Information and Software Technology 95 (2018) 313–327

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

Empirical evaluation of software maintainability based on a manually validated refactoring dataset

Péter Hegedűs[a], István Kádár[b], Rudolf Ferenc[*,b], Tibor Gyimóthy[b]

[a] MTA-SZTE Research Group on Artificial Intelligence, Szeged, Hungary
[b] University of Szeged, Hungary

ARTICLE INFO

Keywords:
Code refactoring
Manually validated empirical dataset
Source code metrics
Software maintainability
Empirical study

ABSTRACT

Context: Refactoring is a technique for improving the internal structure of software systems. It has a solid theoretical background while being used in development practice also. However, we lack empirical research results on the real effect of code refactoring and its application.
Objective: This paper presents a manually validated subset of a previously published dataset containing the refactorings extracted by the RefFinder tool, code metrics, and maintainability of 7 open-source systems. We found that RefFinder had around 27% overall average precision on the subject systems, thus our manually validated subset has substantial added value. Using the dataset, we studied several aspects of the refactored and non-refactored source code elements (classes and methods), like the differences in their maintainability and source code metrics.
Method: We divided the source code elements into a group containing the refactored elements and a group with non-refactored elements. We analyzed the elements' characteristics in these groups using correlation analysis, Mann–Whitney U test and effect size measures.
Results: Source code elements subjected to refactorings had significantly lower maintainability than elements not affected by refactorings. Moreover, refactored elements had significantly higher size related metrics, complexity, and coupling. Also these metrics changed more significantly in the refactored elements. The results are mostly in line with our previous findings on the not validated dataset, with the difference that clone metrics had no strong connection with refactoring.
Conclusions: Compared to the preliminary analysis using a not validated dataset, the manually validated dataset led to more significant results, which suggests that developers find targets for refactorings based on some internal quality properties of the source code, like their size, complexity or coupling, but not clone related metrics as reported in our previous studies. They do not just use these properties for identifying targets, but also control them with refactorings.

1. Introduction

Source code refactoring is a popular and powerful technique for improving the internal structure of software systems. The concept of refactoring was introduced by Fowler [1] and nowadays IT practitioners think of it as an essential part of the development process. Despite the high acceptance of refactoring techniques by the software industry, it has been shown that practitioners apply code refactoring differently than Fowler originally suggested. He proposed that code smells should be the primary technique for identifying refactoring opportunities in the code and a lot of research effort [2–5] has been put into examining them. However, there are statements in the literature [6–8] that engineers are aware of code smells, but are not really concerned on their impact as refactoring activity is not focused on them. A similar counter intuitive result by Bavota et al. [9] suggests that only 7% of the refactoring operations actually remove the code smells from the affected class. Besides exploring how, when and why refactoring is used in the everyday software development, their effects on short and long-term maintainability and costs are vaguely supported by empirical results.

To help addressing the further empirical investigations of code refactoring, we proposed a publicly available refactoring dataset [10] that we assembled using the *RefFinder* [11,12] tool for refactoring extraction and the *SourceMeter*[1] static source code analyzer tool for source code

* Corresponding author.
E-mail addresses: hpeter@inf.u-szeged.hu (P. Hegedűs), ikadar@inf.u-szeged.hu (I. Kádár), ferenc@inf.u-szeged.hu (R. Ferenc), gyimothy@inf.u-szeged.hu (T. Gyimóthy).
[1] http://www.sourcemeter.com/.

https://doi.org/10.1016/j.infsof.2017.11.012
Received 14 January 2017; Received in revised form 11 November 2017; Accepted 19 November 2017
Available online 21 November 2017
0950-5849/

metric calculation. The dataset consists of refactorings and source code metrics for 37 releases of 7 open-source Java systems. We applied the dataset for a preliminary analysis on the effects of code refactoring on source code metrics and maintainability [10,13]. After the analysis, however, it turned out that the quality of the refactoring data is quite low due to the false positive instances extracted by RefFinder, thus in this paper[2] we propose an improved dataset that is a manually validated subset of our original dataset. It contains one manually validated release for each of the 7 systems. Besides the list of true positive refactoring instances in the dataset every refactoring is also mapped 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 to supports reproducibility. Additionally to the source code metrics, the dataset includes the relative maintainability indices of source code elements, calculated by the *QualityGate*[3] tool, an implementation of the *ColumbusQM quality model* [15]. Being a direct measure of maintainability, it allows the analysis of the connection between source code maintainability and code refactoring as well.

Although the manually validated refactoring dataset is in itself a major contribution, we also utilized it to replicate and extend our preliminary studies [10,13] and re-examine the connection between maintainability and code refactoring as well as the distribution of the individual source code metrics in the refactored and non-refactored source code elements. The previous studies used the original (i.e. not validated) dataset, thus it is a question how the results change using the manually validated dataset. Our empirical investigation focused on the low-level quality attributes of refactored (and non-refactored) classes and methods, and we tried to find patterns that may explain the motivations of the developers to perform refactoring and how the internal structure of the source code elements change upon refactoring.

To concisely describe our research motivations, we framed the following research questions, which we investigated with the help of the improved dataset:

RQ1. *Are source code elements with lower maintainability subject to more refactorings in practice?*

Since refactoring is by definition a change to improve the internal code structure by preserving its functionality, it is an intuitive assumption that poor code structure is the primary driver behind code refactoring. To verify this, we investigated the maintainability values of the refactored and non-refactored source code elements to see whether there are patterns that support or contradict this assumption. By applying statistical methods on the refactoring data contained in our dataset we found that the low maintainability values of source code entities indeed triggered more code refactorings in practice.

RQ2. *What are the typical values of source code metrics of the refactored and non-refactored elements and how do they change upon refactorings?*

The first research question investigates the maintainability of the refactored and non-refactored source code elements, but we were also interested in the typical source code metric values of these elements and the effects of refactorings on these metrics. Although the RMI itself relies on source code metrics, it uses and combines only a small fraction of the available metrics (i.e. those extracted by SourceMeter). We wanted to analyze each and every metric by itself to get a deeper insight about the effect of refactorings on them. Moreover, besides the sheer metric values we were also interested in their changes throughout the releases.

Therefore, in RQ2 we examined how do the well-known source code metrics, like complexity, lines of code, coupling, etc., shape and change for the refactored and non-refactored source code elements. In general, we found that source code elements that were refactored had significantly different (typically higher) size related metrics (e.g. lines of code, number of statements), complexity (e.g. McCabe's cyclomatic complexity [16], nesting level) and coupling (e.g. coupling between object classes and number of incoming invocations) on average than source code elements not refactored at all.

Moreover, these were the metrics that changed more significantly in the refactored elements than in the non-refactored ones. Additionally, we found no such metric that would be consistently larger in the non-refactored classes and/or would grow much slower in non-refactored classes than in the refactored ones.

We also compared the findings with the previous results obtained on the not validated refactoring dataset and found that most of the metric groups found to be relevant in connection with refactoring was the same for both datasets. However, while previous results displayed 2–4 significant cases out of 7, we obtained 3–6 significant cases with much stronger p-values using the manually validated dataset. We also identified that clone related metrics had no strong connection with refactoring, even though previous results on the not validated dataset suggested so due to the false positive refactoring instances.

The main contributions of the paper can be summarized as follows. In the conference version [14] we already presented:

- A manually validated dataset containing true positive refactoring instances attached to source code elements at method and class level and their source code metrics and maintainability scores.
- An extension of the RefFinder tool that allows batch-style analysis and result reporting attached to the source code elements.
- An empirical investigation of the maintainability scores of the source code classes and methods affected by at least one refactoring and those of not.

On the basis of the achieved positive results so far, in this paper we extend our previous analysis with:

- An empirical evaluation of the main quality properties (i.e. source code metrics) and their changes due to refactoring (an entirely new research question).
- A comparison of the findings with the previous results obtained on the not validated refactoring dataset.
- Detailed information of the existing and new statistical test results and an extended discussion of them.
- We made our data analysis results available online just like the dataset itself.

The rest of the paper is organized as follows. First, we start with a related literature overview in Section 2. Next, Section 3 outlines the data collection and validation process of creating the dataset. We describe the data analysis methodology applied for answering the research questions in Section 4. In Section 5, we display the results of our empirical investigation on the maintainability and source code metrics of refactored and non-refactored source code entities. The threats to the validity of our results are listed in Section 7. Finally, we conclude the paper in Section 8.

2. Related work

There are several studies that have investigated the relationship between practical refactoring activities and the software quality through different quality attributes. Many of them used the RefFinder tool [11] to extract refactorings from real-life open-source systems, similarly as we did.

Bavota et al. [9] made observations on the relations between metrics/code smells and refactoring activities. They mined the evolution history of 2 open-source Java projects and revealed that refactoring operations are generally focused on code components for which quality metrics do not suggest there might be a need for refactoring operations.

[2] This journal paper is an extended version of our conference paper [14].
[3] http://www.quality-gate.com/.

In contrast to this work, by considering maintainability instead of code smells, we found significant and quite clear relationship with refactoring activities. Bavota et al. also provided a large refactoring dataset with 15,008 refactoring operations, but it contains file level data only without exact line information. Our open dataset contains method level information as well and refactoring instances are completely traceable.

In a similar work to ours, Murgia et al. [17] studied whether highly coupled classes are more likely to be targets of refactoring than less coupled ones. Classes with high fan-out (and relatively low fan-in) metric consistently showed to be targets of refactoring, implying that developers may prefer to refactor classes with high outgoing rather than high incoming coupling. Kataoka et al. [18] also focused on the coupling metrics to evaluate the impact of refactorings and showed that their method is effective in quantifying the impact of refactoring and helped them to choose the appropriate refactoring types.

Contrary to these two works [17,18], we did not select a particular metric to assess the effect of refactorings, but rather used statistical tests to find those metrics that change meaningfully upon refactorings. This way we could identify that complexity and size metrics also play an important role in connection with refactorings applied in practice.

Kosker et al. [19] introduced an expert system for determining candidate software classes for refactoring. They focused on the complexity measures as primary indicators for refactoring and built machine learning models that can predict whether a class should be refactored or not based on its static source code metrics. In lack of real refactoring data, they assumed that classes with decreasing complexity over the releases are the ones being refactored actively. Using this heuristic, they were able to build quite efficient prediction models.

Although it might seem that our work is very similar to that of Kosker et al., there are numerous differences. We mined and manually verified real refactoring instances instead of using heuristics to determine which classes are refactored. We also analyzed the values of static source code metrics of the refactored and non-refactored elements, but our focus was not on selecting the best predictors for building machine learning models, but to generally explore the connection between each and every metric and refactorings. Moreover, we examined 50+ metrics, which is almost the double that Kosker et al. used and also contain for example, cohesion and clone related metrics that were not examined by them. Furthermore, we applied a statistical approach instead of machine learning, and published results at the level of methods as well, not just for classes as Kosker et al.did. A significant part of our work was dedicated to the analysis of the changes in metric values that was entirely omitted by Kosker et al.

In the study conducted by Silva et al. [20] the authors monitored Java projects on GitHub and asked the developers to explain the reasons behind their decision to refactor the code. They composed a set of 44 distinct motivations of 12 refactoring types such as “Extract reusable method” or “Introduce alternative method signature” and found that refactoring activity is mainly triggered by changes in the requirements and much less by code smells. The authors also made the collected data and the tool called RefactoringMiner publicly available, which was used to detect the refactorings.

The case study by Ratzinger et al. [21] investigated the influence of refactoring activities on software defects. The authors extracted refactoring and non-refactoring related features that represent several domains such as code measures, team and co-change aspects, or complexity that served as input to build prediction models for software defects. They found that the number of software defects decreased if the number of refactorings increased in the preceding time period.

Similarly to us, Murphy-Hill et al. [22] empirically analyzed how developers refactor in practice. They found that automatic refactoring is rarely used: 11% by Eclipse developers and 9% by Mylyn developers. Unlike this paper, we did not focus on how refactorings are introduced (i.e. manually or using a tool), but rather on their effect on source code.

Negara et al. [23] conducted an empirical study considering both manual and automated refactoring. Using a continuous refactoring inference algorithm, they composed a corpus of 5371 refactoring instances collected from developers working in their natural environment. According to their findings, more than half of the refactorings were performed manually, more than one third of the refactorings performed by developers were clustered in time, and 30% of the applied refactorings did not reach the version control system.

The approach presented by Hoque et al. [24] 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.

Tsantalis et al. [25] identified that refactoring decision-making and application is often performed by individual refactoring “managers”. They found a strong alignment between refactoring activity and release dates and revealed that the development teams apply a considerable amount of refactorings during testing periods.

Measuring clones (code duplications) and investigating how refactoring affects them has also attracted a lot of research effort. Our dataset also includes clone metrics, thus clone oriented refactoring examinations can also be performed.

Choi et al. [26] identified that merged code clone token sequences and differences in token sequence lengths vary for each refactoring pattern. They found that “Extract method” and “Replace method with method object” refactorings are the most popular when developers perform clone refactoring.

Choi et al. [27] also presented an investigation of actual clone refactorings performed in open-source development. The characteristics of refactored clone pairs were also measured. From the results, they again confirmed that clone refactorings are mostly achieved by “Replace method with method object” and “Extract method”.

We found that refactoring activities are not related to clone metrics significantly in general. However, we did not distinguish our analysis based on the types of refactorings (due to the relatively small number of true positive refactoring instances), which might introduce new results for specific refactoring types that differ from the overall case.

An automated approach to recommend clones for refactoring by training a decision tree-based classifier was proposed by Wang et al. [28]. The approach achieved a precision of around 80% in recommending clone refactoring instances for each target system, and similarly good precision is achieved in cross-project evaluation. By recommending which clones are appropriate for refactoring, the approach allows for better resource allocation for refactoring itself after obtaining clone detection results.

Fowler informally linked bad code smells to refactorings and according to Beck, bad smells are structures in the code that suggest refactoring [1]. Despite that many studies showed that practitioners apply code refactoring differently, probably the most widespread approach in the literature to detect program parts that require refactoring is still the identification of bad smells.

Tourwé and Mens recommended a semi-automated approach based on logic meta programming to formally specify and detect bad smells and to propose adequate refactorings that remove these bad smells [29]. Another approach to point out structural weaknesses in object-oriented programs and solve them in an automated fashion using refactorings was proposed by Dudziak and Wolak [30]. Tahvildari and Kontogiannis proposed a framework in which a catalog of object-oriented metrics was used as indicators to automatically detect where a particular refactoring can be applied to improve software quality [31]. Szőke et al. [32] introduced a tool called FaultBuster that identifies bad code smells using static source code analysis and automatically applies algorithms to fix selected code smells by refactoring.

Table 1
Descriptive statistics of the systems included in the refactoring base dataset.

System	Git URL	# Rel.	Time interval t
antlr4	https://github.com/antlr/antlr4	5	21/01/2013–22/01/2015
junit	https://github.com/junit-team/junit	8	13/04/2012–28/12/2014
mapdb	https://github.com/jankotek/MapDB	6	01/04/2013–20/06/2015
mcMMO	https://github.com/mcMMO-Dev/mcMMO	5	24/06/2012–29/03/2014
mct	https://github.com/nasa/mct	3	30/06/2012–27/09/2013
oryx	https://github.com/cloudera/oryx	4	11/11/2013–10/06/2015
titan	https://github.com/thinkaurelius/titan	6	07/09/2012–13/02/2015

Although RefFinder can detect 63 refactoring types from Fowler's catalog and many studies used it to extract refactorings [27,33–35], there are other approaches for refactoring detection in practice. A method by Godfrey and Zou [36] identified merge, split and rename refactorings using extended origin analysis in procedural code, which served as a basis of refactoring reconstruction by matching code elements. Demeyer et al. [37] proposed an approach that compares two program versions based on a set of lightweight, object-oriented metrics such as method size, class size, and the number of method calls within a method to detect refactorings. Rysselberghe and Demeyer exploited also clone detection to detect move refactorings [38]. Xing et al. [39] presented an approach by analyzing the system evolution at the design level. They used a tool called UMLDiff to match program entities based on their name and structural similarity. However, the tool did not analyze method bodies, so it did not detect intra-method refactoring changes, such as a 'Remove Assignment To Parameter'.

The survey by Soares et al. [40] compared different approaches to detect refactorings in a pair of versions. They performed comparisons by evaluating their precision and recall in randomly selected versions of JHotDraw and Apache Common Collections. The results showed that Murphy-Hill [22] (manual analysis) performed the best, but was not as scalable as the automated approaches. Ratzinger's approach [21] is simple and fast, but it has low recall; SafeRefactor [41] is able to detect most applied refactorings, although they get low precision values in certain circumstances. According to experiments, RefFinder has a precision of around 35% and a recall of 24%, which is similar to our evaluation results.

A history querying tool called QWALKEKO [42] was also applied to the problem of detecting refactorings. The main difference between QWALKEKO and RefFinder is that RefFinder is limited to reason about two predefined versions while QWALKEKO is able to detect refactorings that happen across multiple versions. Besides the ones presented above, many other approaches exist in the literature [43–47], however, our focus is not on refactoring miner tools, but to utilize refactoring instances found by those tools to analyze their connection with software maintainability in practice.

3. Dataset construction

In order to support empirical research on source code refactorings, we built a manually validated dataset of the applied refactorings and source code metrics between two subsequent releases of 7 open-source Java systems available on GitHub. The dataset published here is the manually validated subset (from now on *improved dataset*) of the one proposed in our previous paper (from now on *base dataset*) [10]. Table 1 provides an overview of the projects, their names, URLs, number of analyzed releases and the covered time interval by the releases in the base dataset.

To reveal refactorings between two adjacent release versions we used the RefFinder [11] refactoring reconstruction tool. In order to use RefFinder to automatically extract refactorings not just between two adjacent versions of a software but between each of the versions in a given version sequence we improved RefFinder to be able to perform an automatic batch analysis. To make further examinations possible, we also implemented an export feature in RefFinder that writes the revealed refactorings and all of their attributes into CSV files for each

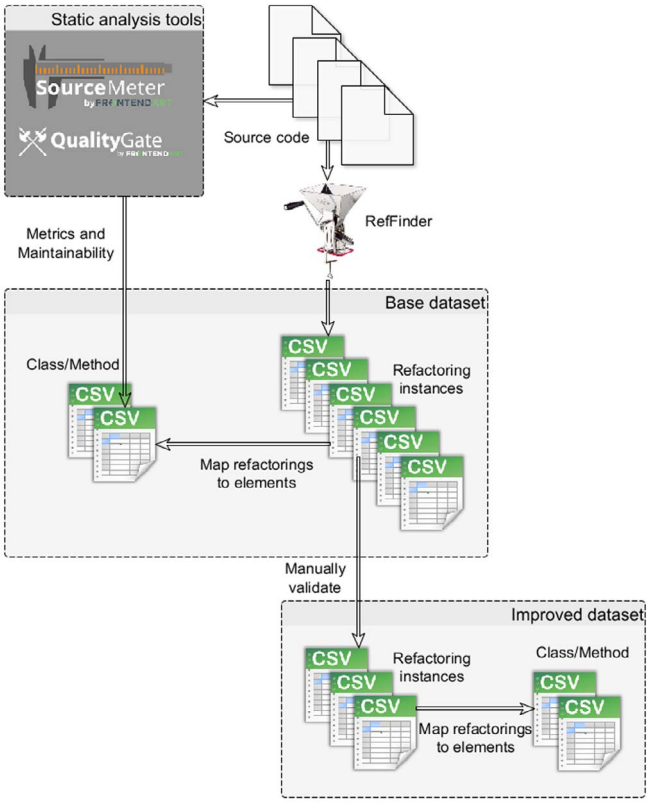

Fig. 1. An overview of the process applied for constructing the base and the improved datasets.

refactoring type.[4] The base dataset is composed of the not validated output of RefFinder grouped by refactoring types (e.g. extract method, remove parameter) and the more than 50 types of source code metrics extracted by the SourceMeter static code analysis tool, mapped to the classes and methods of the systems. The full list of extracted source code metrics is available on the tool's website.[5] Instead of selecting several metrics to analyze, we applied all the statistical methods on each of the provided metrics, which include all the most widely used code metrics.

The refactoring types are different at the class and method levels: there are 23 refactoring types at class level, and 19 at method level. For a complete list of method and class-level refactorings see our previous paper [13]. Beyond the plain source code metrics the datasets include the so-called *relative maintainability index* (RMI) which was measured by QualityGate SourceAudit [48] for each method and class of the systems. RMI, similarly to the well-known maintainability index [49], reflects the maintainability of a code element, but it is calculated using dynamic

[4] The corresponding code changes can be found in a pull-request to the original repository: https://github.com/SEAL-UCLA/Ref-Finder/pull/1.

[5] https://www.sourcemeter.com/resources/java/.

```
old method;new method;added parameter
removeAllConfigsNotInRuleStopState()
@/antlr4_base1/.../ParserATNSimulator.java:1027:1;
removeAllConfigsNotInRuleStopState()
@/antlr4_base2/.../ParserATNSimulator.java:846:1;
boolean:lookToEndOfRule
```

Listing 1:. Sample Add Parameter output of RefFinder.

Table 2
Number of all and manually validated refactorings in each subject system with precision information.

System	# All	Release	# Eval.	TP	FP	Prec.
antlr4	269	30/06/2013 [3468a5f]	112	50	62	44.64%
junit	1080	08/04/2010 [a30e87b]	29	14	15	48.28%
mapdb	4547	30/07/2014 [967d502]	171	4	167	2.34%
mcMMO	448	11/07/2013 [4a5307f]	63	6	57	9.52%
mct	716	27/09/2013 [f2cdf00]	97	28	69	28.87%
oryx	123	11/04/2014 [0734897]	71	25	46	35.21%
titan	3661	13/02/2015 [fb74209]	84	18	66	21.43%
Total	**10,844**	–	**627**	**145**	**482**	**23.13%**

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 benchmark [50].

The high-level overview of the dataset creation process is shown in Fig. 1. First of all, the Java source code is processed by the extended RefFinder version that reveals and exports the refactoring instances to CSV files for each refactoring type. The source code is also analyzed by the SourceMeter and QualityGate tools to calculate the source code metrics and RMI values for each method and class of the input project. The base dataset is assembled by mapping the extracted refactorings to the affected code elements and extending the output of static code analysis with the number of refactorings for each type that is mapped to the element.

To compose the improved dataset we performed a manual validation that resulted in a subset of the refactoring instances detected by RefFinder (i.e. we left only the true positive instances), then we mapped this validated subset of refactoring instances to the code elements again.

The datasets are available in the *PROMISE* data repository [51]:
http://openscience.us/repo/refactoring/refact.html
http://openscience.us/repo/refactoring/refact_val.html
and also at the following location:
http://www.inf.u-szeged.hu/~ferenc/papers/RefactDataSet

3.1. Dataset validation

As false positive instances may seriously affect the validity of empirical investigations using the dataset, we decided to manually validate the refactoring instances extracted by RefFinder and propose an improved dataset. The raw output of the tool enumerates all the source code elements with path and line information in the source code versions before and after refactoring.

For example, if the tool reports an “Add Parameter” refactoring, the output contains the old method to which a new parameter is being added and the same method in the new version (where the new parameter is already present in the source code) as well as the name of the parameter that was added. Listing 1 shows a simplified “Add Parameter” sample output of RefFinder for the *antlr4* system.

The evaluation process consisted of the following steps for each such refactoring instance selected for manual validation:

1. We located the files affected by the actual refactoring instance both in the old and new versions.
2. We opened the old and new files in a diff viewer tool (Araxis[6] or WinMerge[7]) and located the source code elements enumerated in the refactoring instance by the reported line numbers and identifier names.
3. By doing a line-by-line code inspection the evaluators decided whether the reported instance is a true refactoring or not (adhering to the definition of the individual refactorings and comprehending the semantic meaning of the reviewed code parts).

Since it requires an enormous amount of human effort, we started by selecting one release from each of the 7 systems and validated every refactoring instance candidate proposed by RefFinder following the above described process. The releases were selected to contain as many different types of refactorings as possible. We also kept in mind that the number of refactorings within each type has to be large enough in the releases given that some of them will be marked as false positives. We did not choose releases with huge amount of refactorings due to the necessity of an enormous validation effort.

Note, that we made a compromise in selecting the refactoring instances for validation. We chose to evaluate all instances between two selected releases for each of our subject systems. This resulted in an uneven proportion of validated refactorings from system to system (e.g. we evaluated almost 58% of refactoring instances for *oryx*, but only about 2% for *titan*), see Table 2. Moreover, there are refactoring types from which we did not evaluate a single instance, Table 3 lists only those refactoring types that were encountered during manual validation (RefFinder is able to extract 23 different types of refactorings [13]). The reason why we did this contrary to choosing for example, a fixed *x%* of refactoring instances for evaluation, is that it would not allow us to answer our research questions meaningfully. Validating a fixed proportion of refactorings for each system would not ensure a fully validated release for each system, instead we would end up with releases containing refactoring instances from a couple of which are manually validated and the rest are not. Analysis on such a dataset would be by

[6] https://www.araxis.com/.
[7] http://winmerge.org/.

Table 3
Total number of refactoring occurrences in the improved dataset grouped by their types.

Refactoring Type	antlr4	junit	mapdb	mcMMO	mct	oryx	titan	Total
Add Parameter	22	2	0	1	11	1	2	**39**
Remove Parameter	2	0	0	0	4	18	5	**29**
Introduce Explaining Variable	6	0	2	0	3	4	2	**17**
Extract Method	4	4	0	2	0	0	0	**10**
Introduce Assertion	2	1	0	0	3	0	4	**10**
Rename Method	0	2	0	1	2	0	4	**9**
Replace Method with Method Object	8	0	0	0	0	0	1	**9**
Inline Temp	0	1	0	1	4	1	0	**7**
Move Method	3	2	1	0	0	0	0	**6**
Move Field	2	1	0	0	0	0	0	**3**
Extract Interface	0	0	1	0	1	0	0	**2**
Inline Method	0	1	0	1	0	0	0	**2**
Remove Assignment to Parameters	1	0	0	0	0	0	0	**1**
Replace Magic Number with Constants	0	0	0	0	0	1	0	**1**
Total	**50**	**14**	**4**	**6**	**28**	**25**	**18**	**145**

no means more precise than using the base dataset, as the unvalidated instances might bias the statistical tests performed on the data between two releases of a system. As the manually validated subset of refactoring instances for analyzing our research questions is meaningful only if we have at least one fully validated release for each system, we made this compromise.

The validation was carried out by two of the authors of this paper. Unfortunately, performing the evaluation in an optimal way, namely to examine all the possible refactoring instances by both of the authors, was not feasible due to our available resources. Instead, the authors distributed the refactorings between them nearly equally and they validated only their corresponding instances. This strategy reduced the amount of required human resources to half of the optimal strategy; however, it also introduced some issues. To mitigate the possible inconsistency in the judgment of the two authors, they performed a random sample cross-validation on about 10% of each other's data. Additionally, in each and every problematic case all the authors of the paper (not just the two evaluators) mutually agreed on how those specific refactorings should be classified.

Table 2 shows the total number of refactoring instances found by RefFinder in all the releases of the systems (# All), the selected revisions for manual validation (Release), the number of manually validated instances per system (# Eval.), the number of true/false positive refactoring instances (TP and FP) and the overall precision of RefFinder on the analyzed systems (Prec.).

The evaluated release means that the refactoring instances between this and the previous release was considered for validation. As can be seen, only the fraction of the total number of refactorings has been validated (less than 6%). Even this work took more than one person month work from the two authors. However, as the overall precision of the RefFinder tool was only around 23% in total (and approximately 27% if we take the average of the system-wise precision values) on the base dataset, even these few hundred manually validated instances of the improved dataset bear a significant additional value compared to the base dataset. Considering the projects, we got the lowest precision value in case of mapdb and mcMMO resulting a relatively low number of refactorings in these projects.

Table 3 summarizes the number of various refactoring types within each subject system. As can be seen, Add and Remove Parameter are the two most frequently applied refactorings types. Together with the third most common Introduce Explaining Variable, they constitute nearly 60% of the total refactoring count. The majority of the Add Parameter refactoring is in the antlr4 system, while most of the Remove Parameter refactorings appear in oryx.

3.2. Dataset structure

The improved dataset contains one folder for each release of the analyzed systems. Within each folder there are two files (*$proj-Class.csv* and *$proj-Method.csv*) and a sub-folder containing a list of CSV (Comma Separated Values) files named by refactoring types. The CSV files with the names of refactorings (e.g. ADD_PARAMETER) lists only the true positive refactoring instances found by RefFinder and manually checked by one of the authors. The structure of these CSV files may differ based on the refactoring types, but they always contain enough information to uniquely identify the entities affected by the refactoring in the previous and actual releases (e.g. unique name, path of classes/methods, parameters or line information). The *$proj-Class.csv* and *$proj-Method.csv* files hold an accumulated result of the above.

Each line of these CSV files represents a class or method in the system (identified in the same way as in the refactoring CSVs). In the columns of the CSV, there are the source code metrics with the RMI scores and the various refactoring types. For each row we have the source code metrics calculated for this element and the number of refactorings of a certain type affecting the source code element (i.e. the source code element appears in the refactoring type CSV in an arbitrary role).

4. Data analysis methodology

To investigate our research questions we utilized the improved dataset in the following ways.

Answering RQ1: Are source code elements with lower maintainability subject to more refactorings in practice? To check if there are significant differences in the maintainability of the refactored and non-refactored elements, we took all the RMI values in release x_{i-1} for each system, where x_i is the release selected for manual validation. We formed two groups by RMI values based on the fact if a corresponding source code entity was affected by any refactorings in release x_i. So we mapped the source code entities (i.e. classes and methods) from version x_{i-1} to x_i and put all the RMI values for the entities in x_{i-1} into the not affected group if the entity had zeros in all refactoring columns in x_i, otherwise we put the RMI values of the entity into the affected group. Once we had these two groups we run a Mann–Whitney U test [52], which is a non-parametric statistical test to analyze whether the distribution of the values differ significantly between two groups. The p-value of the test helped us judging whether there is significant difference in the maintainability values between the source code entities subjected to refactoring and the entities unaffected by refactoring. Moreover, we used the mean rank values produced by the test to decide the direction of the differences, namely whether the maintainability value is lower or higher within one of the groups. To assess the volume of the differences, we calculated the so-called Cliff's δ non-parametric effect size measure as well [53].

Answering RQ2: What are the typical values of source code metrics of the refactored and non-refactored elements and how do they change upon refactorings? To analyze the differences in the metric values, we followed a very similar approach to that of answering RQ1. We formed the two groups in the same way as before, but instead of the RMI values, we run the Mann–Whitney U test on each and every source code metric of the refactored and non-refactored classes and methods. With this test we could identify those source code metrics that are significantly different in the refactored group compared to the elements unaffected by refactorings (from now on, we refer to this test as MWU^{Prev}). To study the effect of code refactoring as well, we calculated the metric differences between versions x_{i-1} and x_i and run the Mann–Whitney U test on the differences as well (from now on, we refer to this test as MWU^{Diff}). Since refactorings aim at improving the internal code structure, we were

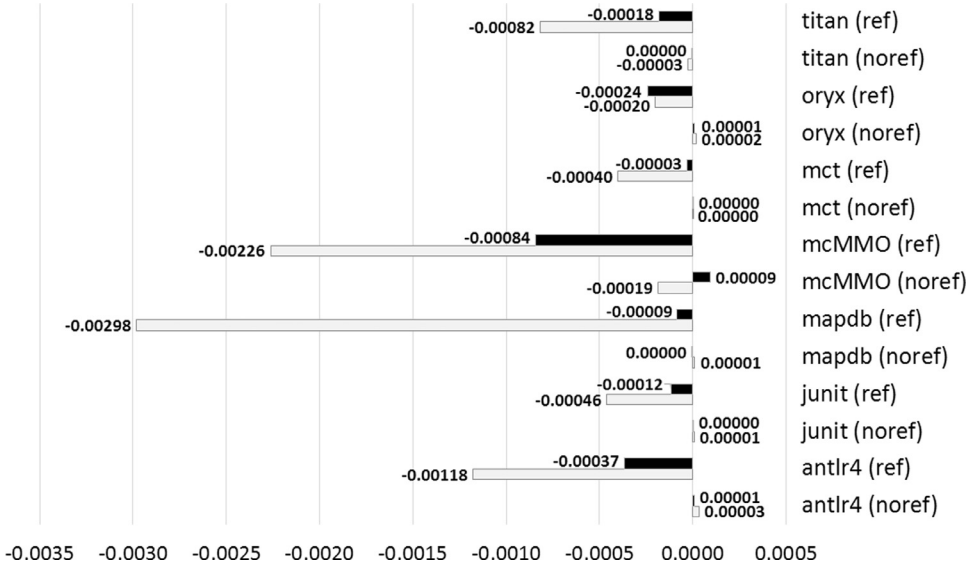

Fig. 2. Average RMI values within the refactored and non-refactored entities.

interested in whether we can observe more significant changes of the metrics in the refactored elements or not. To get a high-level overview of the most important source code metrics, we counted the number of cases (for the 7 different subject systems) when there were significant differences both in the metric values before the refactoring and in the amount of metric change after the refactoring in a source code element. We also counted the number of cases where only one of the test produced significant results (i.e. only the metric values before the refactoring or the amount of change for a metric value differed significantly in the refactored group). To assess the effect sizes in case of significant differences, we calculated the Cliff's δ measures here as well.

5. Results

5.1. Maintainability analysis

RQ1 – Are source code elements with lower maintainability subject to more refactorings in practice?

As described in the previous section, for answering RQ1 we divided the methods and classes of the systems into two groups. The first group was formed out of the entities affected by at least one refactoring between the two releases we validated. The second group contained all the other entities (i.e. the ones being untouched by refactorings between the releases). Fig. 2 depicts the average RMI values of the entities falling into these groups. Light gray columns denote the average RMI values within methods, while black color is for classes. The *ref* and *noref* marks next to the systems stand for the two groups, *ref* is the first group of entities (i.e. the ones affected by refactorings) and *noref* is the second group.

To formally evaluate whether there is a difference in the maintainability values we performed a Mann–Whitney U test on the RMI values of the two groups defined above. We executed the test on each system both for the groups of classes and methods. Tables 4 and 5 summarize the results of the test runs.

Table 4
The Mann–Whitney U test results for refactored and not refactored classes.

System	p-value	N^o noref cl.	N^o ref cl.	M. rank^{noref}	M. rank^{ref}	Cliff's δ
antlr4	**.00001**	385	23	210.61	102.26	0.53
junit	**.00628**	646	9	330.38	156.83	0.53
mapdb	**.01186**	415	4	211.46	58.37	0.73
mcMMO	.27604	85	4	45.65	31.25	0.32
mct	**.00000**	2013	15	1019.94	284.60	0.73
oryx	**.04467**	489	15	254.78	178.13	0.30
titan	**.00009**	1145	13	583.61	217.50	0.63

Table 5
The Mann–Whitney U test results for refactored and not refactored methods.

System	p-value	N^o noref mth.	N^o ref mth.	M. rank^{noref}	M. rank^{ref}	Cliff's δ
antlr4	**.00000**	3104	40	1583.10	750.16	0.53
junit	**.00466**	2253	12	1135.84	600.21	0.47
mapdb	.20610	3358	3	1681.63	973.00	0.42
mcMMO	.06529	813	5	410.69	215.40	0.48
mct	**.00346**	11,068	16	5545.88	3205.34	0.42
oryx	**.00034**	2333	19	1181.03	620.82	0.48
titan	**.00530**	7950	17	3987.32	2431.26	0.39

The main result of the test is the p-value (two-tailed) shown in the second column. This indicates whether the null-hypothesis should be rejected, which states that *there is no significant difference between the RMI values of the entities affected by refactorings in adjacent releases and the RMI values of non-refactored entities*. Thus, a p-value below .05 indicates that the hypothesis should be rejected and the alternative hypothesis should be accepted, namely that there is a significant difference in the RMI values between the two groups.

To tell something about which group has higher RMI values, thus better maintainability, we should observe the mean ranks. The column Mean rank^{noref} displays the mean ranks in the not refactored groups, while Mean rank^{ref} shows the mean rank values within the refactored groups. If the mean rank value of one group is higher, it means the RMI values in that group are significantly higher than in the other group. We report Cliff's δ values as well in the last columns, which measure how often the values in one distribution are larger than the values in another 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 maintainability values are higher in the non-refactored group, while negative value means that RMIs are higher in the refactored group. The closer the $|\delta|$ is to 1, the more values are larger in one group than the values in the other group.

5.2. Source code metrics analysis

RQ2 – What are the typical values of source code metrics of the refactored and non-refactored elements and how do they change upon refactorings?

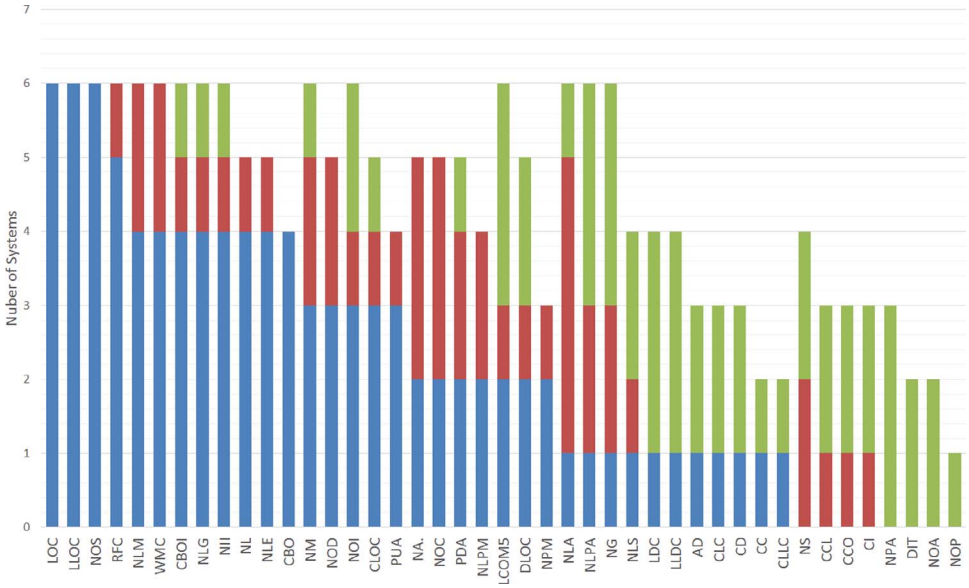

Fig. 3. High-level visualization of the Mann–Whitney U test (MWU^{Prev}) results for classes. (For interpretation of the references to color in the text, the reader is referred to the web version of this article.)

To find those properties that are significantly different of the refactored and non-refactored source code elements, we performed a Mann–Whitney U test according to the methodology described in Section 4. We used source code metrics to describe source code properties extracted by the SourceMeter static code analysis tool (the extraction process is highlighted in Section 3).

The possible differences in the distribution of source code metrics may help to shed light on the main characteristics of refactoring targets in practical development. As we saw in Section 5.1, the maintainability values are significantly lower in the source code elements targeted by refactorings. As the maintainability value is derived from source code metrics, it suggests that developers tend to (consciously or unconsciously) pay attention to some source code properties finding refactoring candidates. To get a more detailed picture of this, we analyzed all the 52 class-level and 17 method-level metrics regarding two aspects. First, how do their distributions differ within the source code elements that are refactored from one version to another and those that are not (test MWU^{Prev}). Second, how do the metric change distributions differ between the refactored and non-refactored elements from one version to another (MWU^{Diff}).

An overview of the results for classes is shown in Fig. 3. Note that the metrics with a prefix "T" (Total) are omitted from the analysis, as they are variants of the same metrics with only slight differences in their calculation (i.e. they strongly correlate with the original metrics). The height of the bars represent for how many subject systems did the statistical tests give significant results, thus their maximal value is 7, as we have 7 subject systems. The blue color means that both tests MWU^{Prev} and MWU^{Diff} gave significant results, meaning that we got a p-value less than .05, thus we could reject the null-hypothesis (i.e. that there is no significant difference in the certain metric values/the metric value changes between the groups of classes being refactored and those of not being refactored). Red color marks the number of cases where only test MWU^{Prev} resulted in significant p-value, while green color means the same for test MWU^{Diff}. That is, in case of red bars only the metric values differ significantly between the classes being refactored later and those of not, while the amount of changes in this metric between the two versions do not differ significantly. Green bars mark the opposite, where there is no significant difference in the bare metric values, but the amount of changes for that specific metric differs significantly between the refactored and non-refactored classes.

To see also the likely direction of the differences in the values as well, we collected the main characteristics of the most relevant metrics (i.e. those showing the strongest connection with refactorings) in Fig. 4. The table contains five columns for each system. The first column (avg^{Ref}) shows the average metric values in version x_{i-1} within the group of classes that are refactored in version x_i, while column two shows the average metric values in the non-refactored classes (avg^{NoRef}). Third column contains the p-values for test MWU^{Prev}. Column four (δ^{Prev}) displays the Cliff's delta effect size measures between the distribution of metric values of the refactored and non-refactored classes in version x_{i-1} (i.e. the effect size measure related to the statistical test MWU^{Prev}). It reflects how often a metric value picked randomly for a class being refactored later is larger than a randomly selected metric value for a non-refactored class. The fifth column (δ^{Diff}) presents the Cliff's delta values for the metric differences between versions x_i and x_{i-1} (i.e. the effect size measure related to the statistical test MWU^{Diff}). Missing cell values mean that the corresponding statistical test yielded no significant results (p-value is above .05), thus we could not reject the null-hypothesis (either for MWU^{Prev} and/or MWU^{Diff}).[8]

The color codes express the magnitude of the effect sizes, darker cell values (green for the metric values, blue for the metric value differences) indicate larger effect sizes.

Regarding the method level metrics, Fig. 5 contains the overview of the statistical test results run for method-level metrics. We can observe the effect size values for the most relevant metrics (i.e. those showing the strongest connection with refactorings) outlined in Fig. 6.

6. Discussion and interpretation of the results

In Section 5, we provided the detailed results of the analysis of the improved dataset performed according to the methodology described in Section 4. In this section, we discuss the findings and compare them with our previous results on the base dataset.

6.1. RQ1 – are source code elements with lower maintainability subject to more refactorings in practice?

6.1.1. Interpretation of the results for classes and methods

As can be seen in Fig. 2, all the average maintainability values of the refactored group (regardless whether for classes or methods) is much lower than the non-refactored group. What is more, almost all the average maintainability values for non-refactored classes and methods are positive (except for titan and classes of mcMMO), while the values

[8] The exact p-values for all the tests can be found in the online appendix available at http://www.inf.u-szeged.hu/~ferenc/papers/RefactDataSet/.

Metric	antlr4					junit					mapdb				
	avg^{Ref}	avg^{NoRef}	p-val.	δ^{Prev}	δ^{Diff}	avg^{Ref}	avg^{NoRef}	p-val.	δ^{Prev}	δ^{Diff}	avg^{Ref}	avg^{NoRef}	p-val.	δ^{Prev}	δ^{Diff}
LLOC	180.26	71.26	0.00001	0.54	-0.53	80.33	18.17	0.00064	0.66	-0.22	460.75	120.55	0.01212	0.73	-0.47
LOC	290.39	91.86	0.00002	0.53	-0.73	119.89	24.36	0.00067	0.66	-0.22	627.00	159.19	0.01271	0.72	-0.47
NOS	104.26	34.71	0.00013	0.47	-0.38	36.44	6.78	0.00066	0.65	-0.11	321.25	85.60	0.01566	0.70	-0.44
RFC	31.17	11.19	0.00003	0.52	-0.60	25.78	5.44	0.00043	0.68	-0.10	44.25	13.56	0.00896	0.76	0.27
NLM	15.57	6.91	0.00003	0.51	-0.27	12.33	3.28	0.00076	0.63	-0.22	28.50	7.80	0.00649	0.77	
WMC	47.35	13.35	0.00001	0.54		18.33	4.17	0.00074	0.64	-0.22	116.25	23.34	0.01072	0.74	-0.45
CBOI	7.65	4.70	0.00145	0.39	-0.17	5.67	2.71	0.04598	0.36	0.12	20.75	2.85	0.00155	0.85	0.52
NLG	4.30	1.11	0.00006	0.41	-0.17	0.67	0.25	0.24892		-0.11	3.50	0.39	0.02462	0.41	-0.24
NII	11.00	5.74	0.00017	0.45	-0.21	9.11	2.96	0.00600	0.38	-0.21	42.75	5.25	0.00180	0.80	0.52
NL	2.52	1.00	0.00001	0.50	0.09	0.89	0.30	0.00032	0.47	0.11	3.25	1.34	0.00826	0.73	0.51
NLE	2.22	0.89	0.00004	0.46		0.89	0.28	0.00030	0.48	0.11	3.25	1.20	0.00599	0.76	0.50
CBO	13.30	5.38	0.00001	0.55	-0.32	11.67	3.70	0.00109	0.63	0.11	9.00	3.77	0.07697		
NOI	15.61	4.28	0.00045	0.43	-0.47	13.44	2.16	0.00015	0.69	0.12	15.75	5.76	0.05527		0.52
CLOC	93.04	13.36	0.00069	0.41	-0.42	27.11	4.63	0.28402		0.11	90.00	17.74	0.00478	0.78	
LCOM5	2.09	1.58	0.00173	0.37		2.56	1.73	0.49541		0.11	3.50	2.07	0.05896		-0.24
	mcMMO					mct									
	avg^{Ref}	avg^{NoRef}	p-val.	δ^{Prev}	δ^{Diff}	avg^{Ref}	avg^{NoRef}	p-val.	δ^{Prev}	δ^{Diff}					
LLOC	126.75	76.27	0.12962			239.47	49.46	0.00000	0.78	-0.85					
LOC	163.75	114.01	0.18432			347.33	69.11	0.00000	0.76	-0.84					
NOS	66.00	40.49	0.10418			124.93	26.86	0.00000	0.75	-0.90					
RFC	28.75	23.05	0.66283			55.07	9.83	0.00000	0.84	-0.78					
NLM	11.00	13.47	0.53732			26.33	5.05	0.00000	0.73	-0.65					
WMC	30.00	26.99	0.38311			43.93	9.20	0.00000	0.71	-0.84					
CBOI	51.25	4.79	0.03898	0.60	-0.73	41.47	2.76	0.13808							
NLG	2.75	5.48	0.56915			7.60	1.17	0.01546	0.32	-0.26					
NII	85.50	9.79	0.02558	0.65	-0.64	47.73	3.77	0.08065		-0.17					
NL	2.75	1.51	0.11487			3.60	0.92	0.01286	0.34	-0.39					
NLE	2.50	1.34	0.07167			2.60	0.87	0.02588	0.30	-0.32					
CBO	12.25	4.64	0.64428			19.07	3.77	0.00000	0.76	-0.71					
NOI	17.75	9.58	0.43498			28.73	4.77	0.00000	0.72	-0.58					
CLOC	11.00	18.71	0.84092			52.07	10.05	0.00066	0.46	-0.64					
LCOM5	1.50	3.76	0.36053			5.80	1.36	0.00009	0.53	-0.33					
	oryx					titan									
	avg^{Ref}	avg^{NoRef}	p-val.	δ^{Prev}	δ^{Diff}	avg^{Ref}	avg^{NoRef}	p-val.	δ^{Prev}	δ^{Diff}					
LLOC	73.60	43.89	0.00419	0.43	-0.31	464.54	52.45	0.00002	0.69	-0.52					
LOC	92.73	56.19	0.00475	0.43	-0.31	633.77	70.47	0.00003	0.68	-0.59					
NOS	39.73	23.07	0.01758	0.36	-0.33	354.38	28.81	0.00006	0.65	-0.67					
RFC	17.53	9.55	0.00186	0.47		46.38	7.70	0.00095	0.53	-0.58					
NLM	6.07	4.51	0.01740	0.36		21.77	5.84	0.00035	0.57	-0.44					
WMC	13.13	8.78	0.02261	0.34		57.08	11.73	0.00004	0.66	-0.59					
CBOI	8.27	3.49	0.00251	0.44		2.08	1.94	0.30352		-0.14					
NLG	1.80	0.83	0.02342	0.28		3.23	1.39	0.03168	0.31	0.08					
NII	10.33	4.35	0.00335	0.42		1.85	1.21	0.96148							
NL	1.60	1.29	0.28548			2.54	1.34	0.00052	0.53						
NLE	1.60	1.23	0.24974			2.38	1.17	0.00050	0.53	-0.07					
CBO	7.47	3.90	0.26713			18.23	3.84	0.00015	0.60	-0.14					

Fig. 4. The average metric values, the p-values and the Cliff's delta effect sizes for the refactored and non-refactored classes. (For interpretation of the references to color in the text, the reader is referred to the web version of this article.)

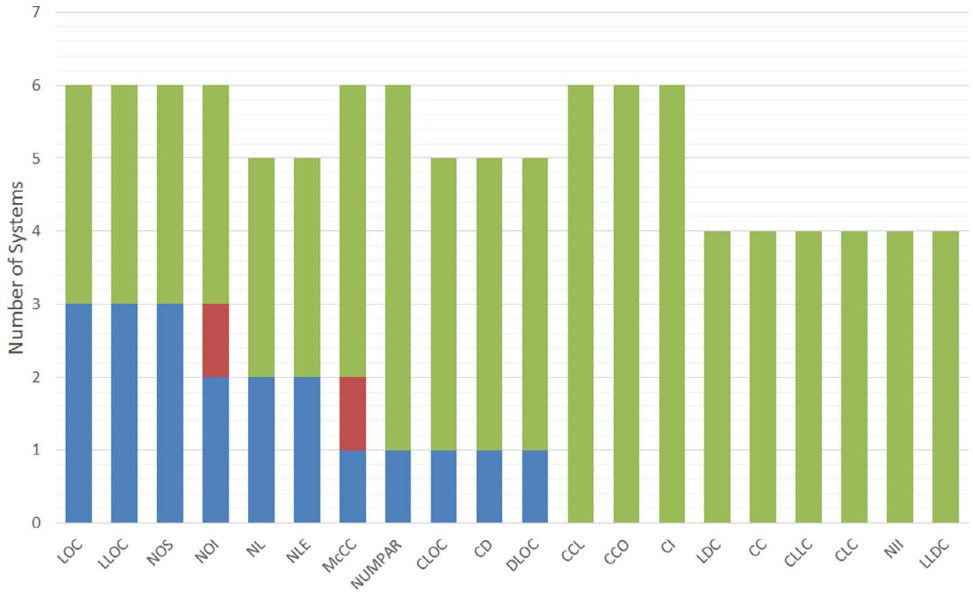

Fig. 5. High-level visualization of the Mann–Whitney U test (MWU^{Prev}) results for methods. (For interpretation of the references to color in the text, the reader is referred to the web version of this article.)

Metric	antlr4					junit					mapdb				
	avg^{Ref}	avg^{NoRef}	p-val.	δ^{Prev}	δ^{Diff}	avg^{Ref}	avg^{NoRef}	p-val.	δ^{Prev}	δ^{Diff}	avg^{Ref}	avg^{NoRef}	p-val.	δ^{Prev}	δ^{Diff}
LLOC	17.20	8.99	0.00002	0.40	-0.26	7.42	4.60	0.00059	0.56	0.17	21.33	14.76	0.30703		-0.65
LOC	21.55	9.83	0.00001	0.40	-0.29	7.58	4.70	0.00077	0.55	0.17	23.00	17.02	0.33966		-0.65
NOS	9.58	4.84	0.00648	0.24	-0.17	3.92	2.06	0.00052	0.55	0.25	14.00	10.88	0.43388		-0.65
NOI	4.03	1.56	0.00786	0.23	-0.18	3.67	1.42	0.00070	0.54		2.33	1.76	0.07744		0.34
NL	1.18	0.46	0.00000	0.36	0.12	0.42	0.19	0.00026	0.37	0.33	1.67	0.70	0.37449		-0.65
McCC	4.35	1.98	0.00000	0.43		1.75	1.27	0.00035	0.38	0.42	6.33	3.01	0.40500		-0.65
CCL	0.05	0.23	0.61937		0.10	0.00	0.02	0.66892			0.00	0.56	0.63967		0.34
CCO	0.15	0.28	0.40854		0.10	0.00	0.02	0.66892			0.00	5.48	0.64552		0.34
CI	0.05	0.23	0.62071		0.10	0.00	0.02	0.66892			0.00	1.21	0.64309		0.34
	mcMMO					mct									
	avg^{Ref}	avg^{NoRef}	p-val.	δ^{Prev}	δ^{Diff}	avg^{Ref}	avg^{NoRef}	p-val.	δ^{Prev}	δ^{Diff}					
LLOC	12.60	5.88	0.05697			12.88	8.34	0.00668	0.39	0.06					
LOC	15.20	6.88	0.06230			15.25	9.47	0.00993	0.37	-0.06					
NOS	8.00	3.32	0.05275			7.88	5.00	0.03724	0.29	-0.06					
NOI	2.80	1.09	0.14667			3.94	1.65	0.00473	0.38	0.06					
NL	0.80	0.39	0.29351			0.63	0.41	0.66898		-0.12					
McCC	3.40	2.12	0.30625			2.06	1.77	0.95825		-0.12					
CCL	2.00	0.12	0.22837		-0.21	0.06	0.27	0.17079		0.19					
CCO	13.80	0.64	0.22349		-0.22	0.06	0.67	0.21984		0.19					
CI	2.40	0.13	0.22261		-0.21	0.06	0.31	0.15128		0.19					
	oryx					titan									
	avg^{Ref}	avg^{NoRef}	p-val.	δ^{Prev}	δ^{Diff}	avg^{Ref}	avg^{NoRef}	p-val.	δ^{Prev}	δ^{Diff}					
LLOC	20.11	8.54	0.67141		-0.55	40.12	7.55	0.27779		-0.47					
LOC	23.42	9.18	0.68675		-0.55	50.24	8.35	0.30344		-0.47					
NOS	13.37	4.89	0.51504		-0.45	36.71	4.66	0.34200		-0.47					
NOI	5.47	1.71	0.58141		-0.40	4.29	0.56	0.73196		-0.06					
NL	0.63	0.53	0.38503			1.35	0.47	0.06560		-0.23					
McCC	2.89	1.90	0.41438		-0.20	6.65	1.91	0.12668		-0.41					
CCL	0.05	0.07	0.26943		-0.05	0.12	0.05	0.52819		-0.06					
CCO	0.16	0.18	0.26963		-0.05	1.47	0.13	0.49409		-0.06					
CI	0.05	0.07	0.26945		-0.05	0.18	0.06	0.50635		-0.06					

Fig. 6. The average metric values, p-values and the Cliff's delta effect sizes for the refactored and non-refactored methods. (For interpretation of the references to color in the text, the reader is referred to the web version of this article.)

related to refactored entities are negative. By the nature of its calculation, the sign of the RMI value captures the fact whether a source code element increases the overall maintainability or decreases it. Thus an RMI value of 0 has a special meaning, namely that the maintainability of a source code element (be it a class or a method) with 0 RMI is exactly as good as the system's average maintainability [50]. This implicates that positive RMI value means above average maintainability, negative RMI means maintainability below the average, which makes it intuitive that the source code entities targeted by refactorings have lower maintainability than the average.

The formal justification of this can be interpreted from the results in Tables 4 and 5. As can be seen all the p-values are well below .05 (highlighted with bold letters), except for mcMMO and mapdb at method granularity. We already showed in Table 3 that the precision of RefFinder is very low on these systems for some reason, thus the number of available true positive refactoring instances is also very low in the improved dataset. To make it even worse, for mcMMO the 6 refactorings are located in just 4 classes further reducing the number of cases (columns two and three display the number of classes/methods not affected by refactorings and those of affected, respectively). This low number of samples for one group might be the reason for the higher p-values. Nonetheless, it poses a question about the reliability of the other tests as well, as the number of samples in the two groups are highly unbalanced in most of the cases. However, the Mann–Whitney U test is designed to work well in such unbalanced sets as well [52], using the exact distribution of the small sized samples. Thus, in general, we can conclude that according to the statistical test, *there is a significant difference in the RMI values between the source code entities being refactored and the entities not affected by any refactorings.*

In each row of both tables (Tables 4 and 5), the mean ranks of refactored group are lower than the not refactored group and all the Cliff's δ values are positive, which means that the maintainability values of not refactored elements are higher, thus in general, elements targeted by refactorings typically have lower maintainability. So based

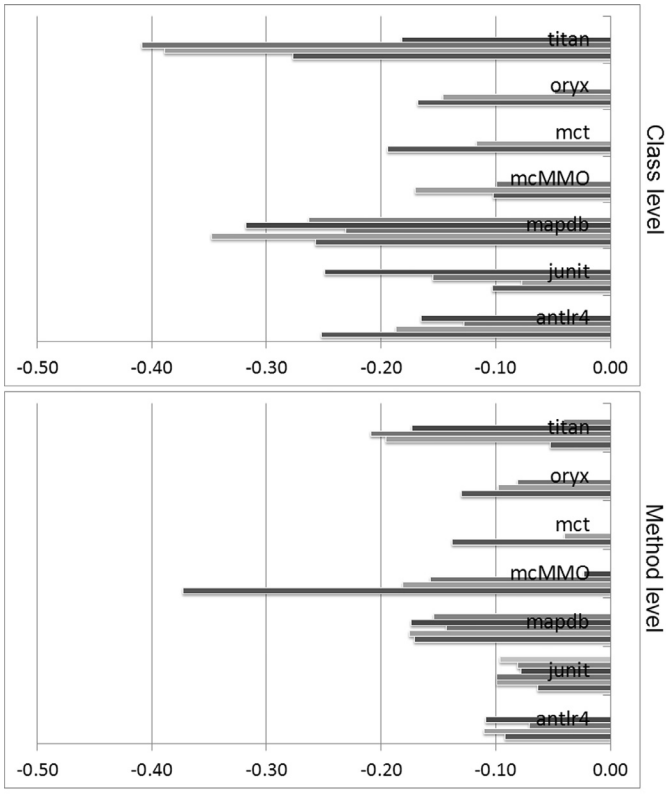

Fig. 7. Correlation of maintainability (RMI) and number of refactorings in classes and methods calculated on the base dataset [13].

Table 6
Spearman's correlation coefficients between RMI and refactoring numbers on the improved dataset (significant values are marked with bold).

System	Class		Method	
	Coeff.	p-value	Coeff.	p-value
antlr4	−0.21163	**.00002**	−0.10281	**.00000**
junit	−0.10665	**.00629**	−0.05947	**.00464**
mapdb	−0.12297	**.01176**	−0.02181	.20615
mcMMO	−0.11661	.27649	−0.06449	.06526
mct	−0.10760	**.00000**	−0.02778	**.00345**
oryx	−0.08828	**.04762**	−0.07381	**.00034**
titan	−0.11507	**.00009**	−0.03125	**.00528**

on these numbers, we can say that *the maintainability of source code entities subjected to refactorings is significantly lower than the maintainability of not refactored entities.*

6.1.2. Comparison with the base dataset results

In a previous study [13] we already presented preliminary results on the connection of code maintainability and refactoring. For that study, we used the original, not validated base dataset containing refactoring instances extracted by RefFinder as is. Since we learned that the precision of RefFinder is quite low, and we created a manually validated, improved subset of the original dataset, it is an interesting question how the new analysis results relate to previous results.

At first, we examined the Spearman's correlation between the RMI and the number of refactorings affecting the source code elements. The previously published results can be seen in Fig. 7 (the various gray bars mark the calculated correlation between the different versions of a system). There is a clear inverse correlation between RMI and the number of refactorings for both classes and methods. That is, the lower the maintainability of a source code element, the more refactorings touch it. We replicated the very same correlation analysis on the manually validated subset of data, and the results are shown in Table 6. Since we have validated data for one version of each subject system (i.e. refactorings are validated between two selected versions), we present only two correlation coefficients for each system, one for the connection between class-level RMI scores and the number of refactorings affecting the classes, and one for the same connection at method level.

We can observe the very same inverse correlations, though the coefficients are slightly lower than those calculated on the base dataset and in case of mcMMO at class level and mapdb at method level, the p-values are well above .05. The p-values are almost the same as the ones presented for the Mann–Whitney U tests (see the detailed discussion and reasoning about them in Section 5.1). Even the tendency that method level coefficients are smaller than class level values is the same. The reason behind the lower coefficients on the improved dataset might be due to the smaller sample sizes. Thus, we can say that the early results seem to remain valid after using the improved dataset. In addition, in Section 5.1 we confirmed that there is a statistically significant difference in the maintainability scores of the refactored and non-refactored source code elements (both at class and method level) with a Mann–Whitney U test. What is more, we also showed that effect size values range from medium to high, and their direction confirms that source code elements to be refactored are likely to have smaller maintainability scores than elements not refactored between the two versions.

6.2. RQ2 – what are the typical values of source code metrics of the refactored and non-refactored elements and how do they change upon refactorings?

6.2.1. Interpretation of the results for classes

The first blue bar in Fig. 3 means that the LOC (Lines Of Code) metric and its changes showed a significant difference between the refactored and non-refactored group of classes for 6 out of the 7 subject systems. If we examine the direction of this difference (i.e. the effect sizes shown in Fig. 4), we can even see that the LOC values of refactored classes are significantly larger in all 6 cases. Thus, developers tend to select and refactor large classes, what is not surprising. LLOC (Logical Lines Of Code) and NOS (Number of Statements) are also size metrics (and strongly correlate with LOC), thus their strong effect is also not surprising.

Another group of metrics showing clear patterns is the coupling metrics: RFC (Response set For a Class), CBO (Coupling Between Object classes), CBOI (Coupling Between Object classes Inverse), NII (Number of Incoming Invocations) and NOI (Number of Outgoing Invocations). The three colored bar for CBOI (i.e. the number of other classes, which directly use a class) means that in 4 out of 7 cases both the metric values and their changes showed a significant difference, while in 1-1 case only the metric values or their changes showed a significant difference between the refactored and non-refactored group of classes. Red cases might suggest that developers consider the given metric to decide which classes to refactor, however, the aim of the refactoring is not (primarily) to improve the given characteristic of the code. Green cases suggest just the opposite, namely that developers do not consider the value of a given metric as a major factor in deciding what classes to refactor, but (intentionally or not) by refactoring a class, they change these properties in a significant way.

The third group of metrics with remarkable patterns are the complexity metrics: WMC (Weighted Methods per Class), NL (Nesting Level) and NLE (Nesting Level Else-If). For all three metrics in 4 out of 7 systems we found significant differences both in the metric values and their changes, while in 2 and 1 cases (for WMC, and NL, NLE, respectively) only the metric values differed between the refactored and non-refactored group of classes. The comment, code clone related and inheritance metrics do not show clear patterns, most of them have only (quite low) green bars.

Looking at the direction of the above discussed differences it is remarkable in Fig. 4 that all the available δ^{Prev} values are positive, which reflects that the appropriate metric values in the refactored group is much likely to be larger than in the non-refactored group. Although most of the values reflect a medium level effect size, such high values like 0.85 (CBOI for mapdb) and 0.84 (RFC for mct) also appear. It suggests that coupling is one of the main factors that developers consider when they select the targets for refactoring (which is in line with other research results [17]). Very similar phenomenon applies to size (i.e. LLOC, LOC, NOS) and complexity (i.e. WMC, NL, NLE) metrics in general.

An interesting observation can be made in connection with the CLOC (Comment Lines Of Code) metric. It measures the amount of comments in a class, and for 5 out of the 7 systems its average value is larger in the classes that are refactored in version x_i (it is even true for the CD – Comment Density metric – for 4 out of 7 cases). Thus, developers refactor classes with more comments, which might seem to be a contradiction at first glance. However, comments are often outdated, misleading or simply comment out unnecessary code, which are all indicators of poor code quality, thus refactoring is justified.

Another interesting question is how refactorings affect the changes of these metric values, do developers intend to control the growth of some metrics with refactorings or not? To analyze this, we observed the values in column δ^{Diff}. One could expect negative values here, which would indicate that the metric value changes are significantly larger in the non-refactored classes (thus eroding much faster due to development [54]). Actually, exactly this pattern can be observed with only a few exceptions (some junit and mapdb complexity and coupling metrics).

The contradictory values in mapdb and the lots of missing values in the mcMMO system (due to not significant test results) are likely to be caused by the very small number of source code elements falling into the refactored group. Thus, the number of metric differences we can use

Table 7
Results of previous studies on class level [10] and method level [13] analysis of the metric value distributions within refactored and non-refactored classes; each cell shows the p-value of the appropriate Mann–Whitney U test.

System name	Class level						Method level			
	CI	WMC	NOI	RFC	LLOC	NOS	CC	LLOC	NOS	NOI
antlr4	**.033**	.428	**.010**	**.031**	**.002**	.122	**.049**	**.000**	**.002**	**.001**
junit	.728	**.042**	.170	N/A	.101	.113	.058	.923	.667	.403
mapdb	**.030**	**.006**	**.005**	**.000**	**.000**	**.000**	**.010**	**.003**	.965	**.002**
mcMMO	**.005**	.608	**.003**	**.013**	.257	.594	.815	.824	.516	.251
mct	.905	.200	N/A	.941	.115	.703	.703	.924	.547	.660
oryx	.667	.575	.381	.533	.743	.159	.654	.555	.306	1.000
titan	**.022**	**.016**	**.000**	**.000**	**.002**	**.042**	.601	**.016**	**.003**	**.000**

is also too small, so hectic or not significant results might occur with higher probability. For the oryx system, many metric values did not change between the two versions, thus we encountered many 0 diff values, which caused the tests to fail in deriving significant results. However, despite the few exceptions, it still quite spectacular that the size, complexity and coupling metrics grow much faster in the non-refactored classes than in the classes subjected to refactoring. This might suggest that developers not just select refactoring targets based on these source code properties, but try to manage and keep these values under control by applying refactorings.

On one hand, we can observe a more or less clear pattern in the presented metrics, namely that the metric values tend to be significantly higher in the classes that are the targets of later code refactorings and the metric values tend to grow much slower (or even decrease) for these classes compared to the non-refactored ones. On the other hand, we did not find metrics that would show the opposite behavior consistently. It means that although there are always some exceptions, there is no such metric that would be consistently larger in the non-refactored classes and/or would grow much slower in non-refactored classes than in the refactored ones, which further strengthens our positive observations.

6.2.2. Interpretation of the results for methods

Looking at Fig. 5 a very similar pattern can be observed for methods than those of presented above for classes, namely that LOC, LLOC and NOS (i.e. size metrics) are at the top of the list. They are followed by complexity metrics (NL, NLE and McCabe). From the coupling group, NOI also plays an important role in selecting refactoring candidates. However, in general, we can say that the results are much weaker for methods; we got significant results for 3 out of the 7 systems for both tests.

Regarding the direction of the differences in the metric values (see Fig. 6), again similar patterns can be observed as for classes, though with smaller effect size values. The reasoning of the contradictory and missing values in case of some systems is the same as for classes, i.e. the small number of refactored elements and/or the small amount of changes in the metric values between the two versions. It looks like examining the effect of code refactorings is much more effective at class level. The granularity of methods might be too fine for such type of evaluation, or we need significantly more refactoring data to be able to apply the tests at method level with stable and meaningful outcome.

6.2.3. Comparison with the base dataset results

We present the summary of the previous study results [10,13] on the analysis of the metric distribution differences using the non-validated base dataset in Table 7. The numbers in the table represent the p-values for the corresponding Mann–Whitney U tests executed on the base dataset (i.e. the non-validated dataset). The results on the manually validated, improved dataset presented earlier in this paper are in line with these preliminary numbers, though they are much more consistent and significant for more subject systems. That is because the preliminary tests were biased by the many false positive refactoring instances that have been removed from the improved dataset. The p-values in column three from the tables presented in Figs. 4 and 6 should be compared to the values in Table 7.

Size, complexity and coupling metrics show the highest differences in their distribution within the refactored and non-refactored groups. However, while previous results displayed 2–4 significant cases out of 7, we had 3–6 significant cases in the new tests on the manually validated, improved dataset, with stronger p-values. For example, in case of mct, we had no significant p-values for the tests run on the base dataset, while on the improved dataset size, complexity and coupling metrics showed a strong connection with refactorings (p-values with zeros in the first 5 decimal places).

Similar can be said for junit and oryx at the class level (p-values with zeros in the first 2–3 decimal places). For antlr4 and titan, there were also lots of significant p-values in the first results, but the results on the improved dataset are at least as strong (some p-values are even a couple of orders of magnitudes lower, like RFC, LLOC or NOS for antlr4, or WMC, NOS for titan at class level). For the mapdb and mcMMO systems we got some weaker values than previously, but that is because the very high number of false positive instances found in these systems, thus only a few refactoring instances remained in the improved dataset (i.e. the small sample size seriously affects the p-values). At method level, the differences are a bit less spectacular, but we could find a connection between method level complexity (McCC – McCabe's cyclomatic complexity) and refactorings in 2 cases that were hidden before.

One more significant difference is that the clone related metrics (CI – Clone Instances at class level and CC – Clone Coverage at method level) are much less significant in the current analysis. Therefore, despite the fact that clone metrics seemed to play an important role in refactorings, it turned out that they were reported only due to the false positive refactoring instances. To summarize, we were able to confirm first results performed on the base dataset, but with an increased confidence and we were able to refuse metrics (i.e. clone metrics) that were reported incorrectly due to false refactoring instances. And what is more, we also showed that these metrics grow much slower (or even decrease) in the refactored source code elements. Furthermore, we found no such metrics that would consistently contradict these results and would show just the opposite behavior.

6.3. Comparison of the results for RQ1 and RQ2

Analyzing RQ1, we concluded that the maintainability – in means of RMI values – of the source code elements subjected to refactorings is significantly lower than the maintainability of the elements not touched by refactorings. Investigating RQ2, we found that several metrics (like size, complexity, and coupling) show significantly higher values in average for those source code elements that are refactored later and they change in greater extent as well. As RMI itself relies on metric values, it is interesting to note that these are exactly the type of metrics it is based on. For calculating RMI, we do not use all the available

Table 8
Correlation coefficients between code lines and number of refactorings affecting source code elements.

Correlation	$Class_0$	$Class_1$	$Method_0$	$Method_1$
Spearman's	0.133	−0.045	0.038	0.029
p-value	.000	.342	.000	.380
Pearson's	0.087	−0.020	0.034	−0.036
p-value	.000	.428	.000	.355

source code metrics extracted by SourceMeter, so this is an interesting observation that most of the metrics affected by code refactorings are the very same that we use to assess the maintainability of the code. This further strengthens that there is a tight connection between source code maintainability and the activity of code refactoring. Code clones are the only exceptions here, which are heavily utilized by the RMI, though turned out to be unrelated to code refactoring in our study. This somewhat counter intuitive result should be addressed by further studies on the subject.

6.3.1. Analysis of the connection between size and number of refactorings

One might argue that the strong connection between the size metrics (and even RMI that depends on the code size as well) and refactorings is due to the simple fact that large code base provides more opportunities for refactoring, thus larger code entities will be refactored more often. To find out whether this is truly the case and to determine the type of connection between size metrics and the number of refactorings affecting a code entity, we performed a correlation analysis.

We took all the classes and methods from the 7 subject systems and run a Pearson and a Spearman's rank correlation analysis between the lines of code (LOC) metrics and the number of refactorings affecting the particular class or method. Given that most of the classes and methods have no refactorings at all, we applied the correlation analysis both on the list of all classes and methods and on the list of classes and methods having at least one refactoring. The results are summarized in Table 8.

Columns $Class_0$ and $Method_0$ show the resulting correlation coefficients and p-values when we included all the classes and methods (even with 0 refactoring counts) and run the correlation analysis on the entire data. Columns $Class_1$ and $Method_1$ display the same results after removing all the classes and methods having 0 refactoring counts. In overall, we can say that there is no strong correlation between lines of code and the number of refactorings. In case of $Class_0$ and $Method_0$ all the coefficients are positive, indicating that the larger the size of a code entity is, the more refactorings affect it, but these coefficients are very small. (The largest one is 0.133 Spearman's ρ coefficient significant at the level of .001.) The values for $Class_1$ and $Method_1$ are not even significant, most probably due to the smaller number of sample sizes.

To summarize, we did not find any strong correlation between code size and number of refactorings. Thus the result showing that refactored classes and methods have larger code sizes in general than the non-refactored classes and methods is not a trivial consequence of the fact that larger code base is more likely to be refactored. The above results also suggest that developers chose refactoring targets based on their combined set of properties (i.e. not just by their sizes).

7. Threats to validity, limitations

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

The key attribute in the datasets is the fully qualified name of the method with parameter descriptions. If a source code element is renamed between two consecutive releases, we do not track it and its metrics, and handle it as a new one in the next release. Following such renamed entities throughout code versions is a really hard task in general, but the number of renaming is relatively small compared to other changes, thus we consider this to be a minor threat.

In addition, there might be arbitrary changes between the two examined releases of the systems, not just refactorings. Therefore, we cannot be sure that changes in a source code element that is affected by a refactoring are only due to the refactoring itself, or other unrelated modifications cause it. The optimal solution would be to find those particular commits that introduce the refactoring, although there is no guarantee that the commit contains only code related to the refactoring itself. Finding those commits would require running RefFinder for each subsequent revision between two releases, which is obviously unfeasible. However, during manual validation we found that most of the refactored source code elements did not contain any additional change, thus the impact of this threat is limited.

Another threat to our results is that we investigated only seven Java systems, which may not represent correctly the general characteristics of all of the software systems considering refactoring activities in practice. Moreover, since manual validation requires huge human effort, the number of refactoring instances in the improved dataset is also limited. Therefore, we plan to continuously extend the number of systems in the improved dataset as well as the number of manually evaluated true positive refactoring instances.

As we employed human evaluation, we cannot be 100% sure that each and every refactoring instance was correctly classified by the authors. However, both evaluators are very experienced researchers and also software developers that mitigates this threat. Moreover, all the authors consulted about refactoring instances that were not straightforward to classify, and resolved these cases by majority voting.

It is hard to ensure the systematic reproducibility of the human evaluations as the classification of refactoring instances is prone to human subjectivity. There is no way we can guarantee that the re-validation of the same instances by someone else would result in the very same classification (true/false instances). To mitigate this, we provide a step-by-step description of our validation process, so that the only part of our study that is not systematically reproducible is the human decision on the refactoring instances. Nonetheless, our purpose was to provide a high-quality, validated golden set of true instances, so that it can be used by other researchers as is, without having to invest the same amount of manual effort we did. We rather encourage the research community to help in validating those instances that have not been classified yet. We believe that the bias caused by the subjectivity of human evaluations based on a line-by-line code review is much less significant than the noise introduced by automatic extraction tools.

By manual validation we can ensure the high precision of the refactoring instances in the improved dataset, however, we cannot guarantee complete recall. It is possible that there are true refactoring instances that RefFinder did not find, thus we also omitted these during the manual validation, what might cause a bias in the evaluation. Nonetheless, the extraction rules of RefFinder are quite conservative, thus the tool is more likely to report false positive instances than to omit true negative ones. Therefore, the effect of missed refactoring instances is low.

Regarding the statistical analysis, the relatively small number of refactoring instances results in unbalanced datasets, which might cause a loss of statistical power. The unbalanced property comes from the fact that there are much less refactored source code elements than unaffected elements and the tests compare the properties of these two sets, the latter one containing a much larger number of samples. However, we chose the Mann–Whitney U method to perform the hypothesis testing, which is not sensitive to population sizes and is able to handle highly unbalanced datasets applying exact distributions of small samples.

8. Conclusions and future work

In this paper we proposed an improved public empirical dataset containing manually validated, fine-grained refactoring data for 7

open-source systems. The dataset can be used for empirical investigations on source code refactoring, like its usage patterns, its effect on source code metrics and maintainability. The published data is a manually validated subset of our previous base dataset containing refactoring instances extracted by the RefFinder tool. With the validation step, we can ensure the high precision of this improved dataset.

With the help of the improved dataset, we examined whether developers tend to refactor source code entities with low maintainability or refactoring activity is not related to the internal quality at all. For this purpose, we analyzed the maintainability values in the improved dataset by running a Mann–Whitney U test on the two groups of entities formed by the fact whether they were affected by any refactorings between two releases or not. The results showed that the overall average maintainability of refactored entities was much lower in the pre-refactoring release than the entities subjected to no refactorings. This strongly suggests that refactoring is indeed used on deteriorated entities in practice no matter if it is a conscious activity of the developers or not.

Moreover, we were interested in how the distribution of typical source code metrics look like in the refactored and non-refactored source code elements. We found that the size, complexity and coupling related metric values were significantly higher in the source code elements being refactored. We could also confirm that developers do not only select their targets for refactoring based on these metrics, but they even try to control and reduce their values, as these metrics grow much slower (or even decrease) in the source code elements touched by refactorings. It would have been interesting to see how these results change by considering individual refactoring types only, but unfortunately, we do not have enough refactoring data in the validated, improved dataset to be able to derive meaningful results at this fine-grained level, so continuous extension of the manually validated instances is one of our major goals.

We also compared the results with our previous findings performed on the base refactoring dataset, where just like in the current study size, complexity and coupling metrics showed the highest differences in their distributions within the refactored and non-refactored groups. However, while previous results displayed 2–4 significant cases out of 7, we had 3–6 significant cases in the new tests on the improved dataset, with much stronger p-values, thus we can be much more confident in the results. We found one major difference as well; the clone related metrics (CI at class level and CC at method level) are much less significant in the current analysis. So despite the fact that clone metrics seemed to play an important role in refactorings, it turned out that they were reported only due to the false positive refactoring instances.

Even though this paper presents a fundamental research, the results can be used as a first step towards understanding refactoring practices more deeply. Having full understanding on the developers' actions we can propose new methods and tools for them that are aligned with their current habits, but help in performing refactoring faster, cheaper, and better. For example, automated refactoring tools would be a great utilization of our results for picking refactoring candidates based on the metric values of the source code elements (considering developer preferences) and suggesting also code changes aligned with developers' taste learned by real-world refactoring samples.

Currently, the improved refactoring dataset contains only the fraction of the data from the base dataset (i.e. around 6%). This is due to the large amount of human effort required for validation. We plan to continuously increase the size of the improved dataset, but a community supported common effort would be very welcome.

Acknowledgment

This research was supported by the EU-funded Hungarian national grant GINOP-2.3.2-15-2016-00037 titled "Internet of Living Things" and the UNKP-17-4 New National Excellence Program of the Ministry of Human Capacities, Hungary.

References

[1] M. Fowler, Refactoring: Improving the Design of Existing Code, Addison-Wesley, 1999.
[2] E. van Emden, L. Moonen, Java quality assurance by detecting code smells, Proceedings of the 9th Working Conference on Reverse Engineering, (2002), pp. 97–106.
[3] F.A. Fontana, S. Spinelli, Impact of refactoring on quality code evaluation, Proceedings of the 4th Workshop on Refactoring Tools, WRT '11, ACM, New York, NY, USA, 2011, pp. 37–40.
[4] F. Khomh, M. Di Penta, Y.-G. Gueheneuc, An exploratory study of the impact of code smells on software change-proneness, Proceedings of the 16th Working Conference on Reverse Engineering, IEEE, 2009, pp. 75–84.
[5] M. Mantyla, J. Vanhanen, C. Lassenius, A taxonomy and an initial empirical study of bad smells in code, Proceedings of the 2003 International Conference on Software Maintenance, ICSM 2003. IEEE, 2003, pp. 381–384.
[6] R. Peters, A. Zaidman, Evaluating the lifespan of code smells using software repository mining, Proceedings of the 16th European Conference on Software Maintenance and Reengineering (CSMR), (2012), pp. 411–416.
[7] A.F. Yamashita, L. Moonen, Do developers care about code smells? An exploratory survey, Proceedings of the 2013 Working Conference on Reverse Engineering, WCRE, 13 (2013), pp. 242–251.
[8] R. Arcoverde, A. Garcia, E. Figueiredo, Understanding the longevity of code smells: preliminary results of an explanatory survey, Proceedings of the 4th Workshop on Refactoring Tools, WRT '11, ACM, New York, NY, USA, 2011, pp. 33–36.
[9] G. Bavota, A. De Lucia, M. Di Penta, R. Oliveto, F. Palomba, An experimental investigation on the innate relationship between quality and refactoring, J. Syst. Softw. 107 (2015) 1–14.
[10] I. Kádár, P. Hegedűs, R. Ferenc, T. Gyimóthy, A code refactoring dataset and its assessment regarding software maintainability, 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), 1 (2016), pp. 599–603.
[11] M. Kim, M. Gee, A. Loh, N. Rachatasumrit, Ref-Finder: a refactoring reconstruction tool based on logic query templates, Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'10), (2010), pp. 371–372.
[12] K. Prete, N. Rachatasumrit, N. Sudan, K. Miryung, Template-based reconstruction of complex refactorings, Proceedings of the 2010 IEEE International Conference on Software Maintenance (ICSM), (2010), pp. 1–10.
[13] I. Kádár, P. Hegedűs, R. Ferenc, T. Gyimóthy, Assessment of the Code Refactoring Dataset Regarding the Maintainability of Methods, Springer International Publishing, pp. 610–624.
[14] I. Kádár, P. Hegedűs, R. Ferenc, T. Gyimóthy, A manually validated code refactoring dataset and its assessment regarding software maintainability, Proceedings of the 12th International Conference on Predictive Models and Data Analytics in Software Engineering, PROMISE 2016, ACM, New York, NY, USA, 2016, pp. 10:1–10:4.
[15] T. Bakota, P. Hegedűs, P. Körtvélyesi, R. Ferenc, T. Gyimóthy, A probabilistic software quality model, Proceedings of the 27th IEEE International Conference on Software Maintenance (ICSM), (2011), pp. 243–252.
[16] T.J. McCabe, A complexity measure, IEEE Trans. Softw. Eng. vol. 2, (1976) 308–320.
[17] A. Murgia, R. Tonelli, M. Marchesi, G. Concas, S. Counsell, J. McFall, S. Swift, Refactoring and its relationship with Fan-in and Fan-out: an empirical study, Proceedings of the 16th European Conference on Software Maintenance and Reengineering (CSMR), (2012), pp. 63–72.
[18] Y. Kataoka, T. Imai, H. Andou, T. Fukaya, A quantitative evaluation of maintainability enhancement by refactoring, Proceedings of the International Conference on Software Maintenance, (2002), pp. 576–585.
[19] Y. Kosker, B. Turhan, A. Bener, An expert system for determining candidate software classes for refactoring, Expert Syst. Appl. 36 (6) (2009) 10000–10003.
[20] D. Silva, N. Tsantalis, M.T. Valente, Why we refactor? Confessions of GitHub contributors, CoRR (2016), abs/1607.02459.
[21] J. Ratzinger, T. Sigmund, H.C. Gall, On the relation of refactorings and software defect prediction, Proceedings of the 2008 International Working Conference on Mining Software Repositories, MSR '08, ACM, New York, NY, USA, 2008, pp. 35–38.
[22] E. Murphy-Hill, C. Parnin, A.P. Black, How we refactor, and how we know it, IEEE Trans. Softw. Eng. 38 (1) (2012) 5–18.
[23] S. Negara, N. Chen, M. Vakilian, R.E. Johnson, D. Dig, A comparative study of manual and automated refactorings, Proceedings of the 27th European Conference on Object-Oriented Programming, ECOOP'13, Springer-Verlag, Berlin, Heidelberg, 2013, pp. 552–576.
[24] M.I. Hoque, V.N. Ranga, A.R. Pedditi, R. Srinath, M.A.A. Rana, M.E. Islam, A. Somani, An empirical study on refactoring activity, ACM Comput. Res. Repos. (2014), abs/1412.6359.
[25] N. Tsantalis, V. Guana, E. Stroulia, A. Hindle, A multidimensional empirical study on refactoring activity, Proceedings of the 2013 Conference of the Center for Advanced Studies on Collaborative Research, CASCON '13, IBM Corporation, Riverton, NJ, USA, 2013, pp. 132–146.
[26] E. Choi, N. Yoshida, K. Inoue, An investigation into the characteristics of merged code clones during software evolution, IEICE Trans. Inf. Syst. 97 (5) (2014) 1244–1253.
[27] E. Choi, N. Yoshida, K. Inoue, What kind of and how clones are refactored?: A case study of three OSS projects, Proceedings of the 5th Workshop on Refactoring Tools, WRT '12, ACM, New York, NY, USA, 2012, pp. 1–7.

[28] W. Wang, M.W. Godfrey, Recommending clones for refactoring using design, context, and history, Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE, 2014, pp. 331–340.
[29] T. Tourwe, T. Mens, Identifying refactoring opportunities using logic meta programming, Proceedings of the 7th European Conference on Software Maintenance and Reengineering, (2003), pp. 91–100.
[30] T. Dudziak, J. Wloka, Tool-supported Discovery and Refactoring of Structural Weaknesses in Code, Technical University of Berlin, Germany, 2002 Unpublished doctoral dissertation.
[31] L. Tahvildari, K. Kontogiannis, A metric-based approach to enhance design quality through meta-pattern transformations, Proceedings of the 7th European Conference on Software Maintenance and Reengineering, IEEE, 2003, pp. 183–192.
[32] G. Szőke, C. Nagy, L.J. Fülöp, R. Ferenc, T. Gyimóthy, FaultBuster: an automatic code smell refactoring toolset, in: M.W. Godfrey, D. Lo, F. Khomh (Eds.), SCAM, IEEE Computer Society, 2015, pp. 253–258.
[33] S. Counsell, X. Liu, S. Swift, J. Buckley, M. English, S. Herold, S. Eldh, A. Ermedahl, An exploration of the 'Introduce Explaining Variable' refactoring, Proceedings of the XP2015 Scientific Workshop, XP '15 workshops, ACM, New York, NY, USA, 2015, pp. 9:1–9:5.
[34] H. Liu, Q. Liu, Y. Liu, Z. Wang, Identifying renaming opportunities by expanding conducted rename refactorings, IEEE Trans. Softw. Eng. vol. 41, (9) (2015) 887–900.
[35] A. Ghannem, G. El Boussaidi, M. Kessentini, Model refactoring using examples: a search-based approach, J. Softw. 26 (7) (2014) 692–713.
[36] M.W. Godfrey, L. Zou, Using origin analysis to detect merging and splitting of source code entities, IEEE Trans. Softw. Eng. 31 (2) (2005) 166–181.
[37] S. Demeyer, S. Ducasse, O. Nierstrasz, Finding refactorings via change metrics, SIGPLAN Not. 35 (10) (2000) 166–177.
[38] F. Van Rysselberghe, S. Demeyer, Reconstruction of successful software evolution using clone detection, in: T. Mikkonen, M.W. Godfrey, M. Saeki (Eds.), Proceedings of the International Workshop on Principles of Software Evolution, IEEE Computer Society Press, 2003, pp. 126–130.
[39] Z. Xing, E. Stroulia, Refactoring detection based on UMLDiff change-facts queries, Proceedings of the Working Conference on Reverse Engineering, 6 Citeseer, 2006, pp. 263–274.
[40] G. Soares, R. Gheyi, E. Murphy-Hill, B. Johnson, Comparing approaches to analyze refactoring activity on software repositories, J. Syst. Softw. 86 (4) (2013) 1006–1022.
[41] G. Soares, B. Catao, C. Varjao, S. Aguiar, R. Gheyi, T. Massoni, Analyzing refactorings on software repositories, Proceedings of the 25th Brazilian Symposium on Software Engineering, SBES, Sao Paulo, Brazil, September 28–30, (2011), pp. 164–173.
[42] R. Stevens, C.D. Roover, C. Noguera, V. Jonckers, A history querying tool and its application to detect multi-version refactorings, in: A. Cleve, F. Ricca, M. Cerioli (Eds.), Proceedings of the 17th European Conference on Software Maintenance and Reengineering, IEEE Computer Society, 2013, pp. 335–338.
[43] Q.D. Soetens, J. Pérez, S. Demeyer, An initial investigation into change-based reconstruction of floss-refactorings, Proceedings of the International Conference on Software Maintenance, IEEE Computer Society, 2013, pp. 384–387.
[44] X. Ge, Q.L. DuBose, E.R. Murphy-Hill, Reconciling manual and automatic refactoring, in: M. Glinz, G.C. Murphy, M. Pezzè (Eds.), Proceedings of the International Conference on Software Engineering, IEEE Computer Society, 2012, pp. 211–221.
[45] K. Taneja, D. Dig, T. Xie, Automated detection of Api refactorings in libraries, Proceedings of the 22nd IEEEACM International Conference on Automated Software Engineering, ASE '07, ACM, New York, NY, USA, 2007, pp. 377–380.
[46] R. Mahouachi, M. Kessentini, M.Ó. Cinnéide, Search-based refactoring detection using software metrics variation, International Symposium on Search Based Software Engineering, Springer, 2013, pp. 126–140.
[47] G. Antoniol, M. Di Penta, E. Merlo, An automatic approach to identify class evolution discontinuities, Proceedings of the 7th International Workshop on Principles of Software Evolution (IWPSE 2004), 6–7 September 2004, Kyoto, Japan, IEEE Computer Society, 2004, pp. 31–40.
[48] T. Bakota, P. Hegedüs, I. Siket, G. Ladányi, R. Ferenc, QualityGate SourceAudit: a tool for assessing the technical quality of software, 2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering, CSMR-WCRE 2014, Antwerp, Belgium, February 3–6, 2014, (2014), pp. 440–445.
[49] P. Oman, J. Hagemeister, Metrics for assessing a software system's maintainability, Proceedings of the International Conference on Software Maintenance, IEEE CS Press, 1992, pp. 337–344.
[50] P. Hegedűs, T. Bakota, G. Ladányi, C. Faragó, R. Ferenc, A drill-down approach for measuring maintainability at source code element level, Electron. Commun. EASST 60 (2013).
[51] T. Menzies, R. Krishna, D. Pryor, The promise repository of empirical software engineering data, (2015). URL http://openscience.us/repo
[52] H.B. Mann, D.R. Whitney, On a test of whether one of two random variables is stochastically larger than the other, Ann. Math. Statist. 18 (1) (1947) 50–60.
[53] M.R. Hess, J.D. Kromrey, Robust confidence intervals for effect sizes: a comparative study of Cohen's d and Cliff's delta under non-normality and heterogeneous variances, Annual Meeting of the American Educational Research Association, (2004), pp. 1–30.
[54] T. Bakota, P. Hegedűs, G. Ladányi, P. Körtvélyesi, R. Ferenc, T. Gyimóthy, A cost model based on software maintainability, Proceedings of the 28th IEEE International Conference on Software Maintenance (ICSM 2012), IEEE Computer Society, Riva del Garda, Italy, 2012, pp. 316–325.