Noname manuscript No.
(will be inserted by the editor)

Characteristics of Method Extractions in Java: A Large Scale Empirical Study

Andre Hora · Romain Robbes

Received: date / Accepted: date

Abstract Extract method is the "Swiss army knife" of refactorings: developers perform method extraction to introduce alternative signatures, decompose long code, improve testability, among many other reasons. Although the rationales behind method extraction are well explored, we are not yet aware of its characteristics. Assessing this information can provide the basis to better understand this important refactoring operation as well as improve refactoring tools and techniques based on the actual behavior of developers. In this paper, we assess characteristics of the extract method refactoring. We rely on a state-of-the-art technique to detect method extraction, and analyze over 70K instances of this refactoring, mined from 124 software systems. We investigate five aspects of this operation: magnitude, content, transformation, size, and degree. We find that (i) the extract method is among the most popular refactorings; (ii) extracted methods are over represented on operations related to creation, validation, and setup; (iii) methods that are targets of the extractions are 2.2x longer than the average, and they are reduced by one statement after the extraction; and (iv) single method extraction represents most, but not all, of the cases. We conclude by proposing improvements to refactoring detection, suggestion, and automation tools and techniques to support both practitioners and researchers.

Keywords Refactoring · Extract Method · Software Repository Mining · Software Evolution · Empirical Software Engineering

Andre Hora
Department of Computer Science, UFMG, Brazil, E-mail: andrehora@dcc.ufmg.br

Romain Robbes
Free University of Bozen-Bolzano, Italy, E-mail: rrobbes@unibz.it

1 Introduction

During software development, it is a common practice to refactor source code, that is, improving its internal structures without changing its external behaviour [16]. Common refactorings include renaming a class or a method, moving a method to another class, extracting a piece code to a new method, pulling up an attribute, among many other. For this purpose, Fowler proposes a catalog of refactoring operations [16], which are broadly adopted by the software industry [4, 10, 25, 30, 37] and studied by the research community [29,39,41,43,46]. Overall, the advantages of refactoring are numerous: removing code duplication, breaking long classes and methods, improving modularisation, increasing code maintainability and readability, to name a few [16].

A key refactoring operation is *extract method* [39].[1] This refactoring extracts a piece of code from an existing method, creates a new method based on the extracted code snippet, and calls it in the original method. There are several reasons to extract methods: developers perform method extraction to introduce alternative method signatures, to extract reusable methods, and to decompose methods, but also to improve testability, to enable overriding, and to enable recursion [39]. Indeed, extract method is recognized as the "Swiss army knife of refactorings" due to its many usages [39, 46]. As a result of its relevance, the literature has also proposed techniques to automatically identify extract method opportunities [38, 45]. These solutions can be integrated to IDEs so that developers receive refactoring suggestions while programming.

Although the rationales and the opportunities to detect method extraction are well explored by the literature [38,39,45,46], to the best of our knowledge, we are not yet aware of many aspects of this operation. In this context, some important questions to characterize this refactoring are still open, such as: how many method extractions are performed over time? which methods are target of the extractions and how do they change? what is the size of the extracted methods? what is the content of the extractions? Answering them would provide the basis (i) to better understand how one of the most important refactorings is performed by developers and (ii) to improve refactoring tools based on the actual behavior of developers.

In this paper, we assess characteristics of the extract method refactoring. We mine 124 popular Java systems, applying a state-of-the-art technique (RefDiff [40]) to detect over 70,000 instances of the extract method refactoring. We analyze these refactorings with respect to five aspects: magnitude, content, transformation, size, and degree, leading to the following research questions:

- *RQ1 (Magnitude): What is the frequency and extension of the extract method refactoring?* We find that extract method is the third most frequent refactoring, after rename and move method. 17% of the refactorings per system are method extraction; this ratio is independent of the system size, commits, and project popularity. 2% of the methods are created due to extraction, affecting 7% of the classes and 30.5% of the packages.

[1] https://refactoring.com/catalog/extractMethod.html

- *RQ2 (Content): What is the content of the methods in the extract method refactoring?* After categorizing methods in broad types of operations, we find that extracted methods are over represented on operations related to *creation*, *validation*, and *setup*. Operations related to *test* and *accessing* are unlikely to happen on extracted methods. On the other hand, the target of the extractions are often operations related to *processing*.
- *RQ3 (Transformation): What is the content of the extracted methods as compared to the target ones?* Extracted methods are likely to perform the same operation as the target ones. Test methods are the only exception: the extracted block of code is mostly extracted to *creation* methods.
- *RQ4 (Size): How large are the methods in the extract method refactoring?* The target of the extractions are methods 2.2x longer than the average; they are reduced by one statement after the extraction. The extracted methods themselves have similar sizes to the average.
- *RQ5 (Degree): How many methods have multiple extractions? How many methods are extracted from multiple places?* Only one method is extracted in the majority of the cases. However, 11% of the methods have multiple extractions (to decompose code), and 19% of the methods are extracted from multiple places (to remove duplication).

Overall, we find some particularities regarding the extract method refactoring: (i) an over concentration of target and extracted methods on certain operations, (ii) a longer size of the target methods and equivalence of the extracted when compared to the average, and (iii) multiple method extractions from the same method are occasional, but not rare. Based on these observations, we propose improvements to refactoring detection, suggestion, and automation tools and techniques to support both practitioners and researchers, for instance, to improve the UI of refactoring automation tools to include common prefixes, to recommend extract method when doing certain programming activities, and to ensure name consistency for multiple extracted methods.

The contributions of this research are threefold:

- We are the first to deeply study the refactoring operation with the most motivations, *i.e.,* the extract method, from a quantitative perspective.
- We perform a large analysis of the extract method refactoring by assessing its magnitude, content, transformation, size, and degree.
- We propose improvements to refactoring detection, suggestion, and automation tools and techniques.

Organization. Section 2 motivates this study by presenting the reasons to carry on research the extract method refactoring. Section 3 presents real world examples of method extraction. Section 4 details the study design while Section 5 presents the results. Section 6 discusses the major findings and implications. Section 7 states the threats to validity and Section 8 presents the related work. Finally, Section 9 concludes the paper.

2 Why Study Method Extraction?

There are important reasons to study the method extraction refactoring: to better understand it from the point of view of developers, to assess a refactoring that grabs attention of developers, and to improve refactoring tools.

2.1 Understanding how one of the most important refactorings is applied in source code by developers

The extract method refactoring is versatile: according to developers, it is the refactoring with the most motivations. Silva *et al.* [39] found 11 motivations for the extract method refactoring by performing a *firehouse interview* [35] (*i.e.,* when developers provide feedback shortly after performing a refactoring and the motivation behind it still fresh in their memory). The most popular motivation is to extract reusable methods so that they can be called in multiple places. The second reason is to introduce alternative method signatures, for example, with extra parameters. The third most frequent motivation to is decompose a method to improve its readability. Other motivations are: facilitate extension; remove duplication; replace a method while preserving backward compatibility; improve testability; enable overriding; enable recursion; introduce factory method; and introduce async operation. Indeed, in the influential book on refactoring, Fowler states the extract method as one of the most common refactoring he performs [16]; he also declares it as a key operation to do more refactorings: "*[...] That's why I see it as a key refactoring. If you can do Extract Method, it probably means you can go on more refactorings*".[2]

The rationales behind the extract method refactoring are well covered by the literature. However, we still lack basic information about how the extractions are actually applied, for example, with respect to their content, transformation, size, and degree. Better understanding these aspects can provide the basis to improve refactoring tools and techniques.

2.2 Assessing a refactoring that grabs attention of developers

As stated in the previous section, the extract method is a key operation with several motivations. In addition, it is also a common term in the software community as it is an operation often referenced during development activities. For instance, we assessed Stack Overflow, which is the most import Question and Answers platform nowadays. We found 671 posts including "extract method" either in the title or in the body.[3] From those posts, 276 are questions, which included 448 answers and 429K views from the community. These questions

[2] https://martinfowler.com/articles/refactoringRubicon.html

[3] Data collected with the Stack Exchange API: https://data.stackexchange.com

have 852 tags; the most common ones are: Java (57 occurrences), C# (42), Refactoring (37), Python (31), and PHP (20). Interestingly, there are also tags to IDEs, such as Eclipse (16), IntelliJ (6), Visual Studio (4), and Xcode (4).

To better understand the problems faced by the developers, we have manually inspected all the questions with score > 3, which represent 42 questions in our dataset. We found four major categories: *using refactoring tools*, *looking for refactoring tools*, *how to perform extraction*, and *side effects of extraction*. In 8 (19%) out of the 42 questions, developers asked very specific questions on how to use refactoring tools. For example, they asked how to automatically extract a method that contains the Java `continue`,[4] how to change the order of the extracted method,[5] the rationale behind the refactoring shortcuts in the IDE,[6] how to extract similar code,[7] and the reason the extracted method was static.[8] In 8 (19%) questions, developers looked for refactoring tools in specific programming languages, IDEs, and platforms, such as Vim, DevExpress, Xcode, Ruby, and Oracle. For example, a developer looked for a refactoring tool in Vim that can perform method extraction like in Eclipse.[9] In 3 (7%) questions, developers presented concrete examples and asked how to manually extract the code, *e.g.,* in the context of abstract classes,[10] tests,[11] and duplication.[12] In 2 (5%) questions, developers asked about the side effect of refactoring operations, particularly, method extraction, for instance, whether it would affect performance[13] and whether it can be applied on database stored procedures.[14] Finally, in other 8 questions, the term "extract method" was simply used to illustrate refactoring operations, while 13 questions were false positives.

"Extract method" is a term often adopted by developers during development activities. We found this term in hundreds of Stack Overflow posts and dozens of questions. Common problems faced by developers on these questions include: using refactoring tools, looking for refactoring tools, how to perform extraction, and side effects of extraction.

[4] Question ID: 1155947
[5] Question ID: 10289461
[6] Question ID: 2619228
[7] Question ID: 26674797
[8] Question ID: 511211
[9] https://stackoverflow.com/questions/2470653
[10] Question ID: 29257032
[11] Question ID: 4930742
[12] Question ID: 1898645
[13] 1247835
[14] 19972611

2.3 Prospect of better refactoring tools and techniques based on the actual behavior of developers

Long ago, performing method extraction with tool support was more challenging due to the lack of good refactoring tools [32]. Nowadays, this operation can be semi automated with the support of popular IDEs, such as Eclipse, NetBeans, IntelliJ, or Visual Studio. However, studies show that refactoring tools are commonly underused [23, 33, 34, 36, 39]. That is to say, developers often prefer to apply the refactoring manually due to several reasons, such as not trusting automated support [39]. To overcome this limitation, the literature proposes techniques to automatically identify extract method opportunities [38, 45] and other refactoring operations (*e.g.,* [6–8, 42, 44]) that can be directly automated by IDE-based refactoring tools.

> The literature proposes techniques to improve automated refactoring tools, aiming to make them more reliable and adopted by developers. We contribute to this research field by providing a large study to assess the actual behavior of developers performing method extraction in order to prospect better refactoring tools and techniques.

3 Method Extraction in a Nutshell

When performing the extract method refactoring, developer extracts a piece of code from an existing method and creates a new one based on the extracted code snippet. Figure 1 presents the simplest case of method extraction, in which a single method is extracted.[15] In this case, method `copyFile()` in version 2 is extracted from `copy()` in version 1. Notice that `copy()` is then updated in version 2 to call the extracted method `copyFile()`, passing the relevant arguments.

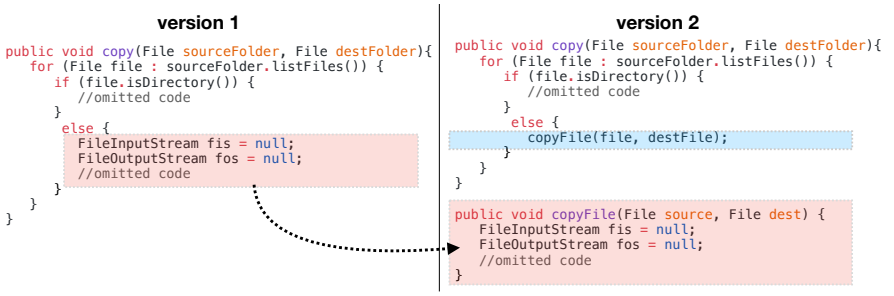

Fig. 1 Example of the extract method refactoring.

[15] Example from Arduino project: https://goo.gl/aD8n1N

Developers may opt to extract two or more methods when performing the refactoring. Figure 2 shows a method extraction example in which two methods are extracted, to decompose code.[16] In this case, methods `checkForUpdatablePlatforms()` and `checkForUpdatableLibraries()` in version 2 are extracted from `run()` in version 1, which is then updated to call the extracted methods in version 2. Notice that the extracted code may be slightly different from the original one. In the example, the extracted code includes the keyword `return`, which is absent in the original. In addition, the extracted code may have other changes, such as renaming, code removal, code addition, among many other; this makes the automatic detection of this operation a challenging task [39].

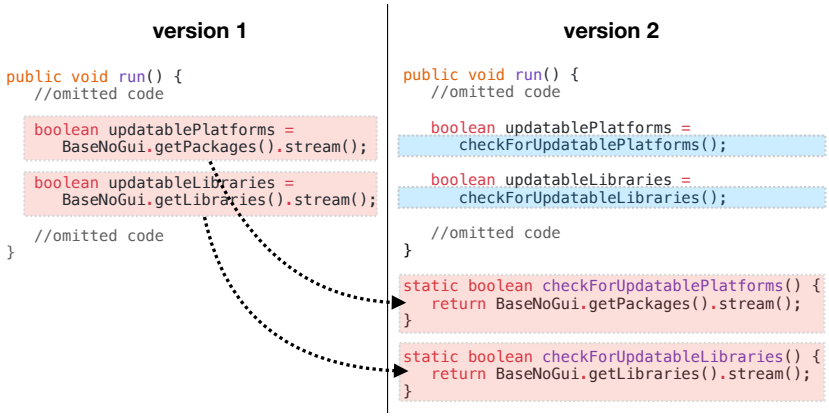

Fig. 2 Example of the extract method refactoring in which two methods are extracted from a single one.

Method extraction is not restricted to these scenarios. To avoid code duplication, several pieces of code may be extracted from distinct methods and consolidated in a single one. Figure 3 presents an example in which `newMonitor()` is extracted from methods `Editor()` and `selectSerialPort()`.[17] As in the previous example, the new created methods in version 2 are distinct when compared to the original one in version 1.

Overall, single and multiple extractions may happen during the refactoring due to several reasons, for example, to break long code or to avoid duplication [39]. In addition, the extracted methods may be different from the original ones, making harder their automated detection.

[16] Example from Arduino project: https://goo.gl/CaQWiB
[17] Example from Arduino project: https://goo.gl/yPvj5M

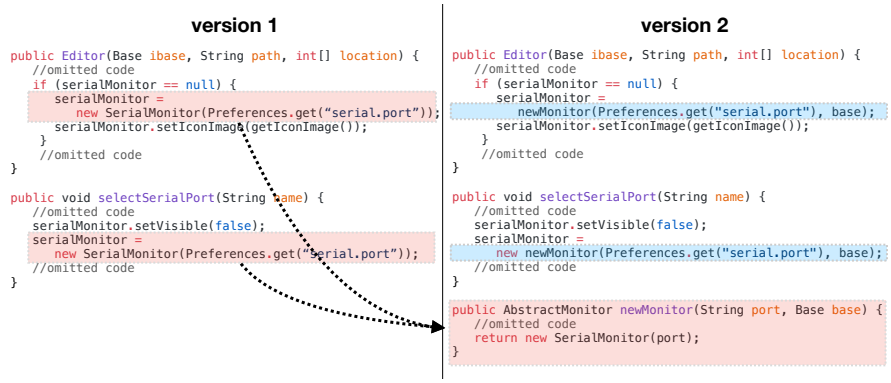

Fig. 3 Example of the extract method refactoring in which one method is extracted from two other methods.

4 Study Design

4.1 Collecting the Case Studies

Our case studies are collected from GitHub, the most widely-used social coding platform nowadays. We started with the 150 most popular Java projects as sorted by the star metric [9]. We then applied the following process to keep only relevant projects [21]. *First*, it was necessary to verify which of these projects are actually real software systems. In this filtering, we manually removed projects that were tutorials, examples, interviews, guides, among others non-systems. For example, *iluwatar/java-design-patterns*[18] was the most popular Java project, but it is not a software system, therefore, it was excluded from our dataset. *Second*, we removed the projects with less than 100 commits to filter out less active projects, and also the ones that are mirrored to GitHub, *i.e.,* not actively developed on it. After this process, we were left with 124 software systems.

Examples of the selected projects include: Elasticsearch, SpringBoot, Spring Framework, Google Guava, Facebook Fresco, Selenium, Jenkins, Arduino, Hadoop, JUnit4, which are projects largely adopted world wide. Figure 4 presents the distribution of the number of (a) Java files, (b) commits, and (c) stars for the selected projects. On the median, these systems have 219.5 Java files, 802.5 commits, and 7,971 stars. The largest one is Hadoop, with 11,299 Java files. Bazel is the project with the most commits: 19,183. Finally, the most popular is RxJava, with 36,472 stars.

[18] https://github.com/iluwatar/java-design-patterns

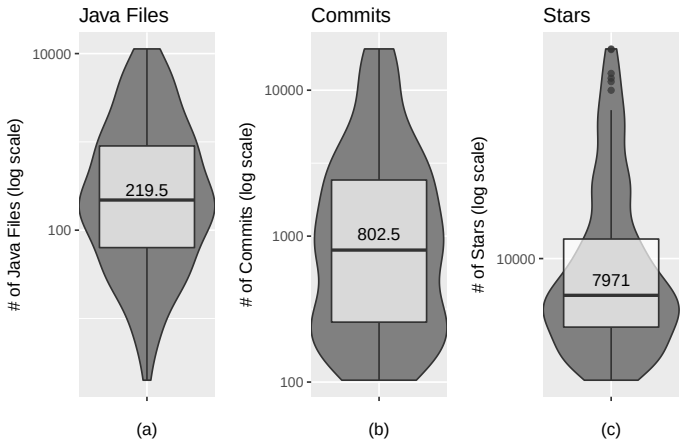

Fig. 4 Number of Java files, commits, and stars of the selected systems.

4.2 Selecting the Versions

After collecting the systems, we need to select the versions (*i.e.,* commits) to be analyzed. Projects using Git may have several branches under development. Thus, to facilitate evolutionary analysis, we assess the evolution of the main project branch, using the command `git log --first-parent`[19] to select the versions, since the Git documentation clearly states: "*This option can give a better overview when viewing the evolution of a particular branch*".

4.3 Computing the Refactorings

In this subsection, we explain how we model and automatically detect the extract method refactoring.

4.3.1 Modeling Method Extraction

We define two categories of methods to model the extract method refactoring: the target and the extracted method. The *target method* is the one in which the extraction is performed. The *extracted method* is the one that is created after the extraction. Consider the examples presented in Figure 5 showing two versions (*v1* and *v2*) of a system: the black node represents the *target method* and the red node represents the *extracted method*. Figure 5(a) illustrates the simplest case of method extraction: one method is extracted from the target one. Notice that from a single target method, one or more methods may be extracted. In Figure 5(b), for example, two methods are extracted from a single target method. Conversely, the extracted method may be originated

[19] https://git-scm.com/docs/git-log#git-log

from one or more target methods. For instance, in Figure 5(c), we see that the extracted method comes from two target ones. To detect the extract method refactoring and create the those models, we rely on the refactoring detection tool RefDiff [40].

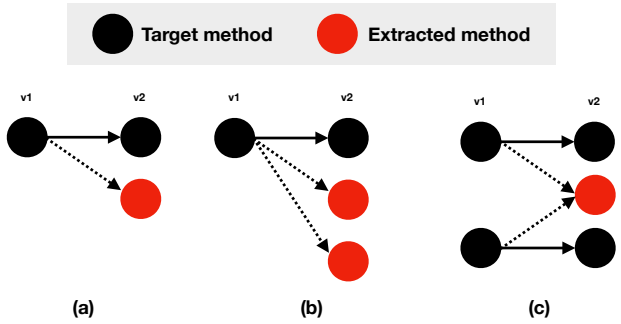

Fig. 5 Method extraction model examples.

4.3.2 Detecting Method Extraction

RefDiff [40] is a state-of-the-art refactoring detection tool on method extraction detection. RefDiff is a tool that detects refactorings performed in the version history of a system. It adopts a combination of heuristics based on static analysis and code similarity to detect 12 well-known refactoring operations, such as rename class, move class, rename method, inline method, pull up attribute, and push down attribute. RefDiff receives as input two versions of a system, and outputs a list of refactorings performed in version $n + 1$, when compared to the previous version n. In our study, we focus solely on the extract method operation.

Accuracy. In our study, we are primarily interested in finding both correct and complete extract method operations. Therefore, we report here the tool accuracy in terms of f-measure, which is the harmonic mean of precision and recall. RefDiff's authors provide two evaluations of their tool. First, they evaluated it using an oracle with well-known refactoring instances performed by students in seven Java projects. In this case, RefDiff achieved an overall f-measure of 96.8% (precision: 100%; recall: 93.9%). Considering solely the extract method refactoring, the f-measure is 94.6% (precision: 100%; recall: 89.7%). Moreover, in this evaluation, RefDiff also outperformed the results of similar tools, Refactoring Miner [39,46], Refactoring Crawler [12], and RefFinder [22]. In the second evaluation, RefDiff's authors analyzed 102 real refactoring instances from 10 GitHub projects. In this case, RefDiff achieved an overall f-measure of 89.3% (precision: 85.4%; recall: 93.6%). Considering solely the extract method refactoring, the f-measure is 84.7% (precision: 73.5%; recall: 100%).

Other refactoring detection tools. Recently, Tsantalis *et al.* [47] proposed the refactoring detection tool RMiner. When considering all refactoring operations, RMiner has an f-measure of 92% (precision: 98%; recall of 87%), improving on RefDiff's overall accuracy. However, when comparing the accuracy to detect the extract method refactoring on the same dataset, RefDiff presented slightly better results [47]. In this case, RMiner had f-measure of 91.2% (precision: 98.6%; recall: 84.7%) while RefDiff had f-measure of 92% (precision: 93%; recall: 90.9%). Therefore, in this paper, we adopt RefDiff because it has a better accuracy *with respect to the extract method refactoring.*

We run the approach to detect the extract method refactoring on each system (Section 4.1) and its respective set of versions (Section 4.2). We then assess this data to answer our five research questions. Our dataset and results are publicly available.[20]

5 Results

5.1 RQ1: What is the frequency and extension of the extract method refactoring?

We first investigate the frequency of extract method as compared to other refactoring operations. Table 1 presents the amount of refactorings in the 124 analyzed systems. In total, we detect 408,448 refactorings. The most frequent refactoring is the rename method, with 167,705 instances (41%). The second one is move method, which includes 90,675 instances (22%). The target refactoring, extract method, is the third most frequent in our dataset, with 70,059 instances (17%).

Table 1 Frequency of refactorings

Refactoring	#	%
Rename Method	167,705	41
Move Method	90,675	22
Extract Method	70,059	17
Move Class	40,218	10
Inline Method	12,499	3
Rename Class	11,716	3
Other	15,576	4
Total	408,448	100

Overall, the literature agrees that the extract method refactoring is among the most popular operations [19, 34, 39, 50] and our empirical study corroborates this finding. For example, Murphy-Hill *et al.* [34] found that the extract method was the fourth most frequently performed by developers in the

[20] https://bit.ly/2NsxgyB

Eclipse IDE: rename 71.8%, extract local variable 7.1%, move 5.6%, and extract method 4.8%. By analyzing code version history, Silva *et al.* [39] detected method extraction to be the most common[21] (extract method 33%, move class 30%, and move attribute 9%). Hora *et al.* [19] found the extract method as the second most frequent (rename method 26%, extract method 23%, and move method 22%). Recently, Vassallo *et al.* [50] found extract method ratios varying from 2.2% to 5.4%, depending of the ecosystem, while the rename method was the most popular operation with ratios between 23.6% and 36.7%, independently of the ecosystem. These variations may be explained by the fact that distinct refactoring detection tools (with different accuracy) were adopted, distinct software systems were analyzed, and distinct methodology were performed (*e.g.,* IDE monitoring and code version analysis). As a last observation, we notice that the rename operations are clearly the most common, independently of the set up; when they are not included in the analysis, other operations may have their ratios increased.

Figure 6(a) presents the distribution of the number of extract method and all refactorings. We notice that each system has on the median 734.5 refactorings. From them, 130 are extract method; in this case, the third quartile is 431.5, that is, 25% of the systems have at least 431.5 extractions. Figure 6(b) shows that the ratio of extract method refactoring per system is 17%; the third quartile achieves 24%.

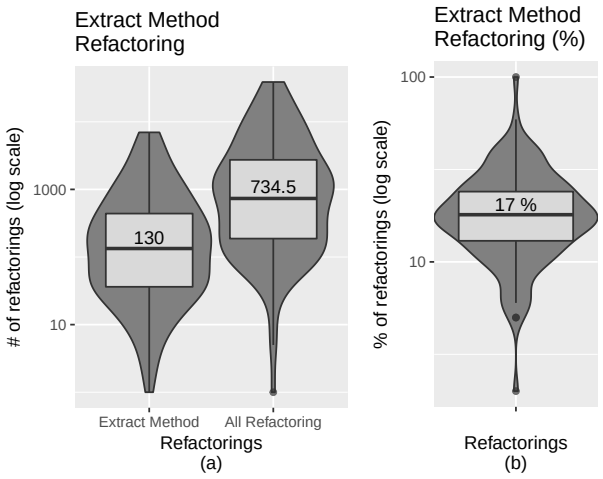

Fig. 6 (a) Number of extract method and all refactorings. (b) Ratio of extract method.

We also investigated whether the amount of extracted methods varied with the size, number of commits, and number of stars of the projects (Figure 7). We compared the top half with the bottom half of the projects, according to each metric (with a Mann-Whitney test of significance and Cliff's Delta

[21] The authors excluded the rename operations in their analysis.

for effect size). There was a very slight tendency (not significant) for smaller projects to have more method extraction; the same was true for projects with more commits, but not for stars.

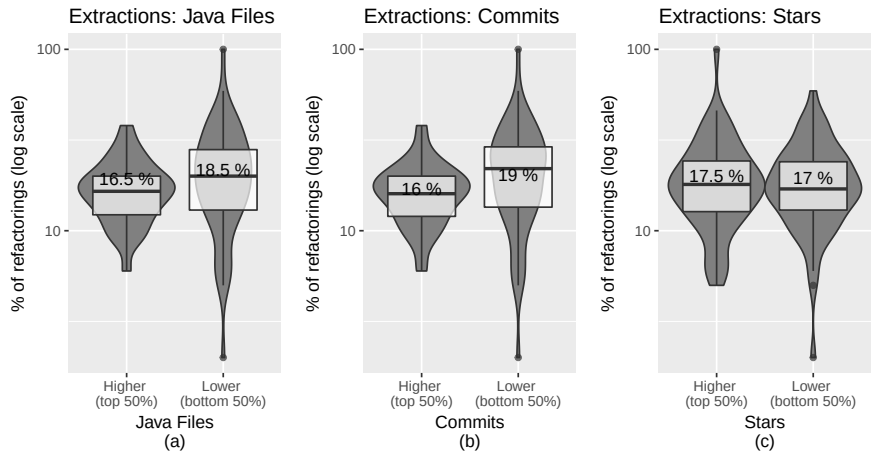

Fig. 7 Extract method by (a) number of Java files, (b) commits, and (c) stars.

Finally, we study the extension of the extract method refactoring in terms of impacted methods, classes, and packages. The median number of methods per system is 4,526.5. From these methods, on the median, 84.5 are created due to method extraction. This indicates that around 2% (one out of 50) of the methods per system are born from method extraction, as shown in Figure 8(a).

To better understand how the extract method refactorings are spread over the systems, we assess the classes and packages that are directly affected by them. In this case, we notice that the ratios are higher. Figure 8(b) shows that on the median 7% of the classes have at least one extracted method. Regarding packages, the median project has 30.5% of its packages with at least one extracted method. These figures support our initial impression that extract method is a relatively common operation.

Summary: Extract method is the third most frequent refactoring; this agrees with previous studies [19,34,39], in the sense that it is a popular refactoring. Overall, method extraction represents 17% of the cases; this ratio is independent of the system size, commits, and popularity. Around 2% of all methods are born due to the extract method refactoring; 7% of the classes and 30.5% of the packages include at least one extracted method.

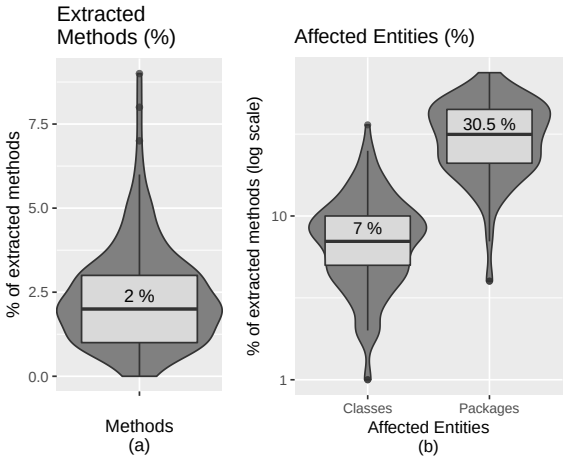

Fig. 8 (a) Ratio of extracted methods. (b) Affected classes and packages.

5.2 RQ2: What is the content of the methods in the extract method refactoring?

In this research question, we explore the content of the target and the extracted methods. As a solution to partially automate this analysis, we focus on the method prefixes, which are very often verbs and indicate the kind of operations they perform [20, 27]. Then, we categorize these prefixes to better understand the distinct prefix usages on the target and extracted methods.

5.2.1 Prefix Analysis

We first compare the prefixes of the target and extracted methods with the prefixes of all system methods, as presented in Tables 2 and 3. We notice that the prefixes are widespread: the top 10 most common prefixes cover 40% of target and 38% of extracted methods (*cf.* bottom of Table 2) and 41% for all methods (*cf.* bottom of Table 3). The top 100 prefixes cover 72% of target and 73% of extracted methods (65% of all methods).[22] Thus, a first observation is that the top 100 prefixes for target and extracted methods have higher coverage than they do in the overall corpus, when considering all methods. This indicates that their names appear to be slightly less varied.

As presented in Table 2, the most common prefix for the target methods is *test* (13%), which is followed by *get* (9%) and *create* (4%), totalizing 26% of the target methods. The most common prefixes for the extracted methods is *get* (15%), *create* (6%), and *is* (4%); the top three prefixes totalize 25% of the extracted methods. We notice some differences with the prefixes across all methods (Table 3). While *get* is more used in the entire corpus than in the

[22] Suffixes were much more widespread, with the top 10 prefixes covering only 7% of methods.

Table 2 Top 10 prefixes of the target and extracted.

Pos	Target Methods					Extracted Methods				
	Prefix	#	%	Sys.	%	Prefix	#	%	Sys.	%
1	test	6,886	13	64	52	get	6,111	15	112	90
2	get	4,864	9	101	81	create	2,646	6	82	66
3	create	2,073	4	75	60	is	1,744	4	96	77
4	should	1,560	3	25	20	set	1,386	3	96	77
5	set	1,457	3	92	74	add	1,244	3	80	65
6	on	1,159	2	79	64	new	799	2	60	48
7	add	1,095	2	72	58	assert	595	1	46	37
8	run	755	1	67	54	check	573	1	65	52
9	do	648	1	51	41	build	536	1	54	44
10	visit	644	1	17	13	to	466	1	60	48
Top 10	-	21,141	40	-	-	-	16,100	38	-	-
Top 100	-	38,450	72	-	-	-	30,529	73	-	-

Table 3 Top 10 prefixes of all methods.

Position	All Methods				
	Prefix	#	%	Sys.	%
1	get	372,719	16	124	100
2	test	175,287	8	95	77
3	set	121,757	5	122	98
4	is	57,420	3	121	98
5	create	53,769	2	112	90
6	on	38,330	2	118	95
7	add	35,076	2	111	90
8	should	31,804	1	87	70
9	to	31,264	1	108	87
10	write	23,554	1	93	75
Top 10	-	940,980	41	-	-
Top 100	-	1,486,565	65	-	-

target methods (16% vs 9%), the second most used prefix across all the corpus, *test*, does not appear in the top 10 of extracted method prefixes, but it is the top one of the target methods. Moreover, we can notice that the prevalence of prefixes among the 124 systems is not equally distributed (see column "Sys"). For example, while the prefix *get* of the extracted methods happens in 112 out of the 124 systems (90%), the prefix *assert* occurs only in 46 (37%).

5.2.2 Prefix Categorization

To better understand the differences of prefix usages over the three corpuses, we performed a manual classification of the top 100 prefixes appearing on these three corpuses (for a total of 127 distinct prefixes) with respect to the operation they indicate. The two authors of the paper independently categorized the prefixes, and after an initial agreement of about 75%, they achieved the consensus summarized in Table 4. The 12 categories are diverse: they include

methods related to creation, validation, and setup as well as methods about condition, assessing, and test. Table 4 also presents the amount of prefixes on each category and prefix examples. For instance, the category *creation* includes nine prefixes (*e.g., create*, *builder*, *generate*, etc.) while the category *validation* has six prefixes (*e.g., validate*, *check*, *verify*, etc).

Table 4 Categorized prefixes.

Categories	#Prefixes	Examples
Creation	9	create, builder, generate, make, new
Validation	6	validate, check, verify, matches, ensure
Setup	8	setup, initialize, configure, init, load
Processing	22	process, extract, calculate, execute, compute
Conversion	9	convert, format, as, to, from
IO	16	open, close, write, save, print
Collection	17	collect, contains, size, insert, iterator
Coordination	14	schedule, notify, wait, await, fetch
Release	8	release, dispose, shutdown, reset, stop
Condition	8	is, compare, equals, should, can
Accessing	7	get, set, value, index, put
Test	3	test, assert, mock

5.2.3 Extracted Methods Analysis

We present the categories with respect to the extracted methods in Table 5. For example, it shows that the category *creation* happen in 4,708 (11.25%) of the extracted methods and overall in 102,723 methods (4.5%). Column "Proportion" presents whether the category is more represented in the extracted methods (ratio greater than one) or for all methods (ratio less than one). Considering the category *creation*, we notice a ratio of 2.50 (*i.e.,* 11.25/4.5), meaning that this category is proportionally much more concentrated in the extracted methods than in the all methods. In addition to *creation*, the extracted methods are also very concentrated in the categories *validation* and *setup*, both being close to twice as common in extracted methods than in all methods. Thus, it seems that initialization steps in general represent behaviour that is easier to isolate. The same applies for validation steps. Interestingly, this is not the case for the *test* category, which is very under represented in extracted methods. While improving testability is a reason for extracting methods, it is possible that such a goal is better achieved by extracting a validation procedure in order to better test it, than by refactoring the tests themselves (indeed, this is confirmed by the classification of the target methods that we present next).

Continuing with the extracted methods, the categories of *processing*, *conversion*, *IO*, *collection*, and *coordination*, also have relatively more extracted methods. Categories *release*, and *condition* have slightly more extracted methods while *accessing* and *test* have proportionally less extractions. Indeed, *test*

Table 5 Categories of the extracted methods (Low: ratio ≤ 1; Medium: $1 <$ ratio ≤ 1.25; High: $1.25 <$ ratio ≤ 1.75; Very high: ratio > 1.75).

Categories	Extracted Methods		All Methods		Proportion (Ext/All)	
	#	%	#	%	Ratio	Concent.
Creation	4,708	11.25	102,723	4.5	2.50	Very high
Validation	1,149	2.75	29,009	1.28	2.15	Very high
Setup	1,468	3.50	41,239	1.81	1.93	Very high
Processing	2,962	7.08	111,202	4.87	1.45	High
Conversion	1,584	3.79	61,088	2.68	1.41	High
IO	2,167	5.17	88,799	3.89	1.33	High
Collection	2,463	5.88	104,262	4.57	1.29	High
Coordination	2,123	5.07	89,933	3.94	1.29	High
Release	684	1.63	31,425	1.38	1.18	Medium
Condition	2,492	5.95	126,171	5.53	1.08	Medium
Accessing	8,163	19.49	521,151	22.87	0.85	Low
Test	1,064	2.54	187,275	8.22	0.31	Low

is the category with the lowest ratio of extraction (0.31), meaning that extractions related to test are more unlikely. The same happens to *accessing*: although this category has 19.49% of the extractions, it is more common in all methods (22.87%).

5.2.4 Target Methods Analysis

Table 6 presents the classification for the target methods.[23] In this case, categories *processing* is over represented. Since processing contains a variety of prefixes describing potentially complex operation, it makes sense these operations would be refactored to isolate steps of these processes. Operations related to *setup* and *creation* happen in both target and extracted methods, although they are more frequent in extracted methods (particularly *creation*). While these initalization steps are good targets for extractions, it is possible that complex initalization procedures would be in need of further decomposition. Notice that *test* is the third most frequent category, meaning that although developers do not tend to extract unit test methods (*i.e.,* to extract methods prefixed with *test*), they do refactor test methods to modularize them, as indeed suggested by good development practices to improve code maintainability [27].

We notice that several categories that denote simpler operations (*e.g., condition*, *conversion*, and, particularly, *accessing*) are under represented in target methods. This is also intuitive, since these likely less complex operations also represent more modular behaviour in themselves. *Accessing* is underrepresented in both extracted and target methods, despite being the most common category, thus it seems that these basic methods are both too simple to be decomposed and somewhat too simple to be a valuable step to extract in a complex operation. *Conversion* methods, on the other hand, being more

[23] We omit the "All Methods" column in Table 6 because it is already presented in Table 5.

Table 6 Categories of the target methods (Low: ratio ≤ 1; Medium: $1 <$ ratio ≤ 1.25; High: $1.25 <$ ratio ≤ 1.75; Very high: ratio > 1.75).

Categories	Target Methods		Proportion (Base/All)	
	#	%	Ratio	Concent.
Processing	4,986	9.39	1.93	Very high
Setup	1,628	3.05	1.69	High
Test	7,243	13.62	1.66	High
Coordination	3,381	6.37	1.59	High
Creation	3,783	7.11	1.58	High
Collection	3,165	5.96	1.30	High
IO	2,769	5.20	1.29	High
Validation	931	1.75	1.28	High
Condition	2,699	5.07	0.91	Low
Conversion	1,148	2.17	0.81	Low
Release	572	1.07	0.78	Low
Accessing	7,084	13.33	0.58	Low

represented in extracted methods, seem to indicate operations that are at the right step of abstraction for such a refactoring.

Summary: Extracted methods are over concentrated on operations related to *creation*, *validation*, and *setup*. Extractions often occur on operations related to *processing*. *Test* methods are often target methods for extraction, but are not extracted themselves; the opposite is true for *conversion* methods. Basic operations, such as *accessing*, are more commonly found in methods not related to the extract refactoring.

5.3 RQ3: What is the content of the extracted methods as compared to the target ones?

This research question assesses the content of the extracted methods as compared to the target methods. For each method category, Table 7 shows the three most extracted categories. We notice that, in the majority of the cases, the extracted methods mostly belong to the same category of their target methods. For example, 49% of the *conversion* methods are extracted to *conversion* themselves. Similarly, the *accessing* methods are mostly extracted to *accessing* (58%). Interestingly, the *test* category is the only distinct from the other ones. In this case, *test* methods are mostly extracted to *creation* (24%). Indeed, from the previous research question, we have seen that developers do refactor test methods. Here, thus, we can observe that developers tend to refactor *test* methods mostly to extract *creation* methods. For example, method `testInsertAndSelect()` is extracted to `createRow(int,String)` in project Apache Storm[24] to avoid repetitive row instantiations.

The second most extracted methods are dominated by *accessing*: 9 out of the 12 categories are extracted to *accessing*, while 2 out of 12 are *creation*

[24] https://bit.ly/32lnFzd

Table 7 Categories of target methods and their respective extracted methods by category. Accessing is highlighted in **bold** and Creation in underline.

Target Method Category	1st Extracted Meths			2nd Extracted Meths			3rd Extracted Meths		
	Cat.	#	%	Cat.	#	%	Cat.	#	%
Conversion	Conver.	551	49	**Acces.**	160	14	Creat.	144	13
Accessing	Acces.	4,096	58	Creat.	656	9	Cond.	486	7
Setup	Setup	593	29	**Acces.**	465	23	Creat.	327	16
Processing	Proces.	1,722	30	**Acces.**	1,127	20	Creat.	581	10
Collection	Collec.	1,325	39	**Acces.**	679	20	Creat.	315	9
Creation	Creat.	2,071	48	**Acces.**	756	18	Collec.	274	6
Test	Creat.	1638	24	Test	1,376	20	**Acces.**	1,043	15
IO	IO	1,352	45	**Acces.**	485	16	Cond.	234	8
Release	Release	308	55	**Acces.**	48	9	IO	38	7
Validation	Valid.	329	34	**Acces.**	154	16	Cond.	144	15
Coordination	Coord.	1,036	28	**Acces.**	819	22	Creat.	343	9
Condition	Cond.	662	27	Creat.	515	21	**Acces.**	392	16

methods. Both categories encompass 11 of the 12 categories, with the only exception occurring for the *test* category, where *creation* methods are first. In addition, the third most extracted methods are often concentrated on *creation*, with 5 out of 12 categories, with *accessing* being in two additional categories. All in all, the *accessing* category appears in the top 3 of all 12 categories, while the *creation* category is in the top 3 for 9 out of 12 categories in total. Thus, while methods that are extracted tend to be first most extracted to the same category, we see that *accessing* and *creation* are predominant choices as well.

Table 8 details the previous analysis by showing the extracted methods by prefix instead of category. We see that *get*, *set*, and *create* prefixes are spread over all categories; this is not surprising because these prefixes belong to the categories *accessing* and *creation*. By checking other relevant prefixes, we verify the specific methods that are born from the extraction. For example, *conversion* methods often originate methods with prefixes *to* and *from*. From the *IO* category is extracted methods with prefixes *write* and *read* while from the category *validation* is extracted *check* and *is*.

Tables 9 and 10 present the top 10 most common prefix transformations in both absolute and relative values. As shown in Table 9, the most frequent prefix transformations in absolute values are *get* → *get* (2,835 times), *test* → *create* (1,063), and *create* → *create* (1,003). Notice that the prefix *test* is also often extracted to *of* and *get*. Table 10 present another view of the data: the transformations with higher proportion. Interestingly, in this case, all transformations remain with the same prefix, suggesting that their content is not changing. For example, 70% of the methods prefixed with *mock* are extracted to *mock* themselves.

Table 11 explores common prefixes in target methods and presents their three most extracted prefixes. For example, target methods prefixed with *get* are often extracted to methods prefixed with *get* (47%), *create* (12%), and *is* (4%). Overall, as in the category analysis, at the prefix level the content of

Table 8 Categories of target methods and their respective extracted methods by prefix.

Target Method Category	1st Extracted Meths			2nd Extracted Meths			3rd Extracted Meths		
	Prefix	#	%	Prefix	#	%	Prefix	#	%
Conversion	to	183	13	get	121	9	from	75	5
Accessing	get	3,164	36	set	564	6	create	389	4
Setup	get	335	14	create	190	8	load	172	7
Processing	get	875	12	create	345	5	parse	329	4
Collection	add	650	15	get	489	11	remove	171	4
Creation	create	1,139	22	get	583	11	new	356	7
Test	create	1,095	12	assert	828	9	of	780	8
IO	get	384	10	write	316	8	read	250	6
Release	clear	68	10	stop	54	8	shutdown	49	7
Validation	check	154	12	get	129	10	is	103	8
Coordination	get	555	12	create	224	5	set	205	4
Condition	is	428	12	get	290	8	create	276	8

Table 9 Top 10 most common prefix transformations in absolute values.

Prefix on...		#
Target Method	Extracted Method	
get	get	2,835
test	create	1,063
create	create	1,003
test	of	778
test	get	734
add	add	569
set	set	532
create	get	375
is	is	363
get	create	284

Table 10 Top 10 most common prefix transformations in relative values.

Prefix on...		%
Target Method	Extracted Method	
mock	mock	70
dispose	dispose	55
is	is	55
wait	wait	50
builder	builder	50
get	get	48
of	of	46
as	as	45
assert	assert	44
new	new	44

the extracted methods are mostly the same of their target methods, with *test* being the sole exception.

Qualitative Analysis. Overall, in this research question, we found that extracted methods often remain in the same category of the target ones. We

Table 11 Common prefixes of target methods and their respective extracted prefixes.

Target Meth	1st Extracted Meths			2nd Extracted Meths			3rd Extracted Meths		
Prefix	Prefix	#	%	Prefix	#	%	Prefix	#	%
get	get	2,835	47	create	1,063	12	is	281	4
test	create	1,063	12	of	778	9	get	734	8
create	create	1,003	35	get	375	13	add	110	4
add	add	569	39	get	164	11	create	43	3
set	set	532	29	get	168	9	create	73	4
is	is	363	55	get	73	11	has	21	3
mock	mock	21	70	create	6	20	initialize	1	3
dispose	dispose	17	55	get	4	13	unregister	4	13
wait	wait	73	50	current	9	6	get	7	5
assert	assert	162	44	get	34	9	create	26	7

manually inspected method extractions to better understand these cases, leading to distinct explanations. First, the extracted method may be changed to another prefix within the same category. For example, the target method `buildBean(Class,Map)` was extracted to `createInjector()`;[25] notice that both methods belong to the *creation* category as they have the *build* and *create* prefixes. In other cases, the methods may remain with the same prefix but change the suffix, as the following example, in which the *accessing* category was preserved: `getInstance()` was extracted to `getErrorReporter()`.[26] Another scenario is to keep the prefix, but add a new suffix, as in the transformation from `startBundle()` to `startBundleLocked()`.[27] Changes also occur in the parameters, with the addition of new parameters (*e.g.,* from `deletePendingReports()` to `deletePendingReports(boolean)`)[28] or the change of parameter types (*e.g.,* from `to(TypeLiteral)` to `to(Key)`).[29] Finally, we also find cases where the method change the class, as in the example in which the target method `ErrorReporter.getCustomData(String)` is extracted to `CrashReportDataFactory.getCustomData(String)`.[30] Table 12 summarizes how methods remain in the same category after being extracted.

Another interesting observation is that the extracted methods tend to be more specific than the target ones, that is, the extracted methods indicate more precisely the performing task. This happens both when the category remains the same and when the category changes. For instance, in Google ExoPlayer, the target method `assertSpans()` was extracted to several specific ones, such as `assertStyle()`, `assertFont()`, `assertBackground()`, `assertUnderline()`, among others.[31] Table 13 presents examples of methods being specialized after the extraction.

[25] https://bit.ly/36EJn4k
[26] https://bit.ly/2pJsogM
[27] https://bit.ly/34vuMXh
[28] https://bit.ly/2oUebxa
[29] https://bit.ly/2rcsqOt
[30] https://bit.ly/2CcVCXY
[31] https://bit.ly/2PWN2Vu

Table 12 Reasons to methods remain in the same category.

Reason	Example
Change prefix in the same category	`buildBean(Class,Map)` → `createInjector()`
Change suffix	`getInstance()` → `getErrorReporter()`
Add new suffix	`startBundle()` → `startBundleLocked()`
Add new parameters	`deletePendingReports()` → `deletePendingReports(boolean,boolean)`
Change parameter type	`to(TypeLiteral)` → `to(Key)`
Change class	`ErrorReporter.getCustomData()` → `CrashReportDataFactory.getCustomData()`

Table 13 Examples of transformation specialization.

Category	Target Method	Extracted Method
Same	`assertSpans()`	`assertStyle()`
	`decode()`	`decodeFile()`
	`post()`	`postSingleEvent()`
	`reselectTracks()`	`releasePeriodsFrom()`
	`injectLinks()`	`injectRuntimeLinks()`
Different	`execute()`	`restoreDatabaseFiles()`
	`update()`	`resolveDependency()`
	`after()`	`closePageCache()`
	`start()`	`connectorPortRegister()`
	`shouldRebuildFromLog()`	`getDatabasePath()`

Summary: Extracted methods are likely to remain with the same content of the original ones. *Test* methods are the only exception: they are mostly extracted to *creation* methods. Overall, *accessing* is the second and *creation* the third category. This tendency is also detected at the prefix level.

5.4 RQ4: How large are the methods in the extract method refactoring?

We assess the size of the methods involved in the extract method refactoring, that is, the target and extracted methods. We measure size in number of statements because this metric is not biased by formatting nor commenting changes. Then, we measure how their number of parameters varies.

5.4.1 Number of Statements

Figure 9 shows the number of statements of the target methods (before and after) as well as the extracted methods. To provide additional context, we present in the last plot the number of statement for all methods in the corpus. Before the extraction, the target methods have 5.76 statements; after the extraction, they shrink down to 4.75 statements. This represents a reduction of 21% on the number of statements (this difference is statistically significant, with *p-value* < 0.001 for Mann-Whitney test and *effect-size* small for Cliff's

Delta). Regarding the extracted methods, they have 2.45 statements, *i.e.*, 42% of the number of statements of the target method. The methods in the entire corpus are slightly larger (2.635 statements) than the extracted ones, however, the difference is not statistically significant. In contrast, they are much smaller than the target methods (with *p-value* < 0.001 and *effect-size* large). Figure 10 makes clear the distinction among the methods: while most of the target methods have 5 or more statements, most of the extracted methods have a single statement. Target methods are unlikely to have only one statement.

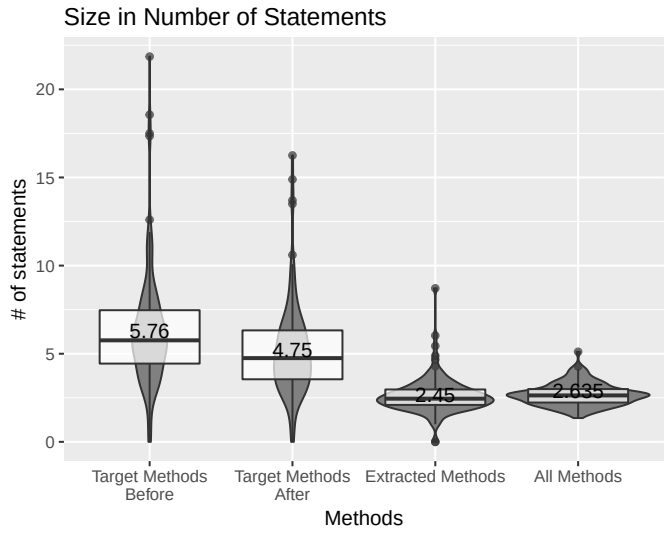

Fig. 9 Number of statements in the target (before and after), extracted, and all methods.

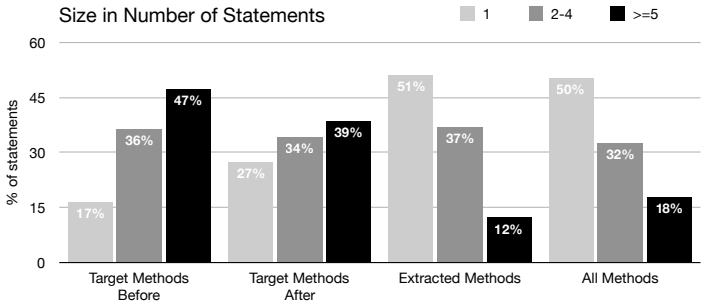

Fig. 10 Statements in the target (before and after), extracted, and all methods.

Table 14 presents the top 10 systems that most reduced their target methods after an extraction. Among the 124 analyzed projects, JetBrains Material

Theme had the highest reduction: its target methods were reduced by 106% (from 8.54 statements to 4.14). Timber JakeWharton, Calligraphy, and Android PickerView also had a large reduction, over 60%. However, we detect that the reduction is not correlated with the number of statements of the target method (Spearman $rho = 0.11$). That is, overall, the biggest extractions do not necessarily happen on the largest target methods. For example, the Android PickerView target methods were reduced by 62% (from 21.86 statements to 13.5), while the Glide Transformations target methods were reduced by only 7% (from 17.5 statements to 16.25).

Table 14 Top 10 systems with most reduction in number of statements.

System	Statement in Target Methods		Reduction (%)
	Before	After	
JetBrains Material Theme	8.54	4.14	-106
Timber JakeWharton	5.33	2.92	-83
Calligraphy	2.89	1.67	-73
Android PickerView	21.86	13.5	-62
Tink	8.7	5.58	-56
AndPermission	2.08	1.35	-54
Eureka	5.35	3.52	-52
CircleImageView	4.0	2.71	-48
Easypermissions	3.79	2.57	-47
EventBus	5.96	4.08	-46

5.4.2 Number of Parameters

In Figure 11, we assess the number of parameters for the same set of methods. The median does not change when we compare the target methods before and after (both with 1.28). The extracted methods also have a similar median amount of parameters (median 1.285), but the distribution is much more narrow. In contrast, looking at all methods, these tend to have less parameters, with a median of 1.06 (this difference is statistically significant, with *p-value* < 0.001 and *effect-size* medium), and an even narrower distribution.

Figure 12 details the parameter analysis and highlights the differences between target and extracted methods: the extracted methods are more likely to have 1 or 2 parameter than the target methods, which are likelier to have either more or less parameters.

Summary: The target methods are 2.2x longer than the average ones, and tend to have more parameters. After the extraction, the target methods are reduced by one statement. The extracted methods are similar in size to the average, but have more parameters, very often one or two parameters.

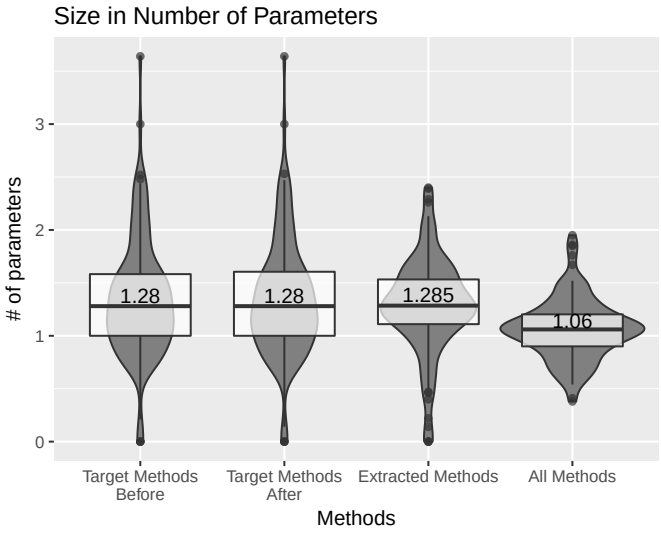

Fig. 11 Number of parameters in the target (before and after), extracted, and all methods.

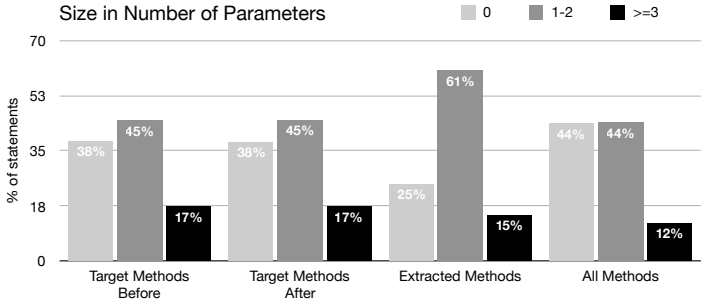

Fig. 12 Parameters in the target (before and after), extracted, and all methods.

5.5 RQ5: How many methods have multiple extractions? How many methods are extracted from multiple places?

In this last research question, we assess the methods that have multiple extractions; for that, we measure the out-degree of the target methods. Conversely, we assess the methods that are extracted from multiple ones; in this case, we compute the in-degree of the extracted methods. Considering the examples presented in Figure 5, the target methods in A and B have out-degree 1 and 2, respectively,[32] while the extracted method in C has in-degree 2. We recall that target methods with out-degree ≥ 2 (*e.g.,* case B) are often related to decomposition of long code, while extracted methods with in-degree ≥ 2 (*e.g.,* case C) are related to the removal of code duplication. Figure 13 presents the

[32] We only count the out-degree in the target methods with respect to the extracted methods, that is, the dashed lines in Figure 5.

distribution per system. Figure 13(a) shows that 89% of the target methods have out-degree 1, while only 11% have out-degree ≥ 2. That is to say, 11% of the extractions produce multiple methods. Moreover, Figure 13(b) presents that 81% of the extracted methods have in-degree 1, while 19% have in-degree ≥ 2. This means that 19% of the extractions are originated from multiple methods. Considering all the systems, 6,843 methods have out-degree ≥ 2 and 9,382 in-degree ≥ 2.

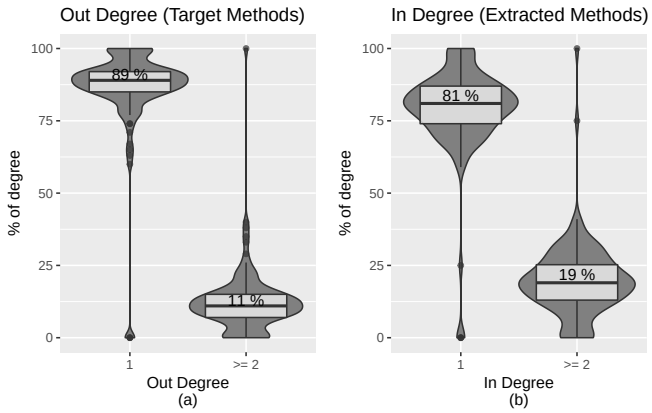

Fig. 13 (a) Out-degree of the target methods (1: single extractions; ≥ 2: multiple extractions). (b) In-degree of the extracted methods (1: extracted from one method; ≥ 2: extracted from multiple methods).

According to developers, code decomposition is one of the most important motivations to apply the extract method refactoring [39]. To better understand these cases, we analyze two aspects related to multiple extractions: their size and consistency.

5.5.1 Size of Multiple Extractions

Figure 14 presents the size of the methods with single and multiple extractions in number of statements. We notice that the ones with single extractions have on the median 5.53 statements while the ones with multiple extractions is much larger, 8.54 statements. That is, methods that undergo multiple extractions are on the median 54% longer than methods with single extractions.

5.5.2 Consistency of Multiple Extractions

We also assess the methods that are created due to multiple extractions. Figure 15(a) shows that each system has on the median 35 methods created due to multiple extractions. From these methods, we notice that on the median, 15.5 have common prefixes. This indicates that 45% of the extracted methods

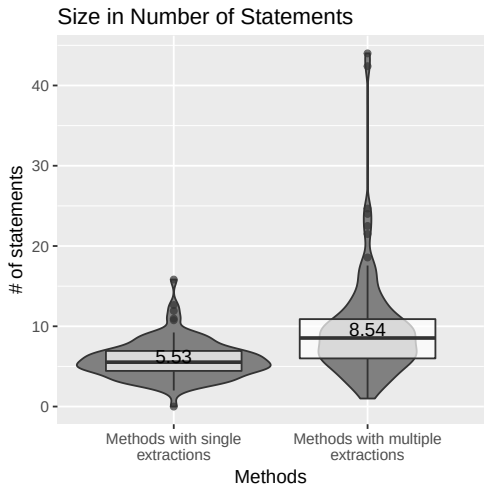

Fig. 14 Number of statements in methods with single and multiple extractions.

due to decomposition have common prefixes, as presented in Figure 15(b). Interestingly, the ratio of common prefixes is particularly high: the same prefixes are frequently used when decomposing a method. For example, in project MPAndroidChart, two methods are extracted from `drawData()` with the same prefix *is*: `isOffCanvasRight()` and `isOffCanvasLeft()`.[33] In project neo4j, five methods are extracted from `shouldReadBasicEntities()` with the same prefix *contains*.[34] In a more extreme case also in project neo4j, 12 methods are extracted from method `HaRequestType210()`, all with the same prefix `register`.[35]

Summary: If most refactorings extract a single method from a single target methods, exceptions are not uncommon: 11% of the methods have multiple extractions (to decompose code); 19% of the methods are extracted from multiple places (to remove duplication).

6 Findings and Implications

In RQ1 we detected that the extract method refactoring is among the most popular operations [19, 34, 39], representing 17% in our dataset. It also supported our initial impression that the extract method is a relatively common refactoring: we found that it impacts 2% of the methods, 7% of the classes, and 30.5% of the packages. In RQs 2, 3, 4, and 5, we dig on the content, transformation, size, and degree to better understand how method extraction

[33] https://goo.gl/qbTHRz
[34] https://goo.gl/otn7dC
[35] https://goo.gl/yeDPLa

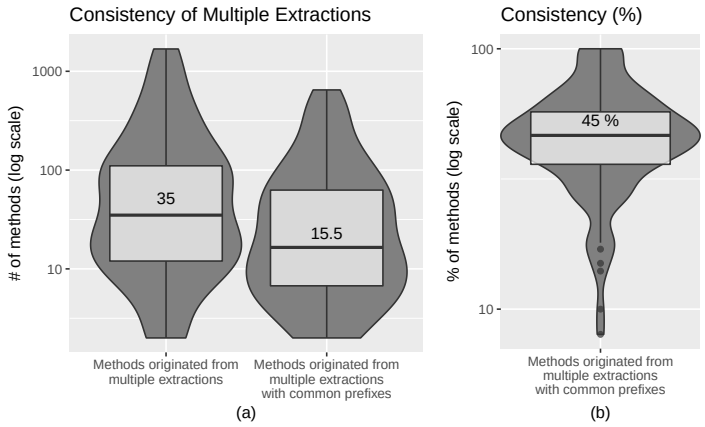

Fig. 15 Consistency of the methods originated from multiple extractions. (a) Methods with common prefixes. (b) Ratio of methods with common prefixes.

is performed by developers. Our major findings and implications are summarized in Table 15 and detailed in the following subsections.

6.1 Target and extracted methods are over concentrated on certain operations

We detected that the extracted methods are very often related to *creation*, *validation*, and *setup* operations. Furthermore, extractions commonly occur on operations related to *processing*. We also detected that test methods are often target methods for extraction, but are not extracted themselves; the opposite is true for conversion methods.

These findings provide the basis to improve two approaches that support the extract method refactoring: (i) techniques to identify extract method opportunities [38, 45] and (ii) refactoring automation tools (as the ones provided by Eclipse, NetBeans, IntelliJ, and Visual Studio). Currently, these techniques and tools do not propose names for the extracted methods, leaving this task to the end users. Therefore, we state the following implications:

1. We suggest that refactoring automation tools can have their UIs improved to include common prefixes (*e.g.,* the ones in Table 2) to help developers name the extracted methods.
2. In a more elaborated solution, the task of naming the extracted methods can be semi automated by inspecting their content. In this case, by solely detecting getting, creating, verification, addition, and setting operations, at least 40% of the extracted methods can be prefixed with intention revealing names. Several machine learning approaches have been proposed to generate a candidate method name given its method body (*e.g.,* [2, 3, 49]); name suggestion for the newly extracted method would be a natural fit for these

Table 15 Summary of findings and implications.

Findings	Implications
Over concentration of target and extracted methods on certain operations	1. Improve the UI of refactoring automation tools to include common prefixes 2. Semi automate the task of naming the extracted methods by inspecting their content 3. Tailor techniques to identify extract method opportunities to certain target and source methods 4. Recommend extract method when doing certain programming activity
Longer size of the target methods and equivalence of the extracted when compared to the average	1. Filter out the methods that are equal or smaller than the usual ones when looking for candidates for extraction 2. Extract methods with size equivalent to or smaller than the usual ones when extracting methods 3. Reduce the target method size in at least one statement to perform better than what is done nowadays by developers
Occasionality of multiple method extractions	1. Propose new techniques to identify multiple method extraction opportunities 2. Ensure name consistency for multiple extracted methods 3. Marry clone detection and extract method suggestion 4. Construct refactoring violation tools to detect bad refactoring practices

approaches, which could perhaps even benefit from the over representation of some categories of extracted methods.

3. Our categorization also provides further insights on which methods are target and source of extractions, which may be useful to improve the precision of techniques that recommend methods to be extracted (*e.g.,* [38,45]). For example, instead of naively analysing all the software system looking for candidate methods to be extracted, these tools can only analyze some category of methods (*e.g.,* the top 5 over represented in Table 6: processing, setup, test, coordination, and creation), while ignoring other categories (*e.g.,* accessing and release). Thus, techniques to identify extract method opportunities can be tailored to certain target and source methods, reproducing the actual behaviour of developers.
4. Our categorization have implications in the way refactoring operations are prioritized and/or recommended. For example, suppose a developer is refactoring a test method; in this case, he may be advised to extract a creation

method, since this is a commonly found transformation. Moreover, the refactoring recommendation engine could be tuned to detect specific categories of methods.

6.2 Target methods are longer than the average while extracted methods have size equivalent to the average

We found that the target methods are 2.2x longer than the average ones. After the extraction, the target methods reduce their size by one statement. Moreover, the extracted methods are similar in size to the average methods, but have more parameters, very often one or two parameters.

Techniques to identify extract method opportunities can benefit from the fact that the target methods are statistically *different* from the usual ones while the extracted methods are *equivalent* in statements. In this context, our results bring numbers to better calibrate these techniques. Thus, we present the following implications:

1. When looking for candidates for extraction, these techniques can filter out the methods that are equal or smaller than the usual ones, since those are rarely target of extractions (*e.g.,* in [38], a threshold of at least three statements was fixed for candidates).
2. Based on our findings, ideally, these techniques should extract methods with size equivalent to or smaller than the usual ones, not longer. Large blocks of code are much more rarely extracted.
3. Techniques that recommend extract method refactorings should strive to reduce the target method size in at least one statement to reproduce what developers actually do nowadays.

6.3 Multiple method extractions are not uncommon

We detected that most refactorings extract a single method from a single target methods. However, exceptions are not rare: 11% of the methods have multiple extractions in order to decompose code while 19% of the methods are extracted from multiple places in order to remove code duplication.

Techniques to identify extract method opportunities [38,45] focus on single method extractions, which are in fact the most common cases. However, multiple extractions related to code decomposition and code duplication removal represent 11% and 19% of the cases, which is not a negligible ratio. Therefore, we present the following implications:

1. New techniques can be proposed to identify multiple method extraction opportunities. Indeed, decomposing long method and removing duplication are among the motivations behind extract method [39], and those are the ones likely to generate multiple extractions. Thus, techniques that

propose to extract multiple code blocks from a single method, would be valuable additions. For instance, we detected that methods in which multiple extractions were made are overall 54% longer than methods with single extractions.

2. As a follow-up to a technique dedicated to extracting multiple code block, an approach that promotes name consistency among the extracted methods would be a valuable addition. A name suggestion approach that is able to take into account previous named extractions could leverage the previous suggestions to promote this consistency.
3. Since it is common to use an extracted method in multiple contexts to reduce duplication, approaches that marry clone detection and extract method suggestion could be designed. After a method is successfully extracted, an additional analysis could look for similar code fragments across the code base, and attempt to use the newly extracted method there.
4. We envision that refactoring detection tools (*e.g.,* [39, 40, 46, 47]) can also be the basis to construct refactoring violation tools (in the sense of code violation tools as FindBugs [5] and PMD [11]), so that developers can detect bad refactoring practices. For example, given that methods originated from decomposition are commonly equally prefixed, a possible violation would be that the extracted methods are not consistently prefixed or not prefixed at all. In fact, we found both cases in our dataset: the ones in which all but one extracted method is prefixed and the ones in which none of the extracted methods are prefixed; although this is not necessarily a problem, it would be important to at least warn the developer who applied it.

7 Threats to Validity

7.1 Construct Validity

The construct validity is related to whether the measurement in the study reflects real-world situations.

Refactoring in other programming languages. Refactoring can be performed in code written in any programming language. For example, recently, Fowler updated his catalog of refactoring operations, originally written for Java, to include JavaScript and functional examples.[36] Previously, the catalog was also transcripted to C++.[37] Thus, even though refactoring operations are more fomented in the Java ecosystem, most operations are language independent.

Extract method operation. The specific refactoring studied in this paper, extract method, is not restricted to Java, but can be applied to any OO or pro-

[36] https://www.oreilly.com/library/view/refactoring-improving-the/9780134757681
[37] http://jczeus.com/refac_cpp.html

cedural programming language.[38] Indeed, due to its many facets (*e.g.,* remove duplication, decompose long method, etc.), method extraction can be seen as a key operation, independently of the programming language paradigm. Moreover, as presented in our motivation section, extract method is a term often adopted by developers during development tasks, and often found in commit logs and issues. Finally, as stated in RQ1, extract method operations are among the most popular refactorings, after rename and move.

7.2 Internal Validity

The internal validity is related to uncontrolled aspects that may affect the experimental results

Accuracy to detect refactoring. We relied on RefDiff [40] to detect the refactorings. To the best of our knowledge, RefDiff is a state-of-the-art refactoring tool to detect method extraction. Its f-measure varies from 84.7% to 94.6% with respect to method extraction detection [40]. Thus, the due to the high accuracy of RefDiff, the risks of false positives and false negatives are reduced.

Manual classification. Our manual classification based on prefixes may have some imprecisions. Two authors performed it, with an agreement of around 75%, with differences being discussed until consensus was reached. Another source of imprecision is that a prefix may cover several use cases. For instance, part of the methods under the *set* prefix are actually *setUp* methods of JUnit tests, and would hence better be classified as testing. However, we find that such cases are limited as compared to the size of our corpus.

7.3 External Validity

The external validity is related to the possibility to generalize our results. We analyzed over 70K extract method refactorings mined from 124 popular, real-world, and open source Java systems. In RQ1, we also detected that the extract method is independent of system size, level of commits, and popularity. Despite these observations, our findings—as usual in empirical software engineering—may not be directly generalized to other systems, particularly to commercial nor to the ones implemented in other programming languages.

8 Related Work

8.1 Techniques to Detect Refactorings

There are several techniques intended to detect refactoring in version histories. These techniques are important to several applications, such as empiri-

[38] https://refactoring.com/catalog/extractFunction.html

cal studies about software evolution [19, 47]. Earlier refactoring detection approaches included the one of Weissgerber and Diehl [52], and Xing and Stroulia [55]. Other tools are Refactoring Miner [39, 46], Refactoring Crawler [12], and RefFinder [22]. RefDiff [40], the tool adopted in this study, combines heuristics based on static analysis and code similarity to detect 11 refactoring operations. Overall, when considering all refactorings, the f-measure of RefDiff varies from 89.3% to 96.8%. Recently, Tsantalis *et al.* [47] proposed the refactoring detection tool RMiner. This solution relies on an AST-based statement matching algorithm without user-defined thresholds. When considering all refactoring operations, RMiner has f-measure of 92%, improving RefDiff's overall accuracy. We recall that in this paper we adopted RefDiff because it has better accuracy *with respect to the extract method refactoring* (see Section 4.3.2 for more details), nevertheless we acknowledge that RMiner could also be used with no significant loss.

8.2 Refactoring Automation Tools and Techniques to Identify Refactorings Opportunities

Refactorings can be semi automated with the support of modern IDEs, such as Eclipse, NetBeans, IntelliJ, or Visual Studio, all of which support dozens of refactorings. Some studies, however, show that these kind of refactoring tools are commonly underused [23,33,34,36,39,48]. Often, developers prefer to apply a refactoring manually due to several reasons, such as not trusting automated support, simplicity of the operation, not finding the proper operation, and not being familiar with the refactoring capabilities [39].

To overcome this limitation, the literature proposes techniques to automatically identify refactoring opportunities. These solutions can, for example, be integrated to IDEs so that developers receive refactoring suggestions while programming. Techniques are then proposed to automatically identify opportunities, for instance, to extract method [38, 45], to extract class [6, 7], and to move method [8, 42, 44]. Our study derives implications to possibly improve tools related to method extraction, regarding the size of the target and extracted methods and their parameters, as well as the name of the involved methods.

8.3 Refactoring Assessment

Other studies focus on assessing refactoring to better understand them and propose better tools. Murphy *et al.* [31] analyzed the Eclipse IDE and captured refactoring usage. Murphy-Hill and Black observed developers performing extract method refactorings, and found several usability issues, particularly when selecting a valid range of source code [33]. Negara *et al.* [36] investigated manual and automated refactorings. They found that over 50% of the refactorings are performed manually and also that 30% are not stored on version control

systems. Vakilian and Johnson instrumented an IDE to detect refactoring usability problems [48], finding several usability issues, such as unclear error messages, or overly strong pre-conditions. Murphy-Hill *et al.* [34] performed a large analysis in four datasets to better understand how developers actually refactor. They found, for example, that developers frequently do not indicate refactorings in commit logs, that developers often do not configure refactoring tools, that about 90% of refactorings are performed manually, and that developers commonly mix refactorings with other programming activities. Hora *et al.* [19] assessed the impact of refactoring operations on software evolution and MSR studies (*e.g.,* [17, 18, 26, 28, 53, 54]). The authors detected that between 10 and 21% of the code changes at method level are about refactoring operations and, consequently, 25% of the code entities may have their histories split. Kim *et al.* [23,24] performed a field study of refactoring benefits and challenges at Microsoft. They detected that developers are not restricted to rigorous definitions on the behavior preserving aspect of refactoring. Wang [51] interviewed professional software developers on self-motivated and management factors for refactoring activities.

Recently, Silva *et al.* [39] investigated why developers perform refactoring activities. They asked the developers who actually performed the changes to explain the reasons behind their decisions. The authors produced a catalogue of 44 distinct motivations for 12 refactoring operations, such as extract method, move class, and move attribute. It was detected that the extract method is the most versatile refactoring, with 11 distinct motivations. They also found that refactoring is mainly driven by changes in the requirements and that the IDE used by the developers affects the adoption of automated tools. Our study contributes to this research topic by providing a deep analysis of the extract method refactoring.

8.4 Class and Method Stereotypes

Dragan *et al.* [14] introduced Method and Class stereotypes. Method Stereotypes would constitute an alternative to our ad-hoc classification of methods via their prefixes. The Stereotypes were defined based on static analysis of the programming language and its idioms, with an instantiation of the approach for C++. The work presents a taxonomy of method stereotypes that covers four main categories (accessor, mutator, collaborational, and creational). Accessor methods return information from the object, without mutating it. The category includes getters, predicates (returning booleans), and property (which derive information from the object state). Mutator methods change the state of the object, including setters and command methods (a command is more complex change, usually spanning more than one attribute). Collaborational methods interact with objects from another class, and may either access these objects, or mutate them. Finally, Creational methods involve creating or destroying new objects, either with constructors, desctructors, or factory methods. Method stereotypes have been used in a variety of contexts: to

characterize software systems [15], to improve on information retrieval-based feature location [1], or to characterize commits [13].

Compared to our prefix-based classification, we see some similarities, but also some differences. In particular, our classification is based on the meaning of the prefixes, rather than the structure of the code. While this may make it more sensitive to ambiguities in the prefixes, it allows it to be finer-grained in the kind of task that methods perform. For instance, our classification has specific categories for methods related to specific types of concerns, such as testing, conversion, collections, coordination, validation, setup, or release. Incorporating method stereotypes to include a taxonomy based on structural aspects would however be a valuable avenue for future work.

9 Conclusion

This paper presented a large empirical study to better understand the characteristics of extract method refactoring. We analyzed 124 software systems and detected 408,448 refactorings, of which 70,059 were instances of the extract method refactoring. Five research questions were proposed to assess the magnitude, content, transformation, size, and degree of the target refactoring. We reiterate the most interesting findings from our analysis:

- The extract method is the third most frequent refactoring, after rename and move method. 17% of the refactorings per system are method extraction.
- Of all methods, 2% are created due to method extraction; 7% of the classes and 30.5% of the packages include at least one extracted method.
- Method extractions are over concentrated on operations related to creation, validation, and setup.
- The target of the extractions are methods 2.2x longer than the average ones, and tend to have more parameters. In contrast, the extracted methods have size equivalent to the average ones.
- In 11% of the cases, more than one method is extracted; while 19% of extracted methods are used in more than one place.

We found particularities regarding the extract method refactoring, for example, an over concentration of target and extracted methods on certain operations and longer size of the target methods as well as equivalence of the extracted when compared to the average. Based on these findings, we highlighted several implications to improve refactoring tools and techniques, particularly to detect, automate, and suggest the extract method refactoring.

As future work, we plan to further explore the method categories, for instance: probably there is a difference in the size of methods across categories (*e.g.,* extracted *accessing* methods are possibly smaller). Moreover, we plan to perform a qualitative analysis with the support of Stack Overflow questions

and answers to better understand the major concerns of developers when applying the extract method operation. Finally, we plan to expand this research to better understand other important and challenging refactoring operations, such as move and inline method.

References

1. Alhindawi, N., Dragan, N., Collard, M.L., Maletic, J.I.: Improving feature location by enhancing source code with stereotypes. In: International Conference on Software Maintenance, pp. 300–309. Ieee (2013)
2. Allamanis, M., Barr, E.T., Bird, C., Sutton, C.: Suggesting accurate method and class names. In: Joint Meeting on Foundations of Software Engineering, pp. 38–49 (2015)
3. Allamanis, M., Peng, H., Sutton, C.: A convolutional attention network for extreme summarization of source code. In: International Conference on Machine Learning, pp. 2091–2100 (2016)
4. Ambler, S.W., Sadalage, P.J.: Refactoring databases: Evolutionary database design. Pearson Education (2006)
5. Ayewah, N., Pugh, W., Hovemeyer, D., Morgenthaler, J.D., Penix, J.: Using static analysis to find bugs. IEEE software **25**(5), 22–29 (2008)
6. Bavota, G., De Lucia, A., Marcus, A., Oliveto, R.: Automating extract class refactoring: an improved method and its evaluation. Empirical Software Engineering **19**(6), 1617–1664 (2014)
7. Bavota, G., Oliveto, R., De Lucia, A., Antoniol, G., Gueheneuc, Y.G.: Playing with refactoring: Identifying extract class opportunities through game theory. In: International Conference on Software Maintenance (ICSM), pp. 1–5 (2010)
8. Bavota, G., Oliveto, R., Gethers, M., Poshyvanyk, D., De Lucia, A.: Methodbook: Recommending move method refactorings via relational topic models. IEEE Transactions on Software Engineering **40**(7), 671–694 (2014)
9. Borges, H., Valente, M.T.: What's in a GitHub star? understanding repository starring practices in a social coding platform. Journal of Systems and Software (2018)
10. Brown, W.H., Malveau, R.C., McCormick, H.W., Mowbray, T.J.: AntiPatterns: refactoring software, architectures, and projects in crisis. John Wiley & Sons, Inc. (1998)
11. Copeland, T.: PMD applied, vol. 10. Centennial Books Arexandria, Va, USA (2005)
12. Dig, D., Comertoglu, C., Marinov, D., Johnson, R.: Automated detection of refactorings in evolving components. In: European Conference on Object-Oriented Programming, pp. 404–428 (2006)
13. Dragan, N., Collard, M.L., Hammad, M., Maletic, J.I.: Using stereotypes to help characterize commits. In: International Conference on Software Maintenance (ICSM), pp. 520–523. IEEE (2011)
14. Dragan, N., Collard, M.L., Maletic, J.I.: Reverse engineering method stereotypes. In: International Conference on Software Maintenance, pp. 24–34. IEEE (2006)
15. Dragan, N., Collard, M.L., Maletic, J.I.: Using method stereotype distribution as a signature descriptor for software systems. In: International Conference on Software Maintenance, pp. 567–570. IEEE (2009)
16. Fowler, M., Beck, K.: Refactoring: improving the design of existing code. Addison-Wesley Professional (1999)
17. Hora, A., Robbes, R., Anquetil, N., Etien, A., Ducasse, S., Valente, M.T.: How do developers react to API evolution? the Pharo ecosystem case. In: International Conference on Software Maintenance and Evolution, pp. 251–260 (2015)
18. Hora, A., Robbes, R., Valente, M.T., Anquetil, N., Etien, A., Ducasse, S.: How do developers react to API evolution? a large-scale empirical study. Software Quality Journal **26**(1), 161–191 (2018)
19. Hora, A., Silva, D., Robbes, R., Valente, M.T.: Assessing the threat of untracked changes in software evolution. In: International Conference on Software Engineering, pp. 1102–1113 (2018)

20. Host, E.W., Ostvold, B.M.: The programmer's lexicon, volume i: The verbs. In: International Working Conference on Source Code Analysis and Manipulation, pp. 193–202 (2007)
21. Kalliamvakou, E., Gousios, G., Blincoe, K., Singer, L., German, D.M., Damian, D.: The promises and perils of mining github. In: Working Conference on Mining Software Repositories, pp. 92–101 (2014)
22. Kim, M., Gee, M., Loh, A., Rachatasumrit, N.: Ref-Finder: a refactoring reconstruction tool based on logic query templates. In: International Symposium on the Foundations of Software Engineering, pp. 371–372 (2010)
23. Kim, M., Zimmermann, T., Nagappan, N.: A field study of refactoring challenges and benefits. In: International Symposium on the Foundations of Software Engineering, p. 50 (2012)
24. Kim, M., Zimmermann, T., Nagappan, N.: An empirical study of refactoring challenges and benefits at microsoft. IEEE Transactions on Software Engineering **40**(7), 633–649 (2014)
25. Lippert, M., Roock, S.: Refactoring in large software projects: performing complex restructurings successfully. John Wiley & Sons (2006)
26. Livshits, B., Zimmermann, T.: DynaMine: finding common error patterns by mining software revision histories. In: International Symposium on the Foundations of Software Engineering, pp. 296–305 (2005)
27. Martin, R.C.: Clean code: a handbook of agile software craftsmanship. Pearson Education (2009)
28. Meng, S., Wang, X., Zhang, L., Mei, H.: A history-based matching approach to identification of framework evolution. In: International Conference on Software Engineering, pp. 353–363 (2012)
29. Mens, T., Tourwé, T.: A survey of software refactoring. IEEE Transactions on software engineering **30**(2), 126–139 (2004)
30. Meszaros, G.: xUnit test patterns: Refactoring test code. Pearson Education (2007)
31. Murphy, G.C., Kersten, M., Findlater, L.: How are Java software developers using the Elipse IDE? IEEE Software **23**(4), 76–83 (2006)
32. Murphy-Hill, E., Black, A.P.: Breaking the barriers to successful refactoring: observations and tools for extract method. In: International Conference on Software engineering, pp. 421–430 (2008)
33. Murphy-Hill, E., Black, A.P.: Refactoring tools: Fitness for purpose. IEEE Software **25**(5) (2008)
34. Murphy-Hill, E., Parnin, C., Black, A.P.: How we refactor, and how we know it. IEEE Transactions on Software Engineering **38**(1), 5–18 (2012)
35. Murphy-Hill, E., Zimmermann, T., Bird, C., Nagappan, N.: The design space of bug fixes and how developers navigate it. IEEE Transactions on Software Engineering **41**(1), 65–81 (2015)
36. Negara, S., Chen, N., Vakilian, M., Johnson, R.E., Dig, D.: A comparative study of manual and automated refactorings. In: European Conference on Object-Oriented Programming, pp. 552–576. Springer (2013)
37. Roberts, D., Brant, J., Johnson, R.: A Refactoring Tool for Smalltalk. Theory and Practice of Object Systems **3**(4) (1997)
38. Silva, D., Terra, R., Valente, M.T.: Recommending automated extract method refactorings. In: International Conference on Program Comprehension (ICPC), pp. 146–156 (2014)
39. Silva, D., Tsantalis, N., Valente, M.T.: Why we refactor? confessions of GitHub contributors. In: International Symposium on the Foundations of Software Engineering, pp. 858–870 (2016)
40. Silva, D., Valente, M.T.: RefDiff: detecting refactorings in version histories. In: International Conference on Mining Software Repositories, pp. 269–279 (2017)
41. Simon, F., Steinbruckner, F., Lewerentz, C.: Metrics based refactoring. In: European Conference on Software Maintenance and Reengineering, pp. 30–38 (2001)
42. Terra, R., Valente, M.T., Miranda, S., , Sales, V.: JMove: A novel heuristic and tool to detect move method refactoring opportunities. Journal of Systems and Software **138**, 19–36 (2018)

43. Tourwé, T., Mens, T.: Identifying refactoring opportunities using logic meta programming. In: European Conference on Software Maintenance and Reengineering, pp. 91–100 (2003)
44. Tsantalis, N., Chatzigeorgiou, A.: Identification of move method refactoring opportunities. IEEE Transactions on Software Engineering **35**(3) (2009)
45. Tsantalis, N., Chatzigeorgiou, A.: Identification of extract method refactoring opportunities for the decomposition of methods. Journal of Systems and Software **84**(10), 1757–1782 (2011)
46. Tsantalis, N., Guana, V., Stroulia, E., Hindle, A.: A multidimensional empirical study on refactoring activity. In: Conference of the Centre for Advanced Studies on Collaborative Research, pp. 132–146 (2013)
47. Tsantalis, N., Mansouri, M., Eshkevari, L.M., Mazinanian, D., Dig, D.: Accurate and efficient refactoring detection in commit history. In: International Conference on Software Engineering, pp. 483–494 (2018)
48. Vakilian, M., Johnson, R.E.: Alternate refactoring paths reveal usability problems. In: Proceedings of the 36th International Conference on Software Engineering, pp. 1106–1116 (2014)
49. Vasilescu, B., Casalnuovo, C., Devanbu, P.: Recovering clear, natural identifiers from obfuscated js names. In: Joint Meeting on Foundations of Software Engineering, pp. 683–693 (2017)
50. Vassallo, C., Grano, G., Palomba, F., Gall, H.C., Bacchelli, A.: A large-scale empirical exploration on refactoring activities in open source software projects. Science of Computer Programming **180**, 1–15 (2019)
51. Wang, Y.: What motivate software engineers to refactor source code? evidences from professional developers. In: International Conference on Software Maintenance, pp. 413–416 (2009)
52. Weissgerber, P., Diehl, S.: Identifying refactorings from source-code changes. In: International Conference on Automated Software Engineering, pp. 231–240 (2006)
53. Wu, W., Gueheneuc, Y.G., Antoniol, G., Kim, M.: AURA: a hybrid approach to identify framework evolution. In: International Conference on Software Engineering, pp. 325–334 (2010)
54. Xavier, L., Brito, A., Hora, A., Valente, M.T.: Historical and impact analysis of API breaking changes: A large scale study. In: International Conference on Software Analysis, Evolution and Reengineering, pp. 138–147 (2017)
55. Xing, Z., Stroulia, E.: Refactoring detection based on umldiff change-facts queries. In: Working Conference on Reverse Engineering, pp. 263–274 (2006)