

Quality Histories of Past Extract Method Refactorings

Abel Mamani Taqui[1], Juan Pablo Sandoval Alcocer[1(✉)], Geoffrey Hecht[2], and Alexandre Bergel[2]

[1] Departamento de Ciencias Exactas e Ingenierías, Universidad Catolica Boliviana "San Pablo", Cochabamba, Bolivia
jsandoval@ucb.edu.bo

[2] Department of Computer Science (DCC), University of Chile, Santiago, Chile

Abstract. Modern programming environments offer the Extract Method refactoring as a way to improve software quality by moving a source code fragment into a new method. This refactoring comes with an immediate positive feedback by shortening the refactored method. It can also increase code re-usage and encourage developers to remove code clones.

The impact of refactorings on the software quality has been the topic of many research efforts. However, these refactorings are usually studied in groups. Therefore the metrics evaluated and the observation are not tailored to a specific refactoring, thus hiding a valuable insight on how practitioners use a refactoring in particular.

In this paper, we conduct an assessment of the quality impact resulting from the Extract Method refactoring. Our results statistically confirm the tendency of the Extract Method to improve complexity and slightly worsen cohesion, respectively in 46% and 70% of the refactoring. In addition, we observe that the Extract Method favors re-use and reduces occurrences of code clones in 56% of the extracted methods. However, our results also show that some specific cases are contrary to the previously mentioned trends and that it is therefore necessary to study refactorings at a low granularity.

Keywords: Refactoring · Software quality · Software maintenance

1 Introduction

Refactoring is now an essential practice for improving the quality of a source code without changing its external behaviors [16,18,19]. Refactoring has multiple expected benefits, such as easing maintenance, reducing code complexity, improving code readability and removing potential code smells. But recent studies have shown that refactorings do not always improve the quality of the source code, and could even worsen it in some scenarios [3,4,6].

© Springer Nature Switzerland AG 2021
O. Gervasi et al. (Eds.): ICCSA 2021, LNCS 12955, pp. 336–352, 2021.
https://doi.org/10.1007/978-3-030-87007-2_24

This paper contributes to this line of research by building on a partial replication of existing efforts on characterizing the impact of the refactoring on software quality. Our paper contributes to the field of code quality and refactoring by focusing on one refactoring, and therefore lowering the granularity of our research. Most of the work on software quality and refactoring consider various refactorings, thus, missing opportunities to go in depth of the Extract Method refactoring.

The Extract Method is one of the most common refactorings [5,16,21]; it allows the developer to take a fragment of code, move it into a new method, and replace the refactored code with a call to the new method. It it supposed to improve readability, reduce code duplication and remove long method code smells.

Modern programming environments make the Extract Method refactoring a central feature. Consider this informal search on Google video: searching for "Extract Method refactoring" lists more than 69K different videos that illustrate how to perform the refactoring using many different programming environment end IDEs. This informal measurement highlights the relevance of this feature for practitioners. Understanding the implication of using this refactoring on software quality is important because it has been shown that a refactoring, when improperly employed, may degrade the quality of refactored software [3].

This paper presents new insights from a quantitative analysis of a large corpus of 2,712 Extract Method refactorings gathered from 200 Java projects hosted on GitHub. Our objective is to explore the usage of the Extract Method refactoring in Java in order to gain insight on how this refactoring helps developers improve the quality of their code. In particular, our study provides a detailed analysis of the impact of the Extract Method refactoring on cohesion, complexity, code clones, code re-usability and method visibility.

Findings. Our study reveals a number of facts on the Extract Method refactoring:

- 46% of the Extract Method refactorings help to reduce at least 4 complexity metrics of the refactored method and therefore of the class;
- 70% of the extracted methods reduce the class cohesion, but overall statistically this effect is very small. In addition, we found that 10% of these methods are not static and do not depend on any attribute/method of their class;
- 56% of the Extract Method refactorings favor code re-usability and help reduce code clones;
- 54% of the extracted methods are private, while 22% of the public extracted methods are overexposed and may reduce their scope to private.

As in previous works [3,4,6], our results confirm that the refactoring may have unexpected effects on code quality metrics in some cases. In addition, our results also show that Extract Method is often exploited partially and that the extracted method could be improved in terms of visibility, code clones and static

modifiers. You will find the information of extract methods and commits under study online[1].

Outline. The paper is structured as follows: Sect. 2 reviews the studies related to refactoring and software quality; Sect. 3 highlights the need for conducting a focused study and highlights the opportunity exploited in this paper; Sect. 4 outlines our empirical setup, the collection process, and metrics we used; Sect. 5 discusses our findings; Sect. 6 captures any threats to the validity of our work; Sect. 7 concludes and presents our future work.

2 Related Work

Numerous studies have exploited mining tools to produce empirical studies on the relationship between code quality and refactorings [4,6–9,11,20].

Elish and Alshayeb [8] propose a classification of 12 refactorings based on their effect on nine internal and six external quality attributes. Evaluating three open source projects and three course projects, they determined if the refactoring tends to decrease or increase the quality attributes. They observed that Extract Method tends to increase Response For a Class, Number Of Methods, Numbers of Lines of Code and Lack of Cohesion in Methods, however the impact is not quantified.

Concerning the relation between Extract Method and code clones, Choi *et al.* [7] investigated how code clones are merged during the software evolution of three Java open-source projects. They observed that Replace Method with Method Object and Extract Method were the most commonly used refactorings to remove clones. They suggest improvements for refactoring tools allowing the detection of clones with different sequences of tokens. They did not study if clones could be introduced by the refactorings.

Bavota *et al.* [4] investigated the relations between metrics or code smells and refactoring activities on 63 releases of three open source applications. They observed that there is no clear relationship between the part of code which developers chose to refactor and quality metrics, moreover about 40% of the refactorings were performed on classes affected by code smells but only 7% of them removed the smell.

Kádár *et al.* [11] evaluated the impact of more than 40 types of refactorings on 50 metrics in seven open-source projects. They observed that the classes with the worst maintainability metrics are subject to more refactorings. Overall, they observed a positive effect of the refactorings on most of their metrics, including code clones occurrences. However, the results of the refactoring are grouped and therefore it is not possible to distinguish the influence of each refactoring on the results.

Based on 25 metrics related to five internal quality attributes: cohesion, coupling, complexity, inheritance, and size. Chávez *et al.* [6] observed that more than 94% of the applied refactorings in 23 open-source projects are performed

[1] http://bit.ly/RefactoringDataset.

on program elements with at least one quality metrics considered as critical. In 65% of the cases, these critical quality metrics were improved and the remaining 35% refactorings had no effect. Overall on all metrics, 55% of the observed refactorings improved internal quality attributes, and 10% were associated with a quality decline.

Al Dallal *et al.* [1] performed a systematic literature review on the impact of object-oriented refactoring on quality attributes. They found numerous studies on the impact of refactoring on quality, and confirmed that refactoring does not always improve all quality attributes. However 85.5% of the studies they considered did not apply any statistical techniques and most of the studies concerned multiple refactoring scenarios, which could not be distinguished from each other. These undesirable practices prevented them from precisely analyzing the individual impact of refactoring on quality.

Pantiuchina *et al.* [17] empirically investigated the correlation between seven commonly used metrics and the declared intentions of developers to improve some quality attributes (i.e. cohesion, coupling, complexity and readability) in 1,282 commits. The study shows that the quality improvements expected by developers is not always reflected in the associated metrics. In addition to pointing out inconsistencies between how code quality attributes are perceived by developers and commonly used metrics, the authors recommend that the combination of many quality metrics should be preferred over one.

AlOmar *et al.* [2] also investigated commits where developers showed intentions to perform refactorings. They discovered that developers use a variety of patterns to perform refactorings, and that they often directly mention quality attributes or removing code smells in the associated commit messages.

In another study, AlOmar *et al.* [3] investigated the correlations between refactorings and 27 quality metrics associated to 8 quality attributes such as cohesion, coupling or complexity in 3,795 open source Java projects. They observed that in most cases metrics can reflect the developer intentions of improving quality reported in the commit messages. However, they did not find any metrics which correlated with the developer's intentions to improve encapsulation, abstraction or design size.

In summary, we observe in most publications that refactoring does not always improve the quality of source code as one might expect. However, it is difficult to derive a more detailed consensus from these previous works. For some publications, the quality metrics are observed to be very relevant and the effect of refactorings on them correspond well to the intentions of the developers [2,3,6]. But for other publications, the usual metrics seem less relevant for analyzing the impact of refactorings [4,17]. However, it is important to note that the refactorings used and the metrics used vary between all these publications. In this paper, we focus on the Extract Method refactoring. In this way we can ensure that the chosen metrics, associated results and observations are relevant to this particular refactoring and that the effects observed are not due to another refactoring.

3 A Focused Study

The previous section shows that there are numerous high-quality works on the impact of refactoring on code quality. They provide valuable results, however the analyzed refactorings are studied together on many different metrics allowing only aggregated observations. As observed by Al Dallal *et al.* [1], it is hard to distinguish between the effects of individual refactoring when multiple refactorings are applied. Especially since the effects of refactorings on metrics can be conflicting. For example, the impact of dozens of refactorings on dozens of metrics are reported by AlOmar *et al.* [3] and Kádár *et al.* [11], but the results of all the refactorings are grouped together. The impact of a particular refactoring is therefore not explicit. Overall refactorings do improve cohesion but is it the case for the Extract Method (or any other particular refactoring)?

Publications like the one from Elish and Alshayeb [8] do report an evaluated impact for each refactoring on different metrics, for example that Extract Method does increase LCOM. However, it is reported as a tendency, which is not quantified and the possible exceptions to this tendency are not considered.

Chavez *et al.* [6] provide more detailed results, by reporting if the effects of each refactoring is positive, neutral or negative on most metrics (or at least one metric) of a group (cohesion, coupling, complexity, inheritance, size). However, the results are not detailed for each metric. Our paper contributes to the field by focusing on only Extract Method, in order to provide an in-depth analysis. It is complemented by statistical tests to allow precise comparisons with future works.

Our goal is to provide a detailed and quantified analysis on each of our metrics. In addition to cohesion and complexity, our research incorporates metrics which are not considered in previous works (e.g. visibility and code reuse) which we found relevant to report for Extract Method. We also consider a large dataset of open-source applications.

The results we obtained match some of the previous results, in particular:

- 46% of the extracted methods improves most of our complexity metrics, similar to the 45% observed by Chavez *et al.* [6] for their complexity group;
- 70% of the refactored methods increase LCOM, whereas Chavez *et al.* [6] observed a worsened cohesion in 59% of cases for their cohesion group.

However, as detailed in the subsequent sections, our effort lead us to new findings:

- 56% of the Extract Method helps reduce code clones;
- 10% of the extracted methods could be static;
- 51% of the extracted methods are called more than once;
- 22% of the public extracted methods are overexposed, and could be privatized.

The following sections details the methodology we adopted, and our results.

4 Empirical Study Setup

Our methodology has four-steps, which are described in the following subsections.

4.1 Collecting Open Source Projects

To build our dataset, we collect the two hundred most popular Java GitHub projects. For this, we use the GitHub API to collect the Java projects that have more stars. The stars in GitHub represents the number of GitHub accounts that follow this projects, which makes it a reliable proxy for popularity. The number of stars of these two hundred projects ranges from 2,199 to 2,786. These projects contain a total of 8,793 software versions and 7,885 files.

4.2 Detecting Extracted Methods

We use the tool Refactoring Miner to detect commits where developers perform an Extract Method refactoring [22]. We analyze all the commit history for each project. In total, refactoring miner reports 80,842 refactorings along the whole commit history of the two hundred projects under analysis. From these, 3,059 correspond to a unique Extract Method refactoring.

4.3 Computing Metrics

As the main difference from previous works [13–15], we focused only on the Extract Method refactoring, therefore we selected metrics relevant to that refactoring. The metrics were computed in the refactored class, refactored method and extracted method; depending on the metric. We categorize these metrics in four groups: complexity, cohesion, reuse, and visibility.

Complexity. We use seven complexity metrics to measure the complexity of the refactored method before and after the method extraction. Table 1 briefly describes each of these metrics.

Table 1. Complexity metrics computed in the refactored method

Name	Description
McCabe cyclomatic complexity	# Unique possible paths through the method
McClure's complexity	# Comparisons plus # control referenced variables
Nested block depth (NBD)	The maximum depth of nesting within a method
Number of control Variables	# Control variables reference within the method
Number of comparisons	# Comparisons in a method
Number of parameters	# Parameters a method takes as input
Input/Output variables	# Parameters plus 1 (assuming 1 as a return value)

Cohesion. We use the metric Lack Of Cohesion (LCOM) to measure the cohesion of a class before and after the extraction. For this, we use the Herdenson-Sellers method: $(< r > -|M|)/(1-|M|)$. Where M is the set of methods defined by the refactored class, and F the defined fields. Let be $r(f)$ the number of methods that access a field f, where $f \in F$, and $< r >$ is the mean of $r(f)$ over F [10]. LCOM is greater when the class methods depend less on the attributes. However, LCOM could be controversial because a class may have methods that depend only on one class field (*i.e.,* an accessor), and accessors increase the lack of cohesion metric. For this reason, we use three additional metrics:

- *Number of used attributes* – It is the number of attributes that the extracted method used.
- *Number of internal method calls* – It is the number of calls performed by the extracted method to other methods within its class.
- *Static methods* – We count how many extracted methods are static. A static method does not depend on the instance variables of the class and therefore we consider this fact as a metric to better characterize the cohesion.

Re-Use & Code Clones. We measure the degree of which an extracted method helps reduce the number of code clones and favor code reuse. In particular, we measure the following metrics:

- *Number of Internal Callers* – We statically count the number of times that an extracted method is called within the class. For this, we only consider method calls that are performed over the `this` keyword, and have the same signature as the extracted method.
- *Code Clones* – We count the number of code clones that exist in the refactored class before and after the Extract Method refactoring. For this, we use the *Open Static Analyzer* tool[2], which detects syntax based code clones, also called Type-2 clones. Although, there are other tools that are useful to detect different kind of code clones, these tools need the compiled version of each program. Compiling GitHub projects is challenging, mainly because not all projects of provide their dependencies or build mechanisms [12].

Visibility. We count the number of extracted methods that are public, private, protected and static. We contrast this information with how many times these methods are called inside and outside the class. Since there may be several methods with the same signature as the extracted method in the system, to rigorously compute this metric we need to determine the type of receiver. However, inferring types is challenging and sometimes even impossible without a compilation process (which is extremely difficult to run over a high number of projects, as we do). One of the major reasons of the difficulties to infer types is that to do the inference accurately one also needs to analyze all the dependencies of the GitHub projects, and such dependencies are not always available or explicit. Since, our

[2] https://github.com/sed-inf-u-szeged/OpenStaticAnalyzer.

goal is to detect over-exposed extracted methods, which are methods that are public, but they are only used inside their class [24]. We compute the following metrics:

- *Internal Calls* – the number of method calls inside the refactored class that calls to a method with the same signature that the extracted method and the receiver the `this` keyword.
- *External Calls* – the number of method calls outside the refactored class that have the same signature as the refactored method. Here, we do not consider the receiver, if there is a potential call to the extracted method outside the refactored class then we consider that the extracted method is not overexposed.

4.4 Comparing Metrics

We compute the previously mentioned metrics in the refactored method and class before and after the refactoring. For this, we search each class and refactored method in the software version after the refactoring. This search was done by looking at the file corresponding to the refactored class, the class name and the method signature of the refactored method. However, in 347 cases, we could not identify the class or the refactored method in the new version after the refactoring. Since, some classes or methods were also renamed in the next version. For this reason, we focus on measuring the impact of Extract Method refactoring, and only keep the 2,712 remaining instances in which we were able to compare the metrics before and after the refactoring.

In particular, we contrast the metric values and then we detect if the metric: increases, decreases or remains the same after the refactoring. We then manually review the refactoring where the metrics reports a contradictory result. For instance, the Extract Method refactoring helps reduce the code complexity of the extracted method, therefore, if we detect the result is contrary to what one would expect, we manually contrast the change to understand these situations.

4.5 Statistical Analysis

We complete our results with a statistical analysis of the metrics that can be measured before and after the refactoring. A Shapiro-Wilk normality test confirms that the distribution of our metrics does not follow a normal distribution. In all cases the p-value is <0.01, so we can conclude that distributions are not normal. Therefore, for the remaining of this paper, we rely on non-parametric tests which do not make assumptions on the distribution of the data.

To observe the statistical significance of the effects of Extract Method on the metrics, we calculate a p-value using a Wilcoxon signed ranks test. It is a non-parametric statistical test suitable to compare paired data (before and after refactoring) and in summary the test aims to determine how different the two sets of paired data are from one another by focusing on the median. We perform

the test with a 99% confidence level, therefore a p-value <0.01 means that the sets are significantly different.

We also compute Cliff's δ effect size to quantify the importance of the effect of the refactoring on the metrics. It is a non-parametric effect sizes measure, which represents the degree of overlap between two distributions. It ranges from −1 (if all the selected values in the first set are larger than the ones of the second set) to +1 (if all the selected values in the first set are smaller than the second set). It evaluates to zero when the two distributions are identical.

Cohen's d is more commonly used to calculate effect size and the effect are usually categorized as small, medium or large, however the accuracy relies of Cohen's d relies on normality. Fortunately, Cohen's d interpretation of results can be mapped to Cliff's δ : $0 \leqslant$ negligible < 0.147, $0.147 \leqslant$ small < 0.33, $0.33 \leqslant$ medium < 0.474 or $0.474 \leqslant$ large. This labeling is useful for comparisons, however it should be noted that this labeling was tuned for social science, and that for some fields of research most effects observed are likely to be small [23]. We also perform this test with a 99% confidence level.

5 Results

All the results of our statistical tests are presented in Table 2. For all the metrics, except Number of Parameters and Input/Output Variables, the Wilcoxon signed-rank test shows a statistically significant difference. The effect size confirms these results for the complexity metrics, however it is more marginal for LCOM. We describe the results in more detail in the rest of this section.

Table 2. Results of Wilcoxon signed-rank test and Cliff's δ on all metrics, a p-value <0.01 is statistically significant while an effect size>0.147 is a visible effect

	McClure	McCabe	Number of comparisons	Number of control vars	NBD	IO variables	Number of parameters	LCOM
p-value	**<0.01**	**<0.01**	**<0.01**	**<0.01**	**<0.01**	0.052	0.035	**<0.01**
Cliff's δ	**0.300**	**0.262**	**0.290**	**0.269**	**0.274**	−0.002	−0.002	−0.054

5.1 Complexity

Figure 1 details how many times the complexity metrics of the refactored methods increase, decrease, or remain the same. Raw values and percentages are reported in Table 3. Figure 1 shows that the complexity of most of the methods decrease or remain the same. The Extract Method refactoring mostly benefit metrics McCabe, McClure's complexity, number of comparisons, number of control variables, and nested block depth as we could observe in Table 2. In particular, 1,391 methods reduce their McCabe complexity (51%). However, 194 (7%) of

them increase their complexity. We can also see that there is a considerable portion (42%) where the complexity of the refactored method remains the same. This means that the extracted code portion has no control-flow structures, and therefore the extracted methods have a low complexity. Number of Parameters and Input/Output Variable remain the same in most of the cases, therefore explaining the results of Table 2 where no significant difference is observed.

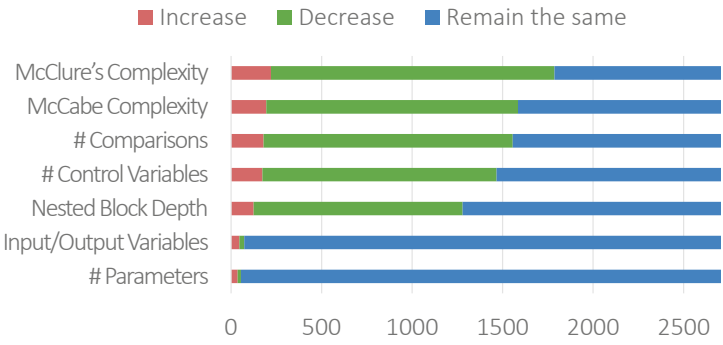

Fig. 1. Method complexity before and after a extract method refactoring

Table 3. Detailed results of complexity metrics

	McClure	McCabe	Number of comparisons	Number of control vars	NBD	I/O Variables	Number of parameters
Increase	220 (8%)	194 (7%)	179 (7%)	173 (6%)	125 (5%)	46 (2%)	36 (1%)
Decrease	1566 (58%)	1391 (51%)	1377 (51%)	1294 (48%)	1154 (43%)	27 (1%)	19 (1%)
Remain	926 (34%)	1127 (42%)	1156 (43%)	1245 (46%)	1433(53%)	2639 (97%)	2657 (98%)

We carefully performed a manual revision of the refactored methods that increase the complexity, and conclude that these methods increase their complexity because of additional changes besides the Extract Method refactoring. For instance, consider the method modification shown in Fig. 2 where the `addStepPanel` method was extracted from `initTitleAndContent` method. The extracted method is called twice after the extraction, one of these method calls was done inside an `if` control structure, which was added in the same commit as the extraction. For 61% of the methods that increase their complexity, these additional changes were related with the recently extracted method, for instance, conditionally calling to the extracted method. The remaining 39% of the methods represent additional changes which are not related to the Extract Method.

For comparison, we also used the "most metrics improved" proposed by Chavez *et al.* [6]. We found that 1249 of the 2712 (46%) Extract Method improve at least 4 complexity metrics, close to the 45% they observed.

Observation 1. Our results confirm that most of the Extract Method refactorings help reduce the complexity of the refactored method and therefore of the class. Also, 42% of the extracted methods have a low complexity.

5.2 Cohesion

In our dataset of 2,712 Extract Method refactoring, 286 Extract Methods were performed in classes which do not have any attribute, and therefore it is not possible to compute LCOM in theses cases since LCOM depends of the number of attributes. Hence, the following results are computed for the remaining 2,426 instances. We found that 1,707 (70%) classes increase the lack of cohesion, essentially because the recently extracted methods do not depend or depend on few attributes of methods of its host class. Although the difference is statistically significant, overall the effect on the value of LCOM is quite small as observed in Table 2. Indeed there is also cases where LCOM decrease, and the median of both distribution only goes from 0.90 to 0.91.

Figure 3 shows the distribution of the number of attributes used by the extracted method. It shows that 1,074 (40%) extracted methods do not depend on any attribute of their class. This affects the metric as we explained earlier. Figure 4 gives the distribution of the number of internal method calls performed by the extracted method. Where zero means that the extracted method does

```
61  +     private void addStepPanel(Step step) {
62  +         StepPanel stepPanel = new StepPanel(session, step, previousStep);
63  +         content.getChildren().add(stepPanel);
64  +         stepPanels.add(stepPanel);
65  +         previousStep = Optional.of(stepPanel);
66  +     }
67  +
68          private void initTitleAndContent() {
    -         setText(scenario.getName());
    -         Optional<StepPanel> previousStep = Optional.empty();
69  +         setText(scenario.getName());
70  +         if (!scenario.isBackgroundDone()) {
71  +             for (Step step : scenario.getBackgroundSteps()) {
72  +                 addStepPanel(step);
73  +             }
74  +         }
75              for (Step step : scenario.getSteps()) {
    -             StepPanel stepPanel = new StepPanel(session, step, previousStep);
    -             content.getChildren().add(stepPanel);
    -             stepPanels.add(stepPanel);
    -             previousStep = Optional.of(stepPanel);
    -         }
76  +             addStepPanel(step);
77  +         }
78      }
```

Fig. 2. Extract method refactoring in the karate GitHub project

not have any single call to a method of its class. In total, 49% of the extracted methods do not depend on the other methods of the class. In addition, we have 506 (19%) methods that do not depend on any attributes and/or methods of the class. These methods normally are declared static, however, we found that 10% of these methods does not depend on the host class and may be easily converted to static or move to another class where they may be more cohesive.

Observation 2. 70% of the extracted method slightly reduce the class cohesion, confirming the trend observed in previous studies. In addition, we found that 10% of these methods are not static and do not depend on any attribute/method of their class.

5.3 Re-Use and Code Clones

Code Clones. We analyze the number of code clones in the refactored class before and after each refactoring. Since, the code clone metric vary depending on a threshold, the minimum number of code lines in the clone, we compute the number of code clones using five thresholds (1 to 5). The idea is to assess whether this threshold may have an impact on the evolution of code clone. Figure 6 shows the proportion of increased, decreased and equal number of code clones remains.

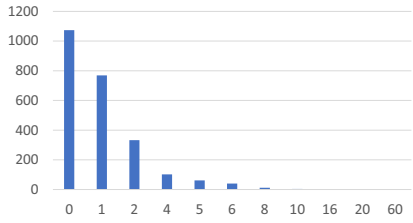

Fig. 3. Extracted method - Number of used attributes

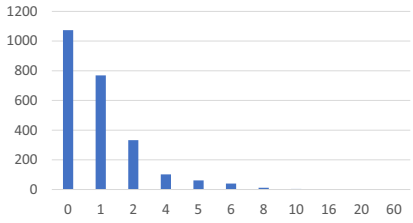

Fig. 4. Extracted method - Number of internal method calls

As one would expect, the number of code clone decreases as the value of the threshold increases. However, the proportion of increase, decrease and remain is similar for all thresholds. For increase, the proportion varies between 23% and 24%, while it is between 55% and 58% for decrease and between 19% and 21% remain the same. For example, using the a threshold of four, we found that from the 2,712 refactorings 818 (30%) of the classes have code clones. In these classes, 473 (58%) of the Extract Method refactorings help reduce the code clones in the refactored class. However, 19% of the code clones remains in the class and there are 23% of the refactored classes that increase the number of code clones in the same commit.

The results of Wilcoxon signed-rank test and Cliff's δ for all the thresholds are presented in Table 4. There is statistical significance in all the cases, however

Table 4. Wilcoxon signed-rank test and Cliff's δ for the thresholds of code clone

Threshold	1	2	3	4	5
p-value	**<0.01**	**<0.01**	**<0.01**	**<0.01**	**<0.01**
Cliff's δ	0.132	0.131	0.125	0.114	0.087

the effect size is arguably small, although close to 0.137 for the firsts thresholds. It decreases with every thresholds since less code clones are found.

Figure 5 illustrate how Extract Method can affect code clones. Consider the method modification done where the method `action` was extracted from the method `actionPrimary`. The recently extracted method was called in two different places, in the source method at line 63 and 69. The method call in line 63 was done as a result of the Extract Method refactoring, where a set of line were replaced by this new method call. However, an additional method call was inserted in line 69, this additional change is not related to the Extract Method refactoring. Furthermore, a method call to `setPrimary` was added before both method calls and duplicate code.

Observation 3. Around 56% of the extracted methods reduce the code clones in the refactored class, and around 20% of the classes remain with code clones after the Extract Method.

```
60      @Override
61      public void actionPrimary(Vector3f point, int textureIndex, AbstractSceneExplorerNode ro
        -       if (radius == 0 || weight == 0)
        -           return;
        -       RaiseTerrainToolAction action = new RaiseTerrainToolAction(point, radius, weight, ge
        -       action.doActionPerformed(rootNode, dataObject);
62      +       setPrimary(true);
63      +       action(point, textureIndex, rootNode, dataObject);
64      }
65
66      @Override
67      public void actionSecondary(Vector3f point, int textureIndex, AbstractSceneExplorerNode
        -       // no secondary option
68      +       setPrimary(false);
69      +       action(point, textureIndex, rootNode, dataObject);
70      }
71
72      +   private void action(Vector3f point, int textureIndex, AbstractSceneExplorerNode rootNode
73      +       if (radius == 0 || weight == 0)
74      +           return;
75      +
76      +       if (!modifying)
77      +           modifying = true;
78      +
```

Fig. 5. Code clones before and after a extract method refactoring

Re-use. When a piece of code is extracted to a new method, this method may be re-used in the system. We count how many times the extracted method is called inside the refactored class. Figure 7 gives the distribution of the numbers of times that an extracted method is called inside their class. 51% of the extracted methods are called more than once, meaning that these extracted methods are reused at least once. However, 49% of the extracted methods are called only once and therefore were not reused.

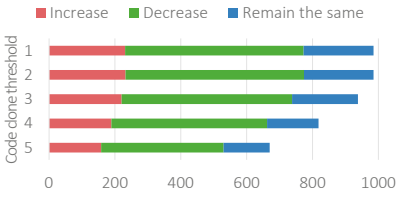

Fig. 6. Number of code clones

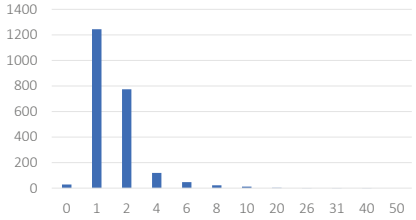

Fig. 7. Number of internal calls to the recently extracted method

Observation 5. 51% of the extracted methods are called more than once, thus favoring code reuse.

5.4 Visibility

One could expect a large majority of extracted methods to be private. However, We found that only 54% of the extracted methods are private, and 33% are public, the remaining are protected and package. We automatically review how many of the public extracted methods are reused from other classes besides the host class. We found that 22% of the public methods are only used inside their class, therefore we consider these methods as overexposed [24], since it is not necessary for them to be public.

Observation 6. 54% of the extracted methods are private, and 22% of the public extracted methods are overexposed and may reduce their visibility scope.

6 Threats to Validity

Construct Validity. We use refactoring miner to detect extract method refactoring instances along software versions. Although refactoring miner has a high precision and recall, it may still report false negatives and positives. In particular, for the Extract Method refactorings, Tsantalis *et al.* reported a precision of 98.63% [22]. We were able to manually analyze and confirm all the extracted

methods we considered in our study. However, there is still a chance that we miss a number of extract method refactorings since studies on the Refactoring Miner report a recall of 84.72%. In addition, we measure the complexity metrics using *Jasome*, an open source project[3]. Therefore, the precision of our analysis is related to the precision of this tool. To verify the precision of this tool, we manually review a random sample of 200 extracted methods and all indicate that this tool computes the metrics accurately.

Internal Validity. A software version may contain different software changes in addition to refactorings. As a consequence, the metrics considered in this study may vary due to the additional code changes in the same class besides the refactoring. Section 5 gives a number of examples of this situation. Therefore, our study as well as the previous work are subject to this threat to validity.

External Validity. Our study only includes open-source projects for obvious accessibility reasons, hence we cannot generalize the results to industrial projects. In addition, we focus on the Java programming languages, therefore our findings are valid for Java.

Conclusion Validity. We believe that our sample considers a great variety of projects. However, our findings might be different for other dataset. We were careful not to violate the assumptions of the performed statistical tests. After observing that the distributions of our metrics were not normal, we only used non-parametric tests that do not require making assumptions about the distribution of the metrics. Concerning effect size, we relied as much as possible on the standard labeling of small, medium and large. However, our interpretations are not strictly limited to this labeling, as long as the Wilcoxon signed-rank test was significant. As mentioned earlier, this labeling was tuned for social science [23] and it is quite possible that it may not be fully adapted to our needs.

7 Conclusion

In this paper, we investigate the refactoring effects of Extract Method on quality. At the difference of previous studies that consider a great variety of refactoring, we focus on the Extract Method refactoring in order to provide a more tailored analysis and detailed results. We conducted an in-depth analysis by considering additional metrics which are related to this particular refactoring, such as, code clones, code re-usability and method visibility. Furthermore, we consider a large set of projects and analyze 2,713 Extract Method refactorings.

Our results are comparable to previous works in term of complexity and cohesion. We found that most of the Extract Method refactorings help reduce the complexity, however, they also tend to slightly reduce the class cohesion. This fact is mainly because the extracted method depends on a few (or none) attributes and methods of its class. In terms of reuse and code clones, we found that 56% of Extract Method refactoring helps reduce the number of code clones,

[3] https://github.com/rodhilton/jasome.

and 51% of the extracted methods are called more than once, favoring code reuse. Although, previous studies analyze the impact of refactorings on code smells, they do not consider code clones. Finally, we found that 22% of the extracted methods are overexposed, therefore, they may easily reduce its scope to private. Thus this study completes and nuances the results of previous research in this topic. As future work, we plan to replicate our experiment in other programming languages and focus on other refactorings, allowing us to adapt the metrics and interpret the results on a case-by-case basis.

Acknowledgments. Bergel thanks ANID Fondecyt 1200067 for partially sponsoring this work. Hecht is sponsored by ANID/FONDECYT Postdoctorado N°318056.

References

1. Al Dallal, J., Abdin, A.: Empirical evaluation of the impact of object-oriented code refactoring on quality attributes: a systematic literature review. IEEE Trans. Softw. Eng. **44**(1), 44–69 (2017)
2. AlOmar, E., Mkaouer, M.W., Ouni, A.: Can refactoring be self-affirmed? an exploratory study on how developers document their refactoring activities in commit messages. In: International Workshop on Refactoring, pp. 51–58. IEEE (2019)
3. AlOmar, E.A., Mkaouer, M.W., Ouni, A., Kessentini, M.: On the impact of refactoring on the relationship between quality attributes and design metrics. In: Proceedings of the ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), pp. 1–11. IEEE (2019)
4. Bavota, G., De Lucia, A., Di Penta, M., Oliveto, R., Palomba, F.: An experimental investigation on the innate relationship between quality and refactoring. J. Syst. Softw. **107**, 1–14 (2015)
5. Charalampidou, S., Ampatzoglou, A., Chatzigeorgiou, A., Gkortzis, A., Avgeriou, P.: Identifying extract method refactoring opportunities based on functional relevance. IEEE Trans. Software Eng. **43**(10), 954–974 (2016)
6. Chávez, A., Ferreira, I., Fernandes, E., Cedrim, D., Garcia, A.: How does refactoring affect internal quality attributes? a multi-project study. In: Proceedings of the 31st Brazilian Symposium on Software Engineering, pp. 74–83 (2017)
7. Choi, E., Yoshida, N., Inoue, K.: An investigation into the characteristics of merged code clones during software evolution. IEICE Trans. Inf. Syst. **97**(5), 1244–1253 (2014)
8. Elish, K.O., Alshayeb, M.: A classification of refactoring methods based on software quality attributes. Arab J. Sci. Eng. **36**, 1253–1267 (2011)
9. Fernandez-Sanz, L., Medina Merodio, J.A., Gómez Pérez, J., Misra, S.: Analysis of expectations of students and their initial concepts on software quality, pp. 284–288 (2016)
10. Henderson-Sellers, B.: Object-Oriented Metrics: Measures of Complexity. Prentice-Hall Inc, USA (1995)
11. Kádár, I., Hegedus, P., Ferenc, R., Gyimóthy, T.: A code refactoring dataset and its assessment regarding software maintainability. In: International Conference on Software Analysis, Evolution, and Reengineering (SANER), IEEE (2016)
12. Martins, P., Achar, R., V. Lopes, C.: 50k-c: A dataset of compilable, and compiled, java projects. In: 2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR), pp. 1–5 (2018)

13. Misra, S., Akman, I., Palacios, R.: Framework for evaluation and validation of software complexity measures. IET Softw. **6**, 323–334 (2012)
14. Misra, S., Adewumi, A., Fernandez-Sanz, L., Damasevicius, R.: A suite of object oriented cognitive complexity metrics. IEEE Access **6**, 8782–8796 (2018)
15. Misra, S., Adewumi, A., Omoregbe, N., Crawford, B.: A systematic literature review of open source software quality assessment models. SpringerPlus 1936 (2016)
16. Murphy-Hill, E., Parnin, C., Black, A.P.: How we refactor, and how we know it. IEEE Trans. Softw. Eng. **38**(1), 5–18 (2011)
17. Pantiuchina, J., Lanza, M., Bavota, G.: Improving code: The (mis) perception of quality metrics. In: Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 80–91. IEEE (2018)
18. Rodríguez, G., Esteberena, L., Mateos, C., Misra, S.: Reducing efforts in web services refactoring. In: International Conference on Computational Science and Its Applications, pp. 544–559 (2019)
19. Rodríguez, G., Mateos, C., Listorti, L., Hammer, B., Misra, S.: A novel unsupervised learning approach for assessing web services refactoring. Communications in Computer and Information Science ICCSA 2019, pp. 273–284 (10 2019)
20. Rodríguez, G., Mateos, C., Misra, S.: Exploring web service QoS estimation for web service composition. In: Lopata, A., Butkienė, R., Gudonienė, D., Sukackė, V. (eds.) ICIST 2020. CCIS, vol. 1283, pp. 171–184. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-59506-7_15
21. Sandoval Alcocer, J.P., Siles Antezana, A., Santos, G., Bergel, A.: Improving the success rate of applying the extract method refactoring. Sci. Comput. Program. **195**, 102475 (2020)
22. Tsantalis, N., Mansouri, M., Eshkevari, L.M., Mazinanian, D., Dig, D.: Accurate and efficient refactoring detection in commit history. In: Proceedings of the 40th International Conference on Software Engineering, pp. 483–494. ICSE, ACM (2018)
23. Valentine, J.C., Cooper, H.: Effect size substantive interpretation guidelines: issues in the interpretation of effect sizes. What Works Clearinghouse 1–7 (2003)
24. Vidal, S.A., Bergel, A., Marcos, C., Díaz-Pace, J.A.: Understanding and addressing exhibitionism in java empirical research about method accessibility. Empirical Softw. Eng. **21**(2), 483–516 (2016)