Assessment of the Code Refactoring Dataset Regarding the Maintainability of Methods

István Kádár, Péter Hegedűs(✉), Rudolf Ferenc, and Tibor Gyimóthy

University of Szeged, Szeged, Hungary
{ikadar,hpeter,ferenc,gyimothy}@inf.u-szeged.hu

Abstract. Code refactoring has a solid theoretical background while being used in development practice at the same time. However, previous works found controversial results on the nature of code refactoring activities in practice. Both their application context and impact on code quality needs further examination.

Our paper encourages the investigation of code refactorings in practice by providing an excessive open dataset of source code metrics and applied refactorings through several releases of 7 open-source systems. We already demonstrated the practical value of the dataset by analyzing the quality attributes of the refactored source code classes and the values of source code metrics improved by those refactorings.

In this paper, we have gone one step deeper and explored the effect of code refactorings at the level of methods. We found that similarly to class level, lower maintainability indeed triggers more code refactorings in practice at the level of methods and these refactorings significantly decrease size, coupling and clone metrics.

Keywords: Code refactoring · Software maintainability · Empirical study · Refactoring dataset

1 Introduction

Source code refactoring is a very powerful technique to improve the internal quality of software systems. Since its introduction by Fowler [6] it become more and more popular and nowadays IT practitioners think of it as an essential part of the development processes. Despite the high acceptance of refactoring techniques by the software industry, there are some aspects that software companies should take into consideration which may affect the practical application of such techniques; for example, time constraint, cost effectiveness, or return on investment. Due to this shift of priorities between industry and research, we should also explore how developers tend to use refactoring in practice and not just focus on the theoretical concepts of code refactoring. There are evidences in the literature [21] that engineers are aware of code smells, but are not very concerned with their impact as refactoring activity is not focused on them. But as Fowler et al. suggested, code smells should be the primary technique for identifying

O. Gervasi et al. (Eds.): ICCSA 2016, Part IV, LNCS 9789, pp. 610–624, 2016.
DOI: 10.1007/978-3-319-42089-9_43

refactoring opportunities in the code and a lot of research effort [4,5] has been put into examining them. A similar contradictory result by Bavota et al. [2] suggests that only 7 % of the refactoring operations actually remove the code smells from the affected class.

All these seemingly negative results only indicate that although some concepts might be very effective in theory, they may not be applied in industry due to practical reasons. So to be able to elaborate new techniques and methods that better suit the practitioners' needs we should further examine how they apply refactorings. To help addressing this goal, we proposed a publicly available refactoring dataset [11] that we assembled using the *RefFinder* [12] tool for refactoring extraction and the *SourceMeter*[1] static source code analyzer tool for source code metric calculation. The dataset consists of refactoring and source code metrics for 37 releases of 7 open-source Java systems. Every refactoring is bound to the source code elements at the level of methods and classes on which the refactoring was performed. We also store exact version and line information in the dataset that supports reproducibility. Additionally to the source code metrics, the dataset includes the relative maintainability indices of source code elements, calculated by the *QualityGate*[2] tool, an implementation of the *ColumbusQM quality model* [1]. This makes it possible to directly analyze the connection between source code maintainability and code refactoring.

Our first results in utilizing the proposed dataset [11] showed that classes with poor maintainability are subject to more refactorings in practice than classes with higher technical quality. Considering metrics, number of clone instances, complexity, coupling, and size metrics have improved, although comment related metrics decreased. In this paper, we focus on a similar empirical investigation, but not at class level, but at the level of individual methods. The literature lacks such studies on the evolution of methods in systems due to refactorings, which we can examine now by using the proposed dataset.

With the help of the assembled dataset, in this paper we examine the connection between refactorings and practical maintainability of the code by investigating the following research questions:

RQ1. *Are methods with lower maintainability subject to more refactorings in practice?*

RQ2. *Which quality attributes (source code metrics) are affected the most by refactoring methods and to what extent?*

By applying statistical methods on the refactoring data contained in our dataset we found that lower maintainability indeed triggers more code refactorings in practice at the level of methods and these refactorings significantly decrease code lines, coupling, and clone metrics.

The rest of the paper is organized as follows. In Sect. 2 we summarize the empirical works in connection with code refactorings. In Sect. 3 the process of assembling the refactoring dataset and its utilization in this paper is described.

[1] http://www.sourcemeter.com/.
[2] http://www.quality-gate.com/.

We present the results of our empirical investigations in Sect. 4. Last, we describe the threats to the validity of our research in Sect. 5 and conclude the paper in Sect. 6.

2 Related Work

In this section we present some relevant works that investigate the relationship between practical refactoring activities and software quality similarly to us. Lot of the below mentioned papers also use the RefFinder tool [12,22] to find refactorings in software systems.

Murgia et al. [17] investigated if highly coupled classes are more likely to be targeted by refactorings than less coupled ones. Classes with high fan-out (and relatively low fan-in) metric was frequently targeted by refactorings, which indicates that developers may prefer to refactor classes with high outgoing rather than high incoming coupling.

Kataoka et al. [10] also examined the coupling metrics and showed that those are quite effective in quantifying the impact of refactoring and helped them to choose the appropriate refactoring types to apply on the source code.

In contrast to the above two works, we did not apply a particular set of metrics to assess the effect of refactorings, but rather performed statistical tests to find those metrics that change significantly upon refactorings and analyzed the way they changed. We could identify that not only coupling, but size and clone metrics also play an important role when doing code refactoring.

Measuring clones and studying how refactoring affects them is also a very popular research topic. The dataset we proposed also includes clone metrics, so code clone related refactoring examinations can also be performed. Choi et al. found [3] that merged code clone token sequences and differences in token sequence lengths vary for each refactoring pattern. They also showed that extract method and replace method with method object refactorings are the most popular choices of the developers performing clone refactoring.

An automated approach presented by Wang et al. [23] recommends clones for refactoring. The built decision tree-based classifier helps the developers to determine if a clone is worth the effort to be refactored or not. The approach achieves a precision of around 80 %, and similarly good precision is achieved in cross-project evaluation. By recommending which clones are appropriate for refactoring, the approach allows better resource allocation for refactoring itself after obtaining clone detection results.

Bavota et al. [2] investigated the relationship between code smells and refactoring activities. They mined the evolution history of 2 open-source Java projects and found that refactoring operations are mainly focused on code components for which quality metrics do not suggest there might be a need for refactoring operations. Contrary to their work, by considering maintainability instead of code smells, we found significant, but not very strong connection with refactoring activities. Bavota et al. also propose a refactoring dataset with 15,008 refactoring operations, but it contains file level data only without precise line

information. Our open dataset contains method level information as well and refactoring instances are completely traceable.

The approach presented by Hoque et al. [9] investigates the refactoring activity as part of the software engineering process and not its effect on code quality. The authors found that it is not always true that there are more refactoring activities before major project release dates than after. The authors were able to confirm that software developers perform different types of refactoring operations on test code and production code, specific developers are responsible for refactorings in the project and refactoring edits are not very well tested.

In another work [7] an automatic reviewing tool was developed with the purpose of helping the code review activities by determining which changes in the change set are the results of refactorings. Correctly performed refactorings, by definition, preserve the behavior of the program so cannot introduce bugs. Thus, spending effort on reviewing refactored changes is undesirable because it is more likely that one finds bugs in non-refactoring changes.

The paper by Parsai et al. [20] proposes to adopt mutation testing as a means to verify if the behavior of the test code is preserved after refactoring. Their experiments indicate that mutation testing is suitable for identifying changes on the external behavior of a refactored test and can also be used to detect those parts of the test that was refactored improperly.

An extensive survey on the field of software refactoring is created by Mens and Tourwe [15]. Among others, refactoring activities, specific techniques and formalisms that are used for supporting these activities, important issues that need to be taken into account when building refactoring tools and the effect of refactoring on the software process were taken into consideration in this paper. One activity in the refactoring process is identifying where to apply refactorings. According to the survey, one of the most widespread approach to detect program parts that require refactoring is the identification of bad smells (especially code clones), which decrease maintainability. Our study identifies similar things in practice, because we found that methods with poor maintainability are subject to higher number of refactorings during their lifetime.

Similarly to us, Murphy-Hill et al. [18] empirically investigated how developers refactor in practice. They found that refactoring tools are rarely used: 11 % by Eclipse developers and 9 % by Mylyn developers. Unlike in this work, we do not focus on how refactorings are introduced (i.e. manually or using a tool), but on their effect on source code.

3 Approach

In order to support the further researches on source code refactorings we built a dataset of source code metrics and the applied refactorings between the releases of the investigated projects. Utilizing the data set, we investigated the effect of refactorings applied on the methods of the programs on their various metrics and quality properties.

3.1 Dataset Construction

The basic methodology of the construction of the dataset is described in our previous paper [11], here we emphasize the method-level specific details. The prepared dataset contains data of release versions of 7 open-source Java systems available on GitHub. Table 1 provides details about the projects, their names, URLs, number of analyzed releases and the covered time interval by the releases.

Table 1. The systems included in the refactoring dataset

System	Github URL	# Rel.	Time interval
antlr4	https://github.com/antlr/antlr4	5	21/01/13–22/01/15
junit	https://github.com/junit-team/junit	8	13/04/12–28/12/14
mapdb	https://github.com/jankotek/MapDB	6	01/04/13–20/06/15
mcMMO	https://github.com/mcMMO-Dev/mcMMO	5	24/06/12–29/03/14
mct	https://github.com/nasa/mct	3	30/06/12–27/09/13
oryx	https://github.com/cloudera/oryx	4	11/11/13–10/06/15
titan	https://github.com/thinkaurelius/titan	6	07/09/12–13/02/15

These projects were found ideal for our research purposes because of the adequate number of release versions and the amount of the code modifications between two adjacent releases. We selected 3 to 8 releases of each project. When selecting the releases to include in the data set, we considered the amount of code modifications between two adjacent releases of a project. As long as there is not enough code modification between two adjacent releases, the number of revealed refactorings is rather low, which can mislead statistic or machine learning algorithms. On the other hand, in order to support researches in this topic there should be a large enough number of releases which allows the investigation of the change of refactoring numbers and source code metrics in time. We found that about a half-year time interval between two releases provides sufficient amount of code modifications which proves to be appropriate for our research goals. Thus, in case of every project, we dropped those release versions that were too close to each other in time.

For every selected release version of every project, class and method level metrics and the number of refactorings grouped by refactoring types (e.g. extract method, remove parameter) are available in the dataset. The refactoring types are different in class and method level: there are 23 refactoring types on class level, and 19 on method level. For a complete list of method and class-level refactorings refer to Table 6 in the appendix. In Table 2 we provide an overview of the total number of classes, methods, and refactorings contained in the dataset.

To reveal refactorings between two adjacent release versions we used the RefFinder refactoring reconstruction tool [12]. We note that according to its authors the precision of the tool is 79 % [22]. RefFinder is implemented as an Eclipse plug-in and is able to reveal refactorings between two Eclipse projects.

Table 2. Total number of classes, methods, and refactorings in the dataset

System	# Classes	# Methods	# Refactorings
antlr4	622	5,280	248
junit	1,267	4,124	553
mapdb	850	6,180	2,973
mcMMO	505	4,767	62
mct	2,175	11,765	763
oryx	551	2,592	121
titan	2,429	14,214	3,152
Total	**8,399**	**48,922**	**7,872**

In order to avoid the manual importation of every release of every project into Eclipse and starting the analysis by hand, we extended RefFinder with batch execution support that enables the automatic analysis of all the specified releases. We also implemented an export feature which writes the found refactorings and all of their attributes into CSV files for each refactoring type which makes the later analysis of the refactorings with external tools possible.

To set up the dataset we mapped all of the refactorings to those methods that were affected by them and then counted their numbers. More specifically, if any of the attributes of a refactoring referred to a method, the refactoring was counted to that method. The reference to a method by a refactoring attribute is defined by method name, and because in Java a method name does not specify the method unambiguously, by source code position too. However, we realized that source code positions in refactoring attributes cannot be always determined precisely by RefFinder and the abstract syntax tree of Eclipse, thus there can be roughness in the mapping of refactorings to methods. In the dataset, for every release version, the accounted refactoring numbers indicate how many refactorings of various types were performed that affected the considered method between the current release and the previous one.

Besides code refactoring numbers, the dataset contains more than 50 types of static source code metrics for every method (and class) of the considered projects which were calculated using the SourceMeter static code analysis tool. Beyond these metrics we added the so-called *relative maintainability index* (RMI) which was measured by QualityGate SourceAudit for each method (and class) of the systems. RMI, similarly to the well-known maintainability index [19], reflects the maintainability of a code element, but it is calculated using dynamic thresholds from a benchmark database, not by a fixed formula. Thus, RMI expresses the maintainability of a code element compared to the maintainability of other elements in the system. The technical details of the RMI can be found in our earlier work [8].

The assembled dataset is published on the *tera-PROMISE* repository [16]: http://openscience.us/repo/refactoring/refact.html.

3.2 Data Analysis Methodology

To answer our research questions we utilized the constructed dataset in the following way. For RQ1, we did a correlation analysis between the RMI values of methods and the number of refactorings affecting these methods. In more detail, we took the maintainability indices of methods of revision x_i and the refactoring numbers of revision x_{i+1}. This way we investigated whether poor quality methods are targets of more refactoring operations or not. We performed Spearman rank correlation analysis because we cannot assume anything about the distribution of the maintainability indices nor the number of refactorings.

To answer RQ2, first we calculated the differences of the static metric values of two consecutive releases. Negative values in differences are indicating improvement, because lower metric values (e.g. lower complexity or coupling) are better in most of the cases. We performed a Mann-Whitney U test to determine whether there is a significant difference among the metric decreases in the refactored and non-refactored methods, which indicates which are those metrics that are changed significantly upon refactoring. To investigate the volume of the changes in metric values, we calculated the Cliff's delta (δ) effect size measure as well.

4 Results

In this section we summarize the assessment results on the connection between refactoring activity and maintainability of methods. First, we describe the results of the analysis on the maintainability of refactored methods to answer RQ1. Afterwards, we present the findings on the effect of refactorings on method-level source code metrics to answer RQ2.

4.1 The Maintainability of Refactored Methods

To answer RQ1, we performed a correlation analysis between the number of refactorings affecting the methods of the subject systems and their maintainability indices in the previous release (as described in Sect. 3). Figure 1 depicts the Spearman correlation coefficients between the RMI values in release x_i and the number of refactorings affecting the corresponding methods in release x_{i+1}. For the sake of easy comparison with our previous results on classes [11] we included the class-level maintainability correlations as well.

As can be seen, all the values are negative. Although the coefficients are not particularly high, they are consistently negative and significant at the level of 0.05. The negative values simply mean an inverse proportionality, namely that the worse the maintainability of a method or class is (the lower its RMI value) the more refactorings touch it (the higher the number of refactorings affecting it). There are less correlation coefficients than releases for some systems because we were unable to calculate them when RefFinder found no refactorings between two releases, which happened a couple of times. Table 3 summarizes the mean correlation coefficients both for method and class-level, their deviation

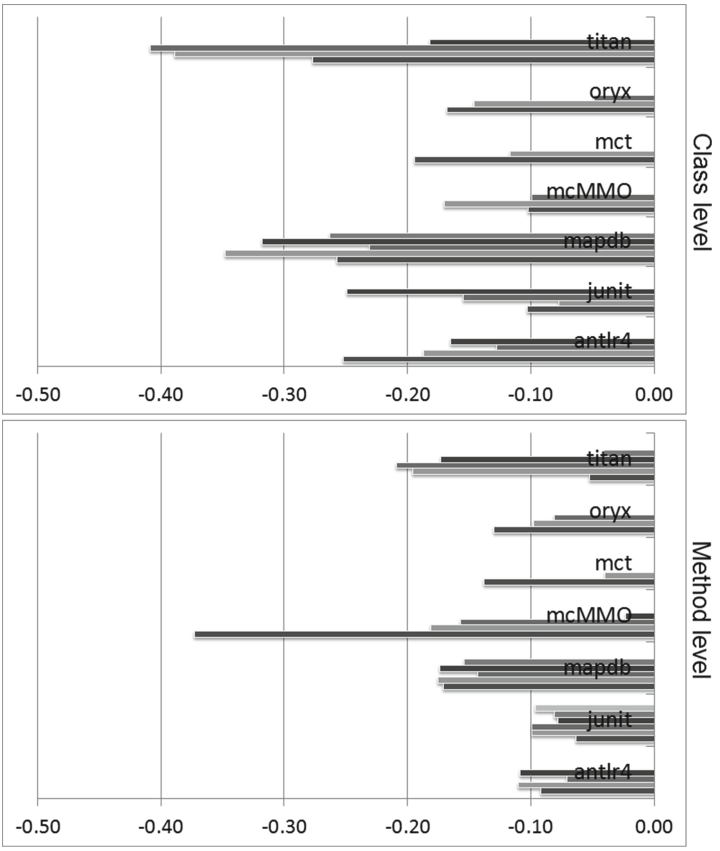

Fig. 1. Correlation of maintainability and refactorings in classes and methods

Table 3. Average Spearman correlation coefficients between RMI and number of refactorings at method and class level

System	Method level			Class level		
	Mean corr.	Deviation	Intervals	Mean corr.	Deviation	Intervals
antlr4	−0.096	0.018	4	−0.183	0.052	4
junit	−0.086	0.014	6	−0.146	0.076	4
mapdb	−0.163	0.014	5	−0.283	0.048	5
mcMMO	−0.183	0.144	4	−0.124	0.040	3
mct	−0.089	0.069	2	−0.156	0.054	2
oryx	−0.103	0.025	3	−0.121	0.063	3
titan	−0.134	0.081	5	−0.314	0.106	4

and the number of evaluated intervals between releases. It can be noticed that the correlation coefficients and the deviations are somewhat larger in the case of classes, but the differences are negligible.

Answer to RQ1: Based on the findings on our dataset we can conclude that methods with poor maintainability are subject to higher number of refactorings during their lifetime compared to those with better maintainability.

4.2 The Effect of Refactorings on Method-Level Source Code Metrics

We found that refactorings affect poorly maintainable code more (i.e. methods), so the question arises whether applying refactorings really improves the internal quality of the code? Furthermore, what are the method level source code metrics that show the highest improvement (i.e. decrease significantly) upon refactoring?

According to the process described in Sect. 3, we first calculated the metric value differences for every method between the adjacent releases. Then, we grouped these metric difference values into two groups: in the first group we put the metric differences of methods affected by at least one refactoring, and in the second group the metric differences of non-refactored methods. Finally, we analyzed which method level metrics show significant differences between the values of the two groups with the help of the Mann-Whitney U test [14].

Table 4. The results of the Mann-Whitney U Test (p-values) for method-level metrics

System name	CC	LLOC	NOS	NOI
antlr4	**0.049**	**0.000**	**0.002**	**0.001**
junit	0.058	0.923	0.667	0.403
mapdb	**0.010**	**0.003**	0.965	**0.002**
mcMMO	0.815	0.824	0.516	0.251
mct	0.703	0.924	0.547	0.660
oryx	0.654	0.555	0.306	1.000
titan	0.601	**0.016**	**0.003**	**0.000**

Out of 50+ source code metrics, the ones listed in Table 4 had the lowest p-values, meaning that the differences in the metric value changes for refactored and non-refactored methods are the most significant for these metrics. We observed that the Number of Outgoing Invocations (NOI), which can be considered as a coupling metric indeed decreases significantly upon refactoring in accordance with the previous findings of other studies [10,17].

But besides NOI, we found a significant decrease in size metrics as well, namely in the case of Logical Lines of Code (LLOC) and Number of Statements (NOS). These can be explained by the fact that typical refactorings, like extract method and pull up method, often have a side effect of reducing the amount

of source code. This phenomena is clearly observable on these pure size related metrics.

While this finding is not really surprising, the fact that McCabe's cyclomatic complexity [13] did not show a significant correlation with the number of refactorings applied on methods is just the opposite of what we were expecting. Our perception was that using better code structures will lead to less complex code, but we could not confirm this hypothesis. It is an even more interesting finding in the light of our previous results [11] on the effect of refactorings on the Weighted Method Complexity (WMC) metric of classes, which shows a significant reduction upon refactorings. However, this is not a contradiction. Consider the *Extract method* refactoring for example. In this case duplicated methods in the child classes are extracted and put into their parent class, leading to the removal of the method from several classes and inserting it to their parent. On one hand, this yields to reduction in the average WMC metric as the complexity of child classes decrease, while only the complexity of their parent class increases. On the other hand, at method level the average McCabe's complexity values do not change. So the above results might indicate that refactoring operations tend to decrease complexity at class-level, but not really at the level of methods.

It is interesting that the Clone Coverage (CC) metric also decreased, thus refactoring activity seems to remove copy-paste code parts in practice. This phenomena is similar to the code size reduction, e.g. by extracting common code snippets into a method reduces the copy-pasted code parts, too.

Table 5. Cliff δ effect size measures for method-level metrics

System name	CC	LLOC	NOS	NOI
antlr4	**0.70**	**0.63**	**0.48**	**0.71**
junit	−0.68	**0.01**	−0.08	**0.14**
mapdb	−0.34	**0.27**	0.00	**0.28**
mcMMO	**0.10**	**0.05**	**0.17**	**0.27**
mct	−0.15	−0.03	−0.18	−0.15
oryx	−0.18	−0.14	**0.31**	−0.02
titan	−0.09	**0.16**	**0.21**	**0.33**
Average	−0.09	**0.14**	**0.13**	**0.22**

To quantify the magnitude of the differences between the metric value decreases of the refactored and non-refactored methods, we calculated the Cliff's delta (δ) effect size measure. The detailed results are presented in Table 5. Cliff's δ measures how often the values in one distribution are larger than the values in a second distribution. It ranges from -1 to 1 and is linearly related to the Mann-Whitney U statistic, however it captures the direction of the difference in its sign as well. Simply speaking, if Cliff's δ is a positive number, the metric value differences (thus the metric value decreases) are higher in the refactored methods, while negative value means that the metric value differences are higher

in the non-refactored methods. The closer the δ is to $|1|$, the more values are larger in one group than the values in the other group. Generally, the Cliff's δ values are quite hectic; however, the average δ values are positive for every metric except for CC. While in case of LLOC, NOS and NOI the majority of values are positive, only two projects have positive δ values for CC. This might suggest that cloned code is decreased by other targeted changes that are not refactorings, while refactorings often have a side effect to remove code clones as well (e.g. extract method). However, this phenomenon needs further investigation.

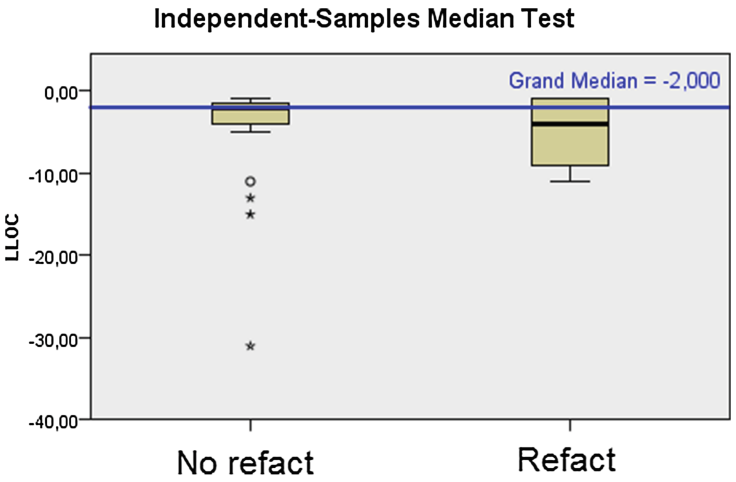

Fig. 2. Boxplot of the LLOC metric decreases in the refactored and non-refactored methods

To have a better overview of the above explained phenomena, we visualized the average size metric differences for the refactored and non-refactored methods in Fig. 2. This boxplot clearly shows that the maximum, minimum, and average numbers of code line reduction are far smaller in case of methods that are not refactored than in the case of refactored methods. While the median of LLOC decrease is 2 in case of non-refactored methods, it is two times larger (around 4) for refactored methods. Based on these findings, we can now conclude RQ2.

Answer to RQ2: We found that size (LLOC, NOS), coupling (NOI), and clone (CC) related metrics decrease the most in refactored methods. Regarding the volumes of the differences, we can say that for these metrics the average Cliff's δ values are mostly positive suggesting a small to medium effect size on the metric decreases in the refactored methods compared to the non-refactored ones.

5 Threats to Validity, Limitations

In this section, we summarize the limitations and threats to validity of our study.

First of all, we note that RefFinder, the tool we used to mine refactorings from the selected projects, is not perfect. According to its authors the precision of the

tool is 79 % [22], which means there might be false positive refactorings included in our dataset. Moreover, we have no data about recall at all; however, it is fairly possible that RefFinder does not find all of the refactorings no matter whether they were committed intentionally by the developers or not. To mitigate this threat we already started to manually validate our entire dataset and eliminate false positive instances.

Another threat occurs during the construction of the dataset when the found refactoring instances are mapped to the methods that they affect. As we described in Sect. 3.1, if any of the attributes of a refactoring matches with the name of a method and its source code position, the refactoring will be mapped to that method. Nevertheless, we noticed that the source code positions in refactoring attributes determined by RefFinder using Eclipse AST are inaccurate sometimes, which implies that in case of method overloading, where name does not necessarily identify the method, there might be roughness in the mapping and therefore in the final dataset. However, the number of such mappings is very low.

The key attribute in the dataset is the fully qualified name of the method with parameter descriptions. If a method is renamed between two consecutive releases we do not track it and its metrics, and handle it as a new method in the next release.

Finally, another threat to our results is that we investigated only seven Java systems which may not represents correctly the general characteristic of all of the software systems considering refactoring activities in practice. Therefore, we are plan to continuously extend the number of systems in our dataset.

6 Conclusions and Future Work

In this paper, we used our previously proposed public dataset, which is indented to assist the research of refactoring activities in practice, to investigate the relationship between maintainability and refactoring activities, and we also assessed how refactorings affect different source code metrics at the level of individual methods. The dataset contains fine-grained refactoring information and more than 50 types of source code metrics for 37 releases of 7 open-source systems at class and method levels.

We found that methods with poor maintainability are subject to more refactorings in practice than methods with higher technical quality. Considering concrete metrics, the clone coverage, size metrics and number of outgoing invocations decreased the most intensively in the methods subjected to frequent refactorings. This might indicate that doing code refactoring in practice indeed mitigates unwanted code characteristics such as clones, size, or coupling, and result in more maintainable software systems.

As a future work we plan to manually validate the whole dataset to get more meaningful results. We also plan to reveal more complex phenomena in connection with practical refactorings, especially the relationship between bugs and refactoring activities.

Acknowledgment. This work was partially supported by the European Union project "REPARA – Reengineering and Enabling Performance And poweR of Applications", project number: 609666.

Appendix

Table 6. The type of refactorings extracted by RefFinder at class and method level

Refactoring type	Class level	Method level
Add parameter	✓	✓
Consolidate conditional expression	✓	✓
Consolidate duplicate conditional fragments	✓	✓
Extract method	✓	✓
Inline temporary variable	✓	✓
Introduce assertion	✓	✓
Introduce explaining variable	✓	✓
Remove assignment to parameters	✓	✓
Remove parameter	✓	✓
Rename method	✓	✓
Replace magic number with constant	✓	✓
Replace method with method object	✓	✓
Inline method	✓	✓
Introduce null object	✓	✓
Remove control flag	✓	✓
Replace exception with test	✓	✓
Replace nested condition with guard clauses	✓	✓
Hide method	✓	✓
Replace temporary variable with query	✓	✓
Move field	✓	
Extract superclass	✓	
Extract interface	✓	
Introduce local extension	✓	

References

1. Bakota, T., Hegedűs, P., Körtvélyesi, P., Ferenc, R., Gyimóthy, T.: A probabilistic software quality model. In: Proceedings of the 27th IEEE International Conference on Software Maintenance (ICSM), pp. 243–252, September 2011
2. 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)

3. 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)
4. van Emden, E., Moonen, L.: Java quality assurance by detecting code smells. In: Proceedings of the 9th Working Conference on Reverse Engineering, pp. 97–106 (2002)
5. Fontana, F.A., Spinelli, S.: Impact of refactoring on quality code evaluation. In: Proceedings of the 4th Workshop on Refactoring Tools, WRT 2011, pp. 37–40. ACM, New York (2011). http://doi.acm.org/10.1145/1984732.1984741
6. Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley Longman Publishing Co., Inc., Boston (1999)
7. Ge, X., Sarkar, S., Murphy-Hill, E.: Towards refactoring-aware code review. In: Proceedings of the 7th International Workshop on Cooperative and Human Aspects of Software Engineering, CHASE 2014, pp. 99–102. ACM, New York (2014)
8. Hegedűs, P., Bakota, T., Ladányi, G., Faragó, C., Ferenc, R.: A drill-down approach for measuring maintainability at source code element level. Electron. Commun. EASST **60**, 20–29 (2013). http://journal.ub.tu-berlin.de/eceasst/article/download/852/846
9. Hoque, M.I., Ranga, V.N., Pedditi, A.R., Srinath, R., Rana, M.A.A., Islam, M.E., Somani, A.: An empirical study on refactoring activity. ACM Computing Research Repository abs/1412.6359 (2014)
10. Kataoka, Y., Imai, T., Andou, H., Fukaya, T.: A quantitative evaluation of maintainability enhancement by refactoring. In: Proceedings of the International Conference on Software Maintenance, pp. 576–585 (2002)
11. Kádár, I., Hegedűs, P., Ferenc, R., Gyimóthy, T.: A code refactoring dataset and its assessment regarding software maintainability. In: Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering. IEEE Computer Society (2016, to appear)
12. Kim, M., Gee, M., Loh, A., Rachatasumrit, N.: Ref-Finder: a refactoring reconstruction tool based on logic query templates. In: Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2010), pp. 371–372 (2010)
13. McCabe, T.J.: A complexity measure. IEEE Trans. Softw. Eng. **2**, 308–320 (1976)
14. McKnight, P.E., Najab, J.: Mann-Whitney U Test. Corsini Encyclopedia of Psychology. Wiley, New York (2010)
15. Mens, T., Tourwe, T.: A survey of software refactoring. IEEE Trans. Softw. Eng. **30**(2), 126–139 (2004)
16. Menzies, T., Krishna, R., Pryor, D.: The Promise Repository of Empirical Software Engineering Data (2015). http://openscience.us/repo
17. Murgia, A., Tonelli, R., Marchesi, M., Concas, G., Counsell, S., McFall, J., Swift, S.: Refactoring and its relationship with fan-in and fan-out: an empirical study. In: Proceedings of the 16th European Conference on Software Maintenance and Reengineering (CSMR), pp. 63–72, March 2012
18. Murphy-Hill, E., Parnin, C., Black, A.P.: How we refactor, and how we know it. IEEE Trans. Softw. Eng. **38**(1), 5–18 (2012)
19. Oman, P., Hagemeister, J.: Metrics for assessing a software system's maintainability. In: Proceedings of the International Conference on Software Maintenance, pp. 337–344. IEEE Computer Society Press (1992)
20. Parsai, A., Murgia, A., Soetens, Q.D., Demeyer, S.: Mutation testing as a safety net for test code refactoring. CoRR abs/1506.07330 (2015)

21. Peters, R., Zaidman, A.: Evaluating the lifespan of code smells using software repository mining. In: Proceedings of the 16th European Conference on Software Maintenance and Reengineering (CSMR), pp. 411–416, March 2012
22. Prete, K., Rachatasumrit, N., Sudan, N., Kim, M.: Template-based reconstruction of complex refactorings. In: IEEE International Conference on Software Maintenance (ICSM), pp. 1–10, September 2010
23. Wang, W., Godfrey, M.W.: Recommending clones for refactoring using design, context, and history. In: 2014 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 331–340. IEEE (2014)