits; then, it uses a set of rules to discriminate the different types of refactoring operations. The tool was empirically measured to have precision and recall of 98%

[1] https://f-droid.org/
[2] https://git.apache.org
[3] https://github.com/eclipse

Table 1: Characteristics of the ecosystems consider in our analysis, in terms of the number of analyzed projects (#Proj.), size ranges (#Classes, KLOC), the overall number of analyzed commits (#Commits) and issues (#Issues), and the projects' age in years (Mean Age, Min-Max Age).

Ecosystem	#Proj.	#Classes	KLOC	#Commits	#Issues	Mean Age	Min-Max Age
Apache	100	4-5,052	1-1,031	207,997	3,486	6	1-15
Android	70	5-4,980	3-1,140	107,555	1,193	3	1-6
Eclipse	30	142-16,700	26-2,610	264,119	124	10	1-13
Overall	**200**	-	-	**579,671**	**4,803**	**6**	**1-15**

Table 2: The refactoring operations considered in this study, derived from the catalogue defined by Fowler [24].

Name	Description
Extract Interface	Extract a subset of a class interfaces into a single interface.
Extract Method	Extract a fragment of a method into a new method whose name explains its purpose.
Inline Method	Put the method's body into the body of its callers and remove the method.
Move Field	Create a new field in the target class, and change all its users.
Move Method	Create a new method with a similar body in the class it uses most. Either turn the old method into a simple delegation, or remove it altogether.
Pull Up Field	Move the field to the superclass.
Pull Up Method	Move the method to the superclass.
Push Down Field	Move the field to the subclasses that actually use it.
Push Down Method	Move the method to the subclasses that actually use it.
Rename Method	Replace the name of a method with a new one.
Rename Class	Replace the name of a class with a new one.

and 93%, respectively, when it turns to the detection of seeded refactorings [50], while it exhibited 98% and 87% of precision and recall, respectively, when detecting real refactoring operations applied by developers [61]. To further assess the suitability of RefactoringMiner for our study, we re-evaluate the precision of the detector[4] on a statistically significant sample composed of 380 refactoring instances identified in the subject software projects. Such a set represents a 95% statistically significant stratified sample with a 5% confidence interval of the 16,660 total refactoring operations detected by the tool. The validation was manually conducted by one of the authors of this paper, who has 5 years of experience in the refactoring research field and more than 10 years of development experience. The validity of the tool recommendations was manually assessed by analyzing the `unix diff` related to the two subsequent commits where RefactoringMiner identified a certain refactoring. As a result of this manual analysis, we found that the precision of the approach in our dataset is 95% (*i.e.*, in line with the previously reported one [50]), as such we deem the accuracy of RefactoringMiner appropriate for our study.

Overall, the tool identified 16,660 refactoring operations over the 200 analyzed systems (1,941 refactorings for Android, 10,047 for Apache, and 4,672 for Eclipse). The data extraction (*i.e.*, repositories cloning and refactoring detection at commit granularity for the 200 systems) took four weeks on two Linux platforms with two quad-core, 3.1GHz CPUs and 16 Gb of RAM.

2.4. RQ1. Distribution of Refactoring Types

To analyze how frequently each type of refactoring operations is applied, we count the number of refactoring operations per each of the 11 refactoring types considered and divide it by the overall number of refactoring operations. We do this analysis by ecosystem. In this way, we can identify the most popular refactoring activities and possibly highlight the types for which developers would benefit from a better support from the research community, project managers, and tool vendors. Section 3.1 presents the results.

2.5. RQ2. When Refactoring Is Applied

To investigate when developers perform refactoring operations, we proceed in two complementary ways.

Firstly, we measure the number of commits that are between the introduction of a class on the repository and the first refactoring operation performed on that class. This can be seen as a coarse-grained analysis aimed at investigating how many times a class is changed before a refactoring takes place. Afterwards, for each commit with a refactoring, we compute two values:

`'project-startup'`: The time between the project initiation and the commit with a detected refactoring. It is

[4]The recall cannot be evaluated because of the lack of a comprehensive oracle of refactoring operations applied on the considered projects

computed as an *ordinal* measure (values from 'one week', 'one month', 'one year', to 'more than one year' after the project's start date) and is automatically assigned by comparing the date of a commit where a refactoring has been detected with the creation date of the project (*i.e.*, the date of the first commit on the repository).[5]

'working-on-release': The time between the commit with a detected refactoring and the closest project release. It is computed as an *ordinal* measure (values from 'one day', 'one week', 'one month', to 'more than one month' before issuing of a major release, or 'one day', 'one week' after issuing of a major release).[6] To compute this tag, one of the authors manually identified the dates of the major releases by analyzing the GIT tags in the overall projects' histories.

We compute these two values to investigate *in which development phase developers are more prone to apply refactoring operations*. More specifically, we use '`project-startup`' to verify whether developers are more prone to refactor code components at the project's startup, when the source code design might not be fully defined, for example to improve the organization of the code. We use '`working-on-release`' to investigate if developers apply refactoring close to a project's deadline or prefer to be conservative and waiting for the release before applying operations that might lead to introduce faults [9].

2.6. RQ3. Developer-oriented Factors And Refactoring

To understand the reasons that lead developers to improve the design of existing code, we compute four values:

'commit-goal': It is focused on what the developer is doing when a refactoring is applied. It is a *categorical* measure that can have *one or more* values (possible values: 'bug fixing operation', 'new feature implementation', 'enhancement operation' and 'refactoring application'), used to assess if developers refactor code while performing other tasks or through dedicated refactoring sessions. To compute '`commit-goal`', we perform a two-step approach. In the first place, we try to assign the tags automatically by exploiting the issue tracker of the considered projects with the aim of automatically link issues and commits. Note that issue trackers do not only contain information about bugs, but can include issues explicitly related to other activities, (in our case, enhancements, new feature implementations, and refactoring). To this aim, we download the issues for all 200 projects from their JIRA or BUGZILLA issue trackers. Afterwards, we check whether any of the 16,660 commits detected as those containing a refactoring operation are related to any of the collected issues. In this step, we rely on two existing approaches to link commits and issues. The first, proposed by Fischer *et al.* [22], uses regular expressions to match the issue ID in the commit message. The second is a re-implementation of the approach proposed by Wu *et al.* [65], named RELINK, that works under the following constraints: (1) It matches the committer/authors with issue tracking contributor name/email; (2) the time interval between the commit and the last comment posted by the same author/contributor on the issue tracker must be less than seven days; (3) the cosine similarity (computed using the Vector Space Model (VSM) [6]) between the commit message and the last comment referred above is greater than 0.7. Wu *et al.* [65] empirically evaluate the RELINK approach, demonstrating high accuracy (precision=89% and recall=78%). With the two aforementioned approaches we look for links between of the 16,660 considered commits and the gathered issues. In the case of a link, we look at the issue type to automatically assign one of the '`commit-goal`' tags to the commit. In the end, we found links in 1,526 cases, *i.e.*, about (9.2%) of the commits, in line with previous findings [5]. As for the remaining 15,073 commits containing a refactoring operation, two of the authors of this paper (the *inspectors*) manually analyzed them with the aim of assigning one or more of the '`commit-goal`' tags. To perform this task, they relied on (i) commit message and (ii) `git diff` between the commit under analysis and the previous, and adopted a two-step grounded theory-like approach [18] described in the following:

Tuning phase: In the first step, the two inspectors independently analyzed the *same* set of 500 commits and assigned '`commit-goal`' tags based on the commit message and/or the `git diff`. This can be seen as a *tuning* phase, namely a preliminary step in which the inspectors tried to understand and find a common procedure to correctly classify commits. After completing the independent classification, they opened a discussion on the tags assigned and the overall procedure followed so far. In so doing, they discussed each of the tags assigned; in only 6 cases there was a disagreement that was then solved through discussion. For instance, in one case the commit message was misleading, as the actual changes applied and visible through the `git diff` led to a different tag. As such, the inspectors decided to mainly focus on the `git diff` to elicit '`commit-goal`' tags.

Classification phase: Once the two inspectors found a common way to assign '`commit-goal`' tags, they independently classified the remaining 14,573 commits, by analyzing 7,287 and 7,286 each. Overall, this step

[5] The '`project-startup`' value may be not properly computed in case of projects that migrated from another VCS to GIT. Indeed, in these cases only a part of the whole change history is available. To mitigate this issue, we check whether the first release tagged in the versioning system is either 0.1 or 1.0. As a result, 31 projects—having 871 commit containing refactoring—do not respect this constraint and are excluded.

[6] We consider only the major releases since they generally represent a real deadline for developers, while minor releases are a consequence of minor bug fixing.

took 90 person/hours and output the commit goals of each refactoring-related commits of the study.

***Developer status*:** This is a group of values focused on whether developer-related factors influence the decisions on the refactoring to apply.

- **'`workload`':** It measures how busy a developer is when the refactoring is applied. Such metric is computed by analyzing time frames of one month, starting from the date in which the developer performed the first commit. The number of commits performed by a developer during one month is the value of '`workload`'. It is worth noting that this metric (i) is approximated because different commits can require different amount of work and (ii) a developer could also work on other projects. However, previous work [63] found this measure to be accurate in describing the actual workload of a developer. Once computed the workload for the developer d during a month m, we compute the workload distribution for all developers involved in the project at m. Then, given Q_1 and Q_3, respectively the first and the third quartile of such distribution, we therefore compute the '`workload`' as an *ordinal* measure with possible values 'low' (if the developer performing the commit has a workload less than Q_1), 'medium' (if $Q_1 \leq workload \leq Q_3$), or 'high' (if the $workload > Q_3$).
- **'`ownership`':** If developer is the owner of a file included in a commit where a refactoring was applied, the ownership value is 'true'. This is a *binary* measure. We compute *ownership* by following the heuristic by Bird *et al.* [13]: A file owner is a developer responsible for more than 75% of the commits performed on the file.
- **'`newcomer`':** We consider whether the developer author of a refactoring was or not a newcomer. This is a *binary* measure. We assign this measure the value 'true' if the commit where the refactoring was applied is one of the first three commits performed by a developer.

For each value considered in this study we report (i) descriptive statistics of the number of commits to which a value is not null and (ii) qualitative examples of the commits having certain values.

3. Analysis of the Results

This section reports the analysis of the results used to answer our research questions.

3.1. RQ1. Distribution of Refactoring Types

Table 3 shows the distribution of refactoring operations over the analyzed ecosystems, with the absolute number and percentage of detected refactoring operations. Moreover, in Table 4 we report how the distribution of refactoring operations varies when grouping the considered projects based on their type, *i.e.*, if they are libraries providing public APIs or standard applications. This is a further analysis with which we verify if developers of libraries are more cautious when performing refactoring to avoid breaking the public API: indeed, they may be more reluctant since such changes might impact their clients.

Refactoring operations are a small fraction of the commits analyzed in the considered ecosystems: On average 1.8%, 4.8% and 1.8% of the commits mined respectively from ANDROID, APACHE and ECLIPSE contain a refactoring operation. This corroborates previous results on the limited adoption of refactoring in practice [11, 44] .

Looking more in depth, the *Rename Method* refactoring is the operation most frequently applied by developers (36.7%, 30.8% and 23.6% in ANDROID, APACHE, and ECLIPSE, respectively): intuitively, this may be due to the simplicity with which developers can apply it. Our findings suggest that developers are mainly concerned with the improvement of the understandability of source code [15] when performing refactoring, strengthening the need for techniques able to recommend consistent naming usage based on the purpose of a method [32]. A similar discussion can be done for the *Rename Class* refactoring, which aims at renaming a class to improve the understandability of its role within a software project. Also in this case, the non-negligible adoption could be due to the simplicity with which it can be applied by developers.

The results obtained for *Move Field* and *Move Method* seem to corroborate the hypothesis that the most frequent refactoring operations are those that can be easily performed. Indeed, also in this case the operation of moving a field/method from a class to another is generally doable using specific features available in most IDEs. As a side effect, this result justifies the amount of work done by the research community with respect to the identification of *Feature Envy* instances [12, 42, 43, 59], *i.e.*, methods exhibiting high coupling with a class different than the one where it is located in and that should be refactored through the application of a *Move Method* [24]. At the same time, the reported findings advocate the study of how to recommend when a field should be moved.

As for the *Extract Interface* refactoring, we find 124, 1,276, and 528 occurrences in ANDROID, APACHE, and ECLIPSE, respectively. This type of refactoring is popular in client/server applications [67], thus this may be the reason why we find a more *Extract Interface* refactoring operations when analyzing the APACHE ecosystem, which is composed of several projects using this architecture [8].

We find few occurrences of the *Extract Method* refactoring over all the considered systems. Despite practitioners perceiving the *Extract Method* as the most versatile operation to apply for improving the source code [50], we observed that developers do not frequently apply it. This might be due to (i) the intrinsic complexity of selecting the part of the method to extract and (ii) the limited support given by IDEs. For instance, consider the case of ECLIPSE: It supports the automatic extraction of methods from an existing one, yet the developer must manually select the

Table 3: **RQ1**: Distribution of the different refactoring operations across all the considered ecosystems.

Refactoring	Android		Apache		Eclipse	
	#	%	#	%	#	%
ExtractInterface	124	6.40%	1276	12.70%	528	11.30%
ExtractMethod	43	2.20%	268	2.70%	252	5.40%
InlineMethod	36	1.80%	119	1.20%	70	1.50%
MoveField	552	28.40%	2004	19.90%	844	18.10%
MoveMethod	177	9.10%	1107	11.00%	747	15.90%
PullUpField	50	2.60%	303	3.00%	245	5.20%
PullUpMethod	58	3.00%	711	7.10%	269	5.80%
PushDownField	4	0.20%	134	1.30%	153	3.30%
PushDownMethod	5	0.30%	148	1.50%	53	1.10%
RenameMethod	713	36.70%	3094	30.80%	1105	23.60%
RenameClass	179	9.20%	883	8.80%	406	8.70%
Overall	**1,941**		**10,047**		**4,672**	

Table 4: **RQ1**: Distribution of the refactoring operations across libraries providing a public API (Lib.) and standard applications (App.).

Refactoring	Android				Apache				Eclipse			
	Lib.		App.		Lib.		App.		Lib.		App.	
	#	%	#	%	#	%	#	%	#	%	#	%
ExtractInterface	0	0%	124	6.40%	630	6.30%	646	6.40%	0	0.00%	528	11.30%
ExtractMethod	0	0%	43	2.20%	132	1.30%	136	1.40%	0	0.00%	252	5.40%
InlineMethod	0	0%	36	1.80%	58	0.60%	61	0.60%	0	0.00%	70	1.50%
MoveField	0	0%	552	28.40%	992	10.00%	1012	10.10%	402	8.60%	442	9.50%
MoveMethod	0	0%	177	9.10%	588	5.90%	519	5.20%	108	2.30%	639	13.60%
PullUpField	0	0%	50	2.60%	140	1.40%	163	1.60%	0	0.00%	245	5.20%
PullUpMethod	0	0%	58	3.00%	335	3.30%	376	3.70%	0	0.00%	269	5.80%
PushDownField	0	0%	4	0.20%	60	0.60%	74	0.70%	0	0.00%	153	3.30%
PushDownMethod	0	0%	5	0.30%	70	0.70%	78	0.80%	0	0.00%	53	1.10%
RenameMethod	0	0%	713	36.70%	1583	15.80%	1511	5.00%	156	3.40%	949	20.20%
RenameClass	0	0%	179	9.20%	440	4.40%	443	5.90%	0	0.00%	406	8.70%
Overall	**0**	**0%**	**1941**	**100.00%**	**5028**	**50.10%**	**5019**	**49.90%**	**666**	**14.30%**	**4006**	**85.70%**

portion of code that will be extracted, *i.e.*, the problem of finding an optimal design solution is up to the developer. Past research provided evidence that this process is far from being trivial [60]. This calls for further research on how to automatically find refactoring opportunities to better assist developers during this type of refactoring.

The other considered refactoring operations are seldom applied to any of the analyzed systems. *Pull Up Field*, *Pull Up Method*, *Push Down Field*, and *Push Down Method* occur in less than 8% of the refactoring-related commits in all the ecosystems, while *Inline Method* in less than 2%. Thus, moving methods or field along the hierarchy is not a common practice among developers. Even less adopted is moving a method's body into the body of its callers.

When splitting projects based on their type (*i.e.*, library or application), the distribution of refactoring operations between the two categories does not variate significantly in the Apache ecosystem, meaning that the number of refactorings in libraries with public APIs and applications are basically the same. This may indicate that the developers' behavior with respect to refactoring is independent from the application type. As for Eclipse, only 8 projects out of the total 30 are libraries with a public API. Thus, the comparison is naturally balanced: this is also reflected in the number of refactoring operations performed in the two sets. Finally, in the Android ecosystem, none of the considered apps provide APIs, therefore all the refactorings fall into the 'applications' set.

Based on our findings, it seems reasonable to think that the refactoring operations more frequently applied are those automatically supported by IDEs. This result somehow provides a different perspective on the observations provided by previous work [11, 37] on the limited usage of refactoring practices. Our results call for further work to verify the hypothesis that *when developers have tools to perform a certain type of refactoring, they do it more frequently* and, if verified, to devise approaches and tools able to deeply support software developers in more

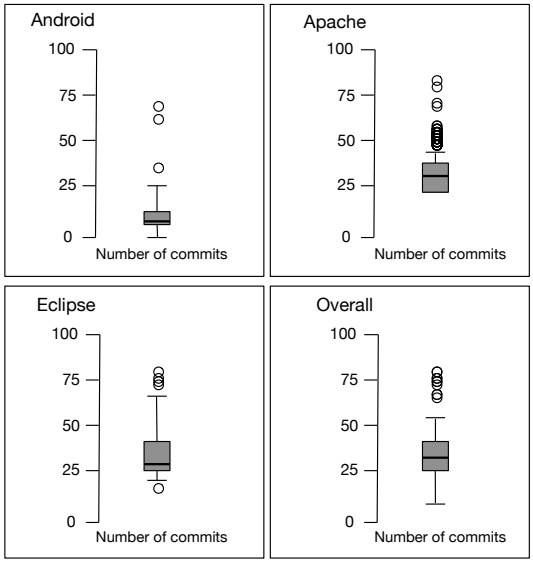
diverse refactoring operations.

Summary for RQ1: Developers do not frequently perform refactoring. When they do, the most prominent types are *Rename Method*, *Rename Class*, and *Move Field*. Other types of refactoring, *e.g.*, *Inline Method* and *Extract Method*, are less popular across the considered ecosystems.

3.2. RQ2. When Refactoring Is Applied

Figure 1 depicts the box plots of the distribution related to the number of commits done to a class before incurring in the first refactoring operation. The results are grouped by ecosystems, but we also report the overall results (*i.e.*, all the ecosystems together).

Figure 1: Number of commits to a class before the first refactoring operation.



We observe substantial differences between the ANDROID apps and the remaining ecosystems. Indeed, the median number of commits required to do refactoring on a mobile app is 8, while respectively 33 and 28 commits in APACHE and ECLIPSE. The difference between ANDROID and the other ecosystems is also statistically significant. In particular, we first run the Mann-Whitney test [20] to verify the differences between distributions; in this case, results are intended as statistically significant at α=0.05. Furthermore, we estimate the magnitude of the measured differences by using Cliff's Delta (or d), a non-parametric effect size measure [26] for ordinal data. We followed well-established guidelines to interpret the effect size values: negligible for $|d| < 0.10$, small for $|d| < 0.33$, medium for $0.33 \leq |d| < 0.474$, and large for $|d| \geq 0.474$ [26]. As a result, we find that the distribution of the number of commits required to do refactoring in ANDROID is significantly lower and with a large effect size with respect to both APACHE and ECLIPSE, while there is not statistical difference between APACHE and ECLIPSE. This result can be explained by ANDROID apps having a smaller lifecycle, than the more complex systems found in APACHE and ECLIPSE [19].

Looking at the overall results, the median value is 32 with very few outliers. This result provides a wider perspective of refactoring: It seems reasonable to think that developers may improve the existing code as a consequence of *software aging* [47] and because of the presence of some forms of technical debt [49, 55], however the relatively low number of commits required for applying a refactoring seems suggesting that developers do not necessarily restructure classes because of the presence of design issues (this result is aligned with previous studies, which reported that refactoring is mostly triggered by functional changes [50]).

The findings concerning **RQ1** can provide an explanation for these results. Indeed, the most frequent refactoring operations are concerned with the understandability and comprehensibility of the source code rather than its maintainability. As such, they can be continuously applied during the software development. This result corroborates the findings by Kim *et al.* [30], who found in readability issues the main trigger for refactoring [30].

A complementary view on when developers perform refactoring is given in Table 6, where we report the percentage of commits containing refactoring operations assigned to each category of the '`project-startup`' value.[7] The results clearly show that most of the refactoring-related commits are performed more than one year after the startup of the project. This may indicate that in the first part of the development programmers are generally busy with the implementation of core functionalities, while the need for refactoring is triggered when existing code must be maintained. This general result slightly varies for the APACHE ecosystem: Indeed we find that 19% of refactoring operations are done during the first month from the project startup. This may be the result of some APACHE projects (*e.g.*, OPEN OFFICE) explicitly suggesting developers (*e.g.*, [1]) to apply some refactoring whenever possible. From a statistical perspective, the differences between APACHE and the other ecosystems is indeed statistically significant, but with a small effect size when considering both ANDROID and ECLIPSE.

The results about '`working-on-release`' are reported in Table 5. Developers do not usually do refactoring operations close the issuing of a new release: The percentage of refactoring operations performed in the week before a new

[7] While in the previous analysis we consider only the commits where a refactoring is applied for the first time on a class, here we take into account the whole set of commits reporting a refactoring operation. In this way, we can monitor the development phases in which developers feel the need of re-structuring the source code.

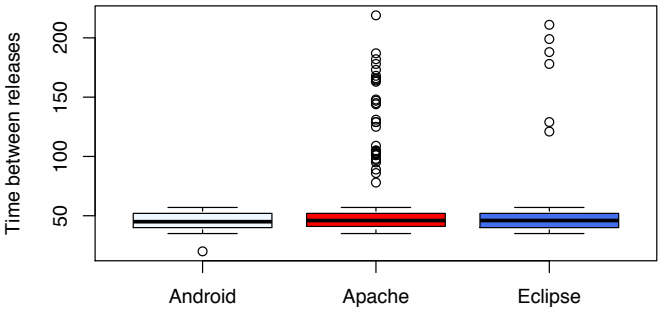
Table 5: **RQ2**: Distribution of categories in '`working-on-release`' for commits containing refactoring operations.

Ecosystem	'`working-on-release`'					
	> 1 month before	1 month before	1 week before	1 day before	1 day after	1 week after
Android	0.27	0.42	0.12	0.05	0.01	0.13
Apache	0.63	0.31	0.02	0.01	0.01	0.02
Eclipse	0.41	0.46	0.05	0.01	0.01	0.06
Overall	**0.49**	**0.42**	**0.04**	**0.01**	**0.01**	**0.03**

Table 6: **RQ2**: Distribution of categories in '`project-startup`' for commits containing refactoring operations.

Ecosystem	'`project-startup`'			
	1 Week	1 Month	1 Year	> 1 Year
Android	0.01	0.03	0.27	0.69
Apache	0.07	0.19	0.33	0.41
Eclipse	0.01	0.03	0.28	0.68
Overall	**0.01**	**0.04**	**0.27**	**0.68**

Figure 2: Distribution of the days elapsing between releases.



release is generally low (ranging between 4% and 12%). This may be explained with the results of previous research [30, 9], which reported that developers do not perceive refactoring as a behavior-preserving operation and therefore, they are concerned about the possible introduction of new faults. Finally, refactoring activities do not particularly occur in proximity of a date of a system's release. The discussion around '`working-on-release`' can be influenced by the release cycle followed by the considered projects. For example, if a project issues a release per year, it would mean that the application of refactoring operations within the last month before a release represents an action done very closely to the release. To control for this aspect, Figure 2 depicts the distribution of the number of days between releases of the projects considered, grouped by ecosystem. As shown, in all cases the median of the distribution tends to be around 50 days, meaning that most of the considered projects (185 out of the total 200) issue a new release every two months. As such, we can conclude that the discussion reported above is not biased by the specific release cycle followed by the considered projects. The discussion done so far does not change when considering libraries and standard applications independently. Indeed, we do not observe divergences with respect to the results reported at ecosystem-level; this also holds when considering the statistical point of view: the differences between libraries and standard application are not statistically significant.

Summary for RQ2: We observed statistically significant differences between ANDROID and the other ecosystems: on average, classes are subject of a refactoring operation after 8 commits from their introduction in the ANDROID projects, while they require 33 and 28 commits in APACHE and ECLIPSE, respectively. Refactoring activities are mainly done after at least one year from the project startup and developers perform few refactoring operations in the proximity of a new system release.

3.3. RQ3. Developer-oriented Factors And Refactoring

To answer **RQ3**, we analyze the percentage of refactoring-related commits classified according to the four values mentioned in Section 2.6: the '`commit-goal`' and the *developer status* ones (*i.e.*, '`workload`', '`ownership`', and '`newcomer`').

Regarding '`commit-goal`', Table 7 reports, for each refactoring operation, the distribution across the different possible categories assigned to commits containing refactoring operations. Among the three considered ecosystems, most of the refactoring operations are performed during activities involving developers improving the system, such as the enhancement of existing features. Indeed, the percentage of commits tagged as *enhancement* ranges between 54% and 100%. This finding confirms what was previously found by Murphy-Hill *et al.* [37]: Developers mainly refactor while improving the system or developing new features (aka *floss refactoring*). For instance, we observed that 37 *Move Field* and 28 *Move Method* refactoring operations were usually performed during enhancement tasks in the APACHE COCOON project. As an example, a developer applied an operation of *Move Method* refactoring when writing in the commit message to "[start]

Table 7: **RQ3**: Distribution of '`commit-goal`' categories assigned to commits containing refactoring operations. We refer as BF to Bug Fixing, as E to Enhancement, as NF to New Feature and as R to Refactoring.

Refactoring	Android				Apache				Eclipse				Overall			
	E	NF	BF	R	E	NF	BF	R	E	NF	BF	R	E	NF	BF	R
Extract Interface	.65	.21	.09	.05	.96	.01	.02	.00	.38	.16	.37	.09	.79	.06	.12	.03
Extract Method	.79	.12	.07	.02	.96	.03	.01	.01	.41	.20	.38	.01	.70	.11	.18	.01
Inline Method	.58	.22	.08	.11	.96	.03	.00	.01	.54	.10	.27	.09	.77	.08	.10	.05
Move Field	.54	.07	.15	.24	.95	.02	.01	.01	.29	.15	.45	.12	.72	.06	.14	.08
Move Method	.53	.05	.16	.27	.88	.03	.07	.02	.54	.08	.31	.06	.73	.05	.17	.06
Pull Up Field	.64	.20	.14	.02	.96	.00	.01	.03	.53	.13	.27	.07	.76	.07	.13	.04
Pull Up Method	.57	.07	.24	.12	.94	.01	.00	.05	.39	.09	.36	.17	.77	.03	.11	.09
Push Down Field	1.00	.00	.00	.00	1.00	.00	.00	.00	.49	.20	.28	.03	.73	.11	.15	.01
Push Down Method	1.00	.00	.00	.00	.92	.07	.01	.00	.53	.13	.23	.11	.82	.09	.06	.03
Rename Method	.63	.16	.16	.50	.94	.03	.02	.01	.62	.12	.24	.02	.82	.07	.09	.02
Rename Class	.65	.10	.11	.14	.92	.01	.07	.01	.35	.10	.50	.05	.73	.04	.19	.03

Table 8: **RQ3**: Distribution of values in the *developer status* group to commits containing refactoring operations.

Ecosystem	'workload'			'ownership'		'newcomer'	
	High	Medium	Low	True	False	True	False
Android	0.28	0.52	0.20	0.84	0.16	0.15	0.85
Apache	0.11	0.64	0.25	0.90	0.10	0.07	0.93
Eclipse	0.05	0.69	0.26	0.85	0.15	0.09	0.91
Overall	**0.13**	**0.64**	**0.23**	**0.86**	**0.14**	**0.08**	**0.92**

the implementation of a new configuration option from the endpoint which is standard way in Camel" in the class `RabbitMQProducer`.

In the Eclipse ecosystem we observed a consistent percentage of refactoring actions performed while fixing bugs; in this case, the differences with the other ecosystems are statistically significant and have a medium effect size when compared to Android and a large one when considering Apache. A similar result had been already noticed by Palomba *et al.* [46], who showed that different refactoring activities aimed at improving both maintainability and comprehensibility of the source code are generally applied during bug fixing tasks. Such a scenario is particularly evident if we look the case of the *Extract Method* refactoring for Eclipse projects, where the commits tagged as *bug fixing* reach 38%. An interesting case regards the method `testStepIntoBinaryMethod` of the class `StepIntoSelectionTests` contained in the Eclipse Debug UI project. During a debugging session, in order to fix a bug causing the destruction of the error reports, that method was decomposed thought an *Extract Method* refactoring in two separate ones. Indeed, the developer reported the following commit message: "Fixed bug 413434: StepIntoSelectionTests destroys error reporting."

The second interesting aspect to consider is related to the *refactoring* tag (see '`commit-goal`' tags in 2.6). On the Android ecosystem, we found several cases in which refactoring operations were the main goal of the commit. As an example, a developer of the Frost Wire app refactored its code by applying a *Push Down Method* operation to "abstract new BittorrentIntent. Code must be beautiful on the inside too" (as explained in the commit message). Even if the percentage of commits tagged as *refactoring operations* is statistically lower considering the Apache (α <0.05, Cliff's d=medium) and Eclipse (α <0.05, Cliff's d=large) ecosystems, also in these cases we found situations in which developers were aware of the design problems of the source code and actually performed program transformations aimed at improving it. This is, for example, the case of the class `FtpConsumer` of the Apache Camel project, where the developer reports in the commit message to have "refactored camel-ftp so the producer and consumer share more code between the FTP and SFTP parts in their common super class."

Finally, Table 8 reports the distribution of commits across the values of *developer status*. From the analysis of the results, the developer's '`workload`' emerges has a key feature when we consider the application of refactoring operations. Indeed, we find that developers with a *medium* workload are more prone to apply refactoring operations, while developers having a *high* one tend to refactor less. On the overall dataset, at least in 52% of cases the developer who refactors the code has medium workload.

Developers who apply refactoring operations are usually not newcomers, meaning that the improvement of source code is done by more experienced and knowledgeable developers. Perhaps more important, developers that apply refactoring operations are almost always owners of the refactored files, as visible in the values Table 8 under the '`newcomer`' and '`ownership`' columns. This may suggest that, in most cases, even developers who have a good knowledge of a file (being the owner) prefer to refactor the code to increase its maintainability. As for the previous research questions, the discussion is similar when considering libraries and standard applications independently.

Summary for RQ3: Developers apply refactoring mainly when enhancing existing features, yet we found several cases in which refactoring operations are made during bug fixing. Developers that refactor code are almost always the owners of the refactored file and they tend to improve the design when their workload is less than medium.

4. Discussion & Implications

After presenting the results of the study, in this section we focus on discussing the main results of our study and what are the key implications they have for researchers and practitioners.

4.1. Discussion

Our results let emerge some topics worth further discussion:

Developers do not perform the refactoring types considered in this study. The first clear result of the study concerns the low frequency of application of the refactoring operations considered in the context of this study. While this is in line with previous findings in the field [11], we provided additional evidence that this is true independently from the characteristics of the system: the lack of refactoring was observed among all the considered ecosystems.

Developers mainly perform tool-supported refactoring. Based on our findings, the most frequently applied refactoring operations, *i.e.*, *Rename Method*, *Rename Class*, and *Move Field*, are those that can be more easily performed either manually or using the features made available by existing IDEs. Thus, it seems that if developers have an automatic support, refactoring is more likely to be applied.

Code improvement is not done before a release. During the days leading to the issue of a new software release, developers tend to avoid the application of refactoring. This is likely due to the fear of introducing defects that cannot be repaired anymore [9]. Interestingly, however, is that exactly the time pressure before a release leads developers to introduce more design flaws [63, 62].

Developers do floss refactoring. As previously shown by Murphy-Hill *et al.* [37], developers rarely perform refactoring operations during dedicated sessions, but rather they generally perform *floss refactoring*, *i.e.*, the interleave enhancements to the source code with its improvement.

Ownership and refactoring. Developers performing refactoring are almost always the owner of the refactored files, likely indicating that it requires enough knowledge of the code to be successfully applied.

4.2. Implications

The aforementioned findings have implications for different stakeholders, such as researchers, tool vendors and project managers:

1. **Enabling automatic support.** Several refactoring types are not supported by the IDEs and, therefore, their application is more difficult for developers. This represents a call for new approaches and tools enabling the management of a wider set of and more complex refactoring operations.

2. **Adaptive refactoring.** Our results provide insights on the circumstances leading to the application of certain refactorings. Specifically, the current development task (*e.g.*, bug fixing, new feature implementation, feature enhancement) the developers' workload and the knowledge developers have about certain software components seem to be related to (i) the decision to refactor or not and (ii) which kind of refactoring operations to perform. These findings may represent an important starting point for researchers interested in devising new methodologies for *adapting* refactoring recommendations not only to the change type [46] but also to the characteristics of developers and the development time-frame of a project.

3. **Triaging of refactoring activities.** Developers performing refactoring are almost never newcomers and, more importantly, they are owners of the refactored files. Likely, this is due respectively to the difficulty to refactor long existing systems and the knowledge of the file logic required to improve the source code. For this reason, the decisions of *who* should be in charge of such operations must account not only for the technical expertise of a developer, but also for the knowledge she has of the overall system.

4. **Understanding the consequences of missing refactoring.** Close to releases, developers introduce a number of design flaws [63] but not perform refactoring. Given the harmfulness of having code smells in source code for change- and fault-proneness [29, 21, 41], an important need for the research community is that to find ways to make developers aware of the possible consequences of poor applications of refactoring.

5. Threats to Validity

In this section we discuss possible threats to the validity of our study and how we mitigated them.

5.1. Threats to Construct Validity

A first threat influencing the relationship between theory and observation is concerned with the dataset exploited to conduct the study. To identify the refactoring operations between two subsequent commits, we rely on the RefactoringMiner tool devised by Silva *et al.* [50]. While the empirical evaluation conducted by the authors of the tool already showed a precision of 98% and a recall of 93%, we ensure its suitability for our study by re-evaluating the precision of the tool on a statistically significant sample composed of 380 refactoring operations it detected. We find a precision of 95%, thus confirming that this tool is actually useful for our purposes. We are aware that the recall might diverge for different projects not involved in the original validation that we take into account in our study, however we cannot evaluate it because of the lack of a golden set reporting the actual refactoring actions over all the history of the considered projects. Moreover, every detecting approach is sensitive to particular situations: commits that mix refactoring operations with other changes and sequence of refactoring that are applied sequentially on the same piece of code and then committed.

To compute values for '`project-startup`', we compare the date of the commit including a refactoring operation with the creation date of the project, identified as the date of the first commit. However, this assignment might lead to incorrect creation dates in the case of project migrated to Git from another VCS. In such cases, part of the history is not available and therefore, going back the exact creation date is not possible. To mitigate this threat, we check the tag of the first release in the versioning system. Whether the tag is 0.1 or 1.0, we consider the project as created originally in Git. Nevertheless, we are aware that this is still an approximation, since a project starting with 1.0 might have a previous 0.x history.

As for the *working on release* tag, we identify the date in which a release has been issued by considering the tags assigned by developers in Git. Furthermore, we consider the major revisions only, since they represent the main deadlines for developers. We cannot exclude possible imprecisions of the tags contained in Git.

To compute values for the '`commit-goal`' we link issues to refactoring commits relying on the ReLink [65] tool; it shows about 89% and 78% of precision and recall, respectively. We argue that such approach is precise enough to not influence the validity of our results. To deal with the missing links, we manually validated about 27,000 cases looking at the commit message and at the `diff` between such a commit and the previous one.

5.2. Threats to Internal Validity

In the context of **RQ1** and **RQ2** we studied tags related to different aspects of the development of a software project, able to characterize commits, developers, and project status. We are aware that there might be other factors that could have influenced the decisions of developers to refactor or not a certain class. In any case, it is beyond the scope of this paper to make claims about the causation between the factors considered and the application of refactoring operations.

5.3. Threats to External Validity

Our study is restricted to open source Java project hosted on GitHub. Therefore, we cannot ensure the generalization of our results to industrial systems or to systems implemented in other languages. However, to deal with such a threat, we conduct our study on a large dataset of 200 projects in total, coming from different application domains.

6. Related Work

We provide an overview of the literature on the empirical studies on refactoring, followed by a brief summary of (semi-)automatic refactoring approaches (on the latter, a more complete overview is available in the surveys by Mens *et al.* [34] and Bavota *et al.* [10]).

6.1. Empirical Studies on Refactoring

Several studies have investigated the motivations behind refactoring operations. Silva *et al.* [50] monitored 748 Java projects for 61 days, asking developers the reasons behind refactoring operations these developers had recently performed. The researchers found that refactoring is often driven by changes rather than by the necessity to fix code smells. Similarly, Wang *et al.* [64] interviewed ten industrial developers to investigate the major reasons motivating refactoring activities. They report 12 different factors classifiable in *intrinsic motivators* and *external motivators*. The former are those that do not lead to external rewards (*e.g.*, an intrinsic motivator is present when developers want to ensure high quality for the code they authored), while the latter are motivated by external rewards, such as the recognition from other developers.

Murphy-Hill *et al.* [37] analyzed 8 different datasets trying to understand how developers perform refactoring. They discovered that developers: (i) perform at least one refactoring session in more than 40% of development activities, (ii) rarely (less than 10% of times) configure refactoring tools, (iii) seldom report their refactoring activities in commit messages, (iv) often perform *floss refactoring* (*i.e.*, interleaving refactoring with other programming activities), and (v) manually perform most of the refactoring operations (close to 90%) without the help of any tool.

The results on floss refactoring were confirmed later by Negara *et al.* [38]; In particular, they performed a comparative study of manual and automated refactoring operations, finding that more than half of the manually performed ones are clustered in time. Moreover, they also investigated the most popular refactoring types, reporting differences between those that are more frequently done

manually (*e.g.*, *Rename Field*) and automatically (*e.g.*, *Extract Local Variable*). Our results confirm some of the findings reported by Murphy-Hill *et al.* [37] and Negara *et al.* [38], such as the scheduling of refactoring during other tasks *e.g.*, enhancing existing features, or the popularity of renaming-related refactoring operations. At the same time, we also provide new crucial insights on the factors that lead developers to apply certain type of refactoring.

Kim *et al.* [30] presented a survey performed with 328 Microsoft engineers investigating (i) when and how developers refactor code, (ii) if they use automated refactoring tools, and (iii) the perception of developers on the benefits, risks, and challenges of refactoring. The main findings from the survey include that: (i) refactoring is not perceived as a behavior-preserving operation, (ii) half of the developers report to manually perform it, (iii) refactoring is spurred by low source code readability, (iv) the main perceived benefits are improved readability and maintainability, and (v) there is an high perceived risk of introducing bugs while refactoring. Kim *et al.* also reported a quantitative analysis performed on the Windows 7 change history showing that code components refactored over time have fewer inter-module dependencies and post-release defects than other modules. Similar results have been reported by Kataoka *et al.* [27] and Gatrell & Counsell [25]. We continue on this line of research by investigating at which development stages developers plan for refactoring and the characteristics that developers usually exhibit while refactoring certain software components.

Palomba *et al.* [46] conducted an empirical study aimed at understanding the relationship between different kind of code changes and various refactoring operations. They found that a higher number of refactoring occurs during bug fixing, while more complex operations aim at improving code cohesion. Differently from this study, we consider not only the undergoing task while refactoring. Indeed, in additional to a considerably larger dataset, we take into account both the system history and developer-oriented metrics.

Previous work also studied the relationship between refactoring and software quality. Bavota *et al.* [9] conducted a study aimed at investigating to what extent refactoring activities induce faults. They found a positive correlation between refactoring involving hierarchies (e.g., *pull down method*) and the number of faults. Conversely, other kinds of refactoring were found to be less likely harmful in practice. In our study, we use the same dataset of refactoring operations. Bavota *et al.* also conducted a study aimed at understanding the relationships between code quality and refactoring [11]. In particular, they analyzed the evolution of 63 releases of three open source software systems to investigate what makes code components more or less prone to being object of refactoring operations. Results indicate that often refactoring is not performed on classes having a low metric profile, while almost 40% of the times refactoring was performed on classes affected by smells. We complement this study by investigating the impact of classes' age (*i.e.*, number of commits since the starting of the project) on their refactoring. Similarly, Cedrim *et al.* [17] analyzed how commonly-used refactoring affect the density of code smells. They discovered that only 9.7% of the refactoring operations removed code smells. Moreover, they observed that typical refactoring operations induce the surfacing of code smells, such as *Move Method* and *Pull Up Method* with respect to the *God Class* smell. Palomba & Zaidman [45] analyzed the correlation between code smells and *flakiness* in unit test cases. One of the main outcome of their empirical investigation is that refactoring is an effective flaky test fixing strategy, *i.e.*, refactoring might turn to deterministic the output of otherwise non-deterministic tests. Stroggylos and Spinellis [53] studied the impact of refactoring operations on the values of eight object-oriented quality metrics. Their results show the possible negative effects that refactoring can have on some quality metrics (*e.g.*, increased value of the LCOM metric). On the same line, Stroullia & Kapoor [54] analyzed the evolution of one system observing a decreasing of LOC and NOM (Number of Method) metrics on the classes in which a refactoring was applied. Szoke *et al.* [56] performed a study on five software systems to investigate the relationship between refactoring and code quality. They found that small refactoring operations performed in isolation rarely impact software quality. On the other side, a high number of refactoring operations performed in block helps in substantially improving code quality. Alshayeb [2] investigated the impact of refactoring operations on five quality attributes, namely adaptability, maintainability, understandability, reusability, and testability. Their findings highlight that the benefits brought by refactoring operations on some artifacts are often counterbalanced by a decrease of quality in some other artifacts. Compared to these work, we differ our focus looking at the circumstances leading to refactoring and not at its consequences.

Finally, Moser *et al.* [36] conducted a case study in an industrial environment aimed at investigating the impact of refactoring on the productivity of an agile team. They provided evidence that refactoring is a factor boosting the productivity itself. Ammerkaan *et al.* [3] investigated the effect of refactoring in legacy industrial systems. Counterintuitively, they observed cases of decrease in understandability, resulting in a productivity penalty due the introduction of a different style not familiar to developers. In our study, we do not analyze the impact refactoring has on the developers' productivity but we investigated the circumstances making usually developers willing to perform refactoring.

6.2. (Semi-)automated Refactoring Approaches

In addition to conducting empirical studies on refactoring, researchers have proposed strategy to (semi-)automatically support refactoring operations. O'Keeffe & O'Cinneide [39] proposed the idea of refactoring as a search problem in the space of alternative designs. Maruyama &

Shima [33] introduced a mechanism for automating refactoring of methods in Object-Oriented frameworks to improve the reusability of frameworks relying on weighted dependence graphs. Atkinson & King [4] presented a low-cost, syntactic approach for automatically discovering opportunities for refactoring the source code using symbol table and reference information. Casais [16] proposed several algorithms to restructure class hierarchies to maximize abstraction, while Moore [35] proposed a method in which the existing classes having low quality are replaced with a new set of classes where their methods are optimally factored aiming at minimizing code duplication.

Opdyke [40] developed the first tool providing semi-automatic refactoring support, which was implemented in the Refactoring Browser presented in [48]. Simon *et al.* [52] devised a metric-based visualization tool to support the software engineer during the identification of code components that need refactoring. Bodhuin *et al.* [14] introduced *SORMASA*, a refactoring decision support tool based on Genetic Algorithms. Tsantalis *et al.* [59] presented *JDeodorant*, a tool able to detect the Feature Envy code smell with the aim of suggesting move method refactoring opportunities. In its current version, JDeodorant can refactor code to remove four additional code smells [58, 60, 31, 23].

7. Conclusion

In this paper, we conduct a large-scale empirical study on the refactoring activities performed by the developers of 200 Apache, Eclipse, and Android systems. The contributions made by this work are:

1. An analysis of the distribution of refactoring operations, which revealed how developers do not refactor source code frequently and, when they do, privilege the application of refactoring actions that improve the understandability of source code.

2. An analysis of the time windows in which refactoring is generally applied in real software systems. We show that refactoring is mostly applied when a deadline is far and only after the initial creation of the structure of the system.

3. A quantitative analysis of the circumstances in which developers refactor source code. We reveal that they do perform focused refactoring activities in very few cases: while they tend to improve the understandability and maintainability of source code and when enhancing existing features. Furthermore, refactoring is mostly applied by expert developers that are owner of the refactored files.

Our findings highlight a number of challenges that should be carefully taken into account by the research community in order to define a new generation of refactoring tools that better fit the developers' needs. This represents the main input for our future research agenda, mainly oriented to the definition of novel refactoring techniques.

References

[1] 2010. Apache open office refactoring guidelines.

[2] Alshayeb, M. 2009. Empirical investigation of refactoring effect on software quality. *Information and Software Technology 51,* 9, 1319 – 1326.

[3] Ammerlaan, E., Veninga, W., and Zaidman, A. 2015. Old habits die hard: Why refactoring for understandability does not give immediate benefits. In *SANER*. IEEE Computer Society, 504–507.

[4] Atkinson, D. C. and King, T. 2005. Lightweight detection of program refactorings. In *Proceedings of the 12th Asia-Pacific Software Engineering Conference*. IEEE CS Press, Taipei, Taiwan, 663–670.

[5] Bachmann, A., Bird, C., Rahman, F., Devanbu, P. T., and Bernstein, A. 2010. The missing links: bugs and bug-fix commits. In *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2010, Santa Fe, NM, USA, November 7-11, 2010*. ACM, 97–106.

[6] Baeza-Yates, R. and Ribeiro-Neto, B. 1999. *Modern Information Retrieval*. Addison-Wesley.

[7] Bavota, G., Canfora, G., Di Penta, M., Oliveto, R., and Panichella, S. 2013. The evolution of project inter-dependencies in a software ecosystem: The case of Apache. In *2013 IEEE International Conference on Software Maintenance, Eindhoven, The Netherlands, September 22-28, 2013*. 280–289.

[8] Bavota, G., Canfora, G., Di Penta, M., Oliveto, R., and Panichella, S. 2013. The evolution of project inter-dependencies in a software ecosystem: The case of apache. In *ICSM*. 280–289.

[9] Bavota, G., De Carluccio, B., De Lucia, A., Di Penta, M., Oliveto, R., and Strollo, O. 2012. When does a refactoring induce bugs? an empirical study. In *Proceedings of the 12th International Working Conference on Source Code Analysis and Manipulation*. SCAM '12. 104–113.

[10] Bavota, G., De Lucia, A., Marcus, A., and Oliveto, R. 2014. Recommending refactoring operations in large software systems. In *Recommendation Systems in Software Engineering*, M. P. Robillard, W. Maalej, R. J. Walker, and T. Zimmermann, Eds. Springer Berlin Heidelberg, 387–419.

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

[12] Bavota, G., Oliveto, R., Gethers, M., Poshyvanyk, D., and Lucia, A. D. 2014. Methodbook: Recommending move method refactorings via relational topic models. *IEEE Trans. Software Eng. 40,* 7, 671–694.

[13] Bird, C., Nagappan, N., Murphy, B., Gall, H., and Devanbu, P. T. 2011. Don't touch my code!: examining the effects of ownership on software quality. In *SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering and 13rd European Software Engineering Conference, Szeged, Hungary, September 5-9, 2011*. ACM, 4–14.

[14] Bodhuin, T., Canfora, G., and Troiano, L. 2007. SORMASA: A tool for suggesting model refactoring actions by metrics-led genetic algorithm. In *Proceedings of 1st Workshop on Refactoring Tools*. Berlin, Germany, 23–24.

[15] Buse, R. P. L. and Weimer, W. 2010. Learning a metric for code readability. *IEEE Trans. Software Eng. 36,* 4, 546–558.

[16] Casais, E. 1992. An incremental class reorganization approach. In *Proceedings of the 6th European Conference on Object-Oriented Programming*. Utrecht, the Netherlands, 114–132.

[17] Cedrim, D., Garcia, A., Mongiovi, M., Gheyi, R., Sousa, L., de Mello, R., Fonseca, B., Ribeiro, M., and Chávez, A. 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. ESEC/FSE 2017. ACM, New York, NY, USA, 465–475.

[18] Charmaz, K. and Belgrave, L. L. 2007. Grounded theory. *The Blackwell encyclopedia of sociology*.

[19] Collins, C., Galpin, M., and Käppler, M. 2011. *Android in practice*. Manning Publications Co.

[20] Conover, W. J. 1998. *Practical Nonparametric Statistics* 3rd Edition Ed. Wiley.

[21] D'Ambros, M., Bacchelli, A., and Lanza, M. 2010. On the impact of design flaws on software defects. In *Quality Software (QSIC), 2010 10th International Conference on*. IEEE, 23–31.

[22] Fischer, M., Pinzger, M., and Gall, H. 2003. Populating a release history database from version control and bug tracking systems. In *19th International Conference on Software Maintenance (ICSM 2003), The Architecture of Existing Systems, 22-26 September 2003, Amsterdam, The Netherlands*. IEEE Computer Society, 23–.

[23] Fokaefs, M., Tsantalis, N., Stroulia, E., and Chatzigeorgiou, A. 2012. Identification and application of extract class refactorings in object-oriented systems. *J. Syst. Softw. 85,* 10, 2241–2260.

[24] Fowler, M., Beck, K., Brant, J., Opdyke, W., and Roberts, D. 1999. *Refactoring: Improving the Design of Existing Code*. Addison-Wiley.

[25] Gatrell, M. and Counsell, S. 2015. The effect of refactoring on change and fault-proneness in commercial c# software. *Sci. Comput. Program. 102,* C, 44–56.

[26] Grissom, R. J. and Kim, J. J. 2005. *Effect sizes for research: A broad practical approach* 2nd Edition Ed. Lawrence Earlbaum Associates.

[27] Kataoka, Y., Imai, T., Andou, H., and Fukaya, T. 2002. A quantitative evaluation of maintainability enhancement by refactoring. In *Software Maintenance, 2002. Proceedings. International Conference on*. 576 – 585.

[28] Kerievsky, J. 2004. *Refactoring to patterns*. Addison Wesley.

[29] Khomh, F., Penta, M. D., Guéhéneuc, Y.-G., and Antoniol, G. 2012. An exploratory study of the impact of antipatterns on class change- and fault-proneness. *Empirical Software Engineering 17,* 3, 243–275.

[30] Kim, M., Zimmermann, T., and Nagappan, N. 2014. An empirical study of refactoringchallenges and benefits at microsoft. *Software Engineering, IEEE Transactions on 40,* 7, 633–649.

[31] Ligu, E., Chatzigeorgiou, A., Chaikalis, T., and Ygeionomakis, N. 2013. Identification of refused bequest code smells. In *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*. 392–395.

[32] Lin, B., Scalabrino, S., Mocci, A., Oliveto, R., Bavota, G., and Lanza, M. 2017. Investigating the use of code analysis and nlp to promote a consistent usage of identifiers. In *Source Code Analysis and Manipulation (SCAM), 2017 IEEE 17th International Working Conference on*. IEEE, 81–90.

[33] Maruyama, K. and Shima, K. 1999. Automatic method refactoring using weighted dependence graphs. In *Proceedings of 21st International Conference on Software Engineering*. ACM Press, Los Alamitos, California, USA, 236–245.

[34] Mens, T. and Tourwé, T. 2004. A survey of software refactoring. *IEEE Transactions on Software Engineering 30,* 2, 126–139.

[35] Moore, I. 1996. Automatic inheritance hierarchy restructuring and method refactoring. In *Proceedings of 11th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM Press, San Jose, California, USA, 235–250.

[36] Moser, R., Abrahamsson, P., Pedrycz, W., Sillitti, A., and Succi, G. 2008. Balancing agility and formalism in software engineering. Springer-Verlag, Berlin, Heidelberg, Chapter A Case Study on the Impact of Refactoring on Quality and Productivity in an Agile Team, 252–266.

[37] Murphy-Hill, E., Parnin, C., and Black, A. P. 2011. How we refactor, and how we know it. *Transactions on Software Engineering 38,* 1, 5–18.

[38] Negara, S., Chen, N., Vakilian, M., Johnson, R. E., and Dig, D. 2013. A comparative study of manual and automated refactorings. In *European Conference on Object-Oriented Programming*. Springer, 552–576.

[39] O'Keeffe, M. and O'Cinneide, M. 2006. Search-based software maintenance. In *Proceedings of 10th European Conference on Software Maintenance and Reengineering*. IEEE CS Press, Bari, Italy, 249–260.

[40] Opdyke, W. F. 1992. Refactoring object-oriented framework. Ph.D. thesis, University of Illinois.

[41] Palomba, F., Bavota, G., Di Penta, M., Fasano, F., Oliveto, R., and De Lucia, A. 2017. On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation. *Empirical Software Engineering*, 1–34.

[42] Palomba, F., Bavota, G., Di Penta, M., Oliveto, R., Poshyvanyk, D., and De Lucia, A. 2015. Mining version histories for detecting code smells. *IEEE Transactions on Software Engineering*.

[43] Palomba, F., Panichella, A., De Lucia, A., Oliveto, R., and Zaidman, A. 2016. A textual-based technique for smell detection. In *Program Comprehension (ICPC), 2016 IEEE 24th International Conference on*. IEEE, 1–10.

[44] Palomba, F., Panichella, A., Zaidman, A., Oliveto, R., and De Lucia, A. 2017. The scent of a smell: An extensive comparison between textual and structural smells. *IEEE Transactions on Software Engineering*.

[45] Palomba, F. and Zaidman, A. 2017. Does refactoring of test smells induce fixing flaky tests? In *ICSME*. IEEE Computer Society, 1–12.

[46] Palomba, F., Zaidman, A., Oliveto, R., and De Lucia, A. 2017. An exploratory study on the relationship between changes and refactoring. In *Proceedings of the 25th International Conference on Program Comprehension*. ICPC '17. IEEE Press, Piscataway, NJ, USA, 176–185.

[47] Parnas, D. L. 1994. Software aging. 279–287.

[48] Roberts, D., Brant, J., and Johnson, R. 1997. A refactoring tool for smalltalk. *Theory and Practice of Object Systems 3,* 4, 253–263.

[49] Seaman, C., Guo, Y., Izurieta, C., Cai, Y., Zazworka, N., Shull, F., and Vetrò, A. 2012. Using technical debt data in decision making: Potential decision approaches. In *Proceedings of the Third International Workshop on Managing Technical Debt*. IEEE Press, 45–48.

[50] Silva, D., Tsantalis, N., and Valente, M. T. 2016. Why we refactor? confessions of github contributors. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE 2016. ACM, New York, NY, USA, 858–870.

[51] Silva, D. and Valente, M. T. 2017. Refdiff: detecting refactorings in version histories. In *Proceedings of the 14th International Conference on Mining Software Repositories*. IEEE Press, 269–279.

[52] Simon, F., Steinbrückner, F., and Lewerentz, C. 2001. Metrics based refactoring. In *Proceedings of the 5th European Conference on Software Maintenance and Reengineering*. IEEE CS Press, Lisbon, Portugal, 30–38.

[53] Stroggylos, K. and Spinellis, D. 2007. Refactoring–does it improve software quality? In *Proceedings of the 5th International Workshop on Software Quality*. WoSQ '07. IEEE Computer Society, Washington, DC, USA, 10–.

[54] Stroulia, E. and Kapoor, R. 2001. Metrics of refactoring-based development: An experience report. In *OOIS 2001*, X. Wang, R. Johnston, and S. Patel, Eds. Springer London, 113–122.

[55] Suryanarayana, G., Samarthyam, G., and Sharma, T. 2014. *Refactoring for software design smells: Managing technical debt*. Morgan Kaufmann.

[56] Szoke, G., Antal, G., Nagy, C., Ferenc, R., and Gyimóthy, T. 2014. Bulk fixing coding issues and its effects on software quality: Is it worth refactoring? In *Source Code Analysis and Manipulation (SCAM), 2014 IEEE 14th International Working Conference on*. IEEE, 95–104.

[57] Tom, E., Aurum, A., and Vidgen, R. 2013. An exploration of technical debt. *Journal of Systems and Software 86,* 6, 1498 – 1516.

[58] Tsantalis, N., Chaikalis, T., and Chatzigeorgiou, A. 2008. Jdeodorant: Identification and removal of type-checking bad smells. In *Software Maintenance and Reengineering, 2008. CSMR 2008. 12th European Conference on*. 329–331.

[59] Tsantalis, N. and Chatzigeorgiou, A. 2009. Identification of move method refactoring opportunities. *IEEE Transactions on Software Engineering 35,* 3, 347–367.

[60] Tsantalis, N. and Chatzigeorgiou, A. 2011. Identification of extract method refactoring opportunities for the decomposition of methods. *J. Syst. Softw. 84,* 10, 1757–1782.

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

[62] Tufano, M., Palomba, F., Bavota, G., Di Penta, M., Oliveto, R., De Lucia, A., and Poshyvanyk, D. 2016. An empirical investigation into the nature of test smells. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM, 4–15.

[63] Tufano, M., Palomba, F., Bavota, G., Oliveto, R., Di Penta, M., De Lucia, A., and Poshyvanyk, D. 2017. When and why your code starts to smell bad (and whether the smells go away). *IEEE Transactions on Software Engineering*.

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

[65] Wu, R., Zhang, H., Kim, S., and Cheung, S.-C. 2011. ReLink: recovering links between bugs and changes. In *SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13rd European Software Engineering Conference (ESEC-13), Szeged, Hungary, September 5-9, 2011*. ACM, 15–25.

[66] Xing, Z. and Stroulia, E. 2005. Umldiff: an algorithm for object-oriented design differencing. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*. ACM, 54–65.

[67] Xing, Z. and Stroulia, E. 2006. Refactoring practice: How it is and how it should be supported-an eclipse case study. In *Software Maintenance, 2006. ICSM'06. 22nd IEEE International Conference on*. IEEE, 458–468.