An Empirical Study on Maintainable Method Size in Java

Shaiful Alam Chowdhury
University of British Columbia
Vancouver, BC, Canada
shaifulc@cs.ubc.ca

Gias Uddin
University of Calgary
Calgary, AB, Canada
gias.uddin@ucalgary.ca

Reid Holmes
University of British Columbia
Vancouver, BC, Canada
rtholmes@cs.ubc.ca

Abstract

Code metrics have been widely used to estimate software maintenance effort. Metrics have generally been used to guide developer effort to reduce or avoid future maintenance burdens. Size is the simplest and most widely deployed metric. The size metric is pervasive because size correlates with many other common metrics (e.g., McCabe complexity, readability, etc.). Given the ease of computing a method's size, and the ubiquity of these metrics in industrial settings, it is surprising that no systematic study has been performed to provide developers with meaningful method size guidelines with respect to future maintenance effort. In this paper we examine the evolution of ~785K Java methods and show that developers should strive to keep their Java methods under 24 lines in length. Additionally, we show that decomposing larger methods to smaller methods also decreases overall maintenance efforts. Taken together, these findings provide empirical guidelines to help developers design their systems in a way that can reduce future maintenance.

CCS Concepts

• **Software engineering → Code metrics**.

Keywords

SLOC, code metrics, maintenance, McCabe, Readability

1 Introduction

Software maintenance has long been identified as challenge for software engineers [43] and maintenance costs often exceed initial development costs [13]. Consequently, both researchers and practitioners are keen to model future maintenance effort from the current state of a software project—to facilitate risk planning and to assist project optimization [20, 45, 53, 76, 79, 85, 93]. These maintenance models rely on various metrics to provide the data necessary to predict future maintenance challenges.

Two of the most widely adopted maintenance indicators are change-proneness [26, 69] and bug-proneness [38, 54, 62, 85, 94]. These metrics suggest that code components (e.g., methods, classes, or modules) that are bug- or change-prone are generally expensive to maintain. Early identification and optimization of such components could thus help reduce future maintenance effort. Unfortunately, change- and bug-proneness of code components are not generally available until after one or more system releases [78].

To address this shortcoming, researchers have investigated static code metrics to learn how these metrics alias with change- and bug-proneness (e.g., [26, 43, 62, 93]). These metrics include McCabe [52], McClure [53], nested block depth [42], C&K [18], and size, just to name a few. The utility of these metrics, however, has long been debated (e.g., [25, 26, 74]). While some research has shown that these static metrics correlate with bug- and change-proneness (e.g.,[6, 10, 42, 47, 78]), other research has criticized these findings (e.g., [26, 71, 74]). One underlying concern with these metrics is how they alias with the size of the component being measured. It has been shown that most code metrics are highly influenced by their correlation with size [23, 26, 77]. This suggests that component size, which is easy for software engineers to both measure and reason about, could be the primary maintenance indicator [26].

Ultimately, developers want to be able to design their systems in a way that reduces future maintenance effort. Unfortunately, there is no evidence-based guidelines that developers can follow to optimize the size of their components. In this paper we focus on method level granularity to measure size (e.g., in contrast to class-, or module-level granularities), because method level granularity is desired by both developers and researchers [29, 62]. There is not currently broad consensus for how long a method should be; for example, consider *'What is the ideal length of a method for you?'* [80]. The suggestions vary greatly: between 5-15 lines, 50 lines, and 100-200 lines. We propose that having evidence-based guidance to help developers understand the relationship between method size and software maintenance would help them better structure their systems to reduce future maintenance effort. Therefore, this paper describes an empirical study to investigate optimal method sizes with respect to software maintenance.

To do this, we examined the evolution of ~785K Java methods from 49 open-source software projects to investigate the relationship between method length and change- and bug-proneness. Based on our study, we make the following four primary contributions to the field of software maintenance:

Maintenance effort correlates with method length (RQ1). Previous work has shown that both file-length [26] and module-length [27] negatively impact maintenance; we extend these findings to method-length. We also investigate the best method length measure to use for maintenance prediction.

Methods should be 24 SLOC or fewer (RQ2). Suggesting shorter methods are better is insufficient guidance for practical application as there are challenges to having arbitrarily small methods. We show that methods that are 24 or fewer SLOC are less maintenance-prone than longer methods. We also show that in practice this threshold is widely achievable.

Decomposed large methods undergo less maintenance (RQ3). Complex problems often require complex solutions, which may exceed the 24 SLOC threshold. A natural question is whether these

complex solutions are inherently maintenance-prone. We show that in most cases, larger methods are more maintenance-prone than methods that have been decomposed into shorter helper methods. **Controlling for size also controls other code quality metrics (RQ4).** We show that keeping method size within 24 SLOC also improves the testability, readability, and maintainability of source methods.

Taken together, this paper provides an evidence-based analysis of the impact on method length on software maintenance that developers can use to reduce future maintenance effort as they write and evolve their systems. We further show that these length guidelines are both achievable in practice and suggest that refactoring complex methods into smaller sub-methods further improves system maintainability. To reproduce our results, we publicly share the dataset of ∼785K methods with their change and bug information.[1]

2 Related Work

Maintenance effort can be estimated through external factors such as correctness and performance of the underlying software system. Unfortunately, these factors are hard to collect and are often unavailable from the beginning of software development life cycle (SDLC) [26]. Internal metrics, also known as code metrics, are easy to collect and are available throughout the SDLC. Therefore, it has been a holy grail to the research community to understand software maintenance using different code metrics [18, 46, 52, 53, 62, 86]. Consequently, a significant numbers of code metrics have been proposed and studied over the last forty years [48].

McCabe [52], or cyclomatic complexity, is a measure of the number of independent paths in a source code component. The assumption is that a source code component with high McCabe score would be hard to maintain; it would be more change- and bug-prone [5, 6, 21, 47, 61, 62, 68, 84, 93]. Another popular code metric is the Halstead metric, which is mainly based on the number of operators and operands in a code component [7, 21, 35, 43, 63]. Abreu *et al.* [2] proposed the coupling factor metric which is based on the client-supplier relationships among different classes.

Chidamber and Kemerer have proposed the popular suite of six OO metrics, known as the C&K metric [18]. This suite includes *Depth of Inheritance Tree*, *Number of Children*, and *Coupling between Object Classes*. Readability as a code metric was developed by Buse *et al.* [15], who modelled different code features, such as number of comments, and number of blank lines, to a readability score. This model was later improved by Posnett *et al.* [63] and Scalabrino *et al.* [72]. Mo *et al.* [56] developed the decoupling level metric to measure how hard it is to modify a class without affecting other classes in a software project. Code metrics are often used to define antipatterns (code smells)—i.e., poor design choices that make a software less maintainable. Studies have found that classes (and modules) with antipatterns are generally more change- and bug-prone than classes with no, or few, antipatterns [30, 44, 70].

Unfortunately, the effectiveness of code metrics for understanding future maintenance has been debated extensively. Although many studies have been positive about the utility of reasoning about maintenance with code metrics [6, 10, 42, 47, 78], some studies found them ineffective [26, 71, 74]. According to these studies,

size is the only code metric that can estimate maintenance to some extent [23, 26, 32, 77], and all the other code metrics are only as effective as their correlation strength with size [26]. A code metric, such as McCabe, Halstead, or even the C&K, cannot offer any new information about maintenance if their correlation with size is neutralized. Size has been acknowledged as the predominant code metric even by the studies that support the usefulness of other code metrics [42, 78]. Even many of the code smells, or antipatterns, were found to be directly associated with size [44]. It is no exaggeration to state that size is *"The One Metric to rule them all"*.

One of the primary goals of code metric studies is to provide guidelines about metrics thresholds for maintainability [3, 4, 8]. And yet, despite being the most important metric, there is no systematic study that provides guidelines about the relationship between size and future maintainability. In this paper we focus on size thresholds at the method-level granularity. Method level is the most desirable to the developers, because class/file, or module-level granularities are often too coarse to be practically useful [24, 29, 62, 75].

3 Methodology

This section describes the methodology of this paper.

3.1 Project Selection

Code metrics research relies on aggregated [26, 62, 78] or individual project analysis [43, 69, 76, 93]. In aggregated approaches, measurements from a set of selected projects are merged together to produce an observation (e.g., a correlation coefficient between McCabe and change proneness). We argue that this approach is inaccurate. For example, the change-proneness of a method could be influenced by factors that are project dependent, such as the number of active contributors, their commit patterns, code review policy, and the number of releases influences how a software evolves [33, 51, 90]. Additionally, aggregated analysis results can be heavily influenced by a few large projects, making observations from smaller projects unnoticeable. A more accurate approach is to analyze each project separately to see if an observation is common across projects. This approach, however, can be criticized for selection bias [26, 64].

We addressed the selection bias threat by selecting all the GitHub projects used by five different previous studies (specifically from [26, 29, 60, 67, 78]), resulting in 49 open-source Java projects. Table 1 summarizes the dataset. The dataset contains projects from different domains with varied number of methods and varied average number of revisions per method. This broad selection of a large number of projects increases the generalizability of the observations about code metrics and maintenance we can make. One thing to note is the low average number of method revisions in Table 1. This is because only a small subset of methods undergo many revisions, which increases the utility of approaches that can detect these methods: if we can detect and improve those few high-churn methods early, we can meaningfully reduce future maintenance effort.

3.2 Data Collection

To understand the relationship between method size and change-/bug-proneness, we need method-level evolution history. We used the state-of-the-art history tracing tool CodeShovel [28, 29], which returns the complete change history of a given method. CodeShovel also collects information about when a method was modified, who

---

[1]https://github.com/shaifulcse/MaintainableSLOC-data

Table 1: Description of the top and the bottom five projects (ranked by # methods) and their method-level statistics. To save space, the full table is shared with our dataset (`stats/table-1.pdf`). In total, 785,606 Java methods were extracted. Getters and setters and methods that are younger than 2 years are excluded. Snapshot SHAs are included to aid reproducibility.

Repository	# Methods	# Revisions (avg)	# Getters-Setters	Age $\geq$ 2 Years	# Used Methods	Snapshot
hadoop	70,524	1.8	11,219	63,378	53,729	4c5cd7
elasticsearch	63,253	3.7	8,875	39,085	34,641	92be38
flink	38,482	1.9	6,790	19,967	15,792	261e72
presto	37,639	1.9	8,520	26,785	20,766	bb20eb
lucene-solr	37,600	1.6	5,625	29,287	24,641	b457c2
...	...	...	...	...	...	. . .
mockito	1,526	4.2	144	1,221	1,113	077562
cucumber-jvm	1,222	2.7	170	435	374	b57b92
commons-io	1,149	3.1	85	952	884	11f0ab
vraptor4	934	1.7	132	933	801	593ce9
junit4	879	3.1	56	856	801	50a285
Total	**785,606**				**520,874**	

modified it, what was modified, and why the modification took place (i.e., commit message). CodeShovel is robust to method or file rename operations while uncovering a method's history. It produces accurate change histories for 90% methods (with 97% accuracy for method changes).

3.3 Maintenance Effort Indicators

User studies are a widely adopted approach [1, 7, 11, 15, 21, 22, 36, 43, 71, 72] for understanding maintenance effort. The maintenance effort of a code fragment can be understood by the cognitive effort and time required to understand the code by a selected group of users. The outcome of user studies, however, are often not generalizable [14], and contradictory conclusions can be made by comparing two independent studies [42, 71]. We have mitigated this threat by using change- and bug-proneness as the maintenance indicators as have been widely used in previous maintenance studies (e.g. [16, 24, 26, 65, 69, 75]).

3.3.1 Change-proneness A maintainable code fragment (a Java method in our case) should be less change-prone: it should not require too many revisions, and none of its changes should be too big. Supporting this hypothesis, we opted for the following four sub-indicators to understand how change-prone a Java method is.

#Revisions: The number of revisions, a common maintenance indicator [5, 6, 58, 76], is the total number of times a method has been modified, regardless of the type of modification—corrective, adaptive, preventive, or perfective [22].

ChangeSize: The number of revisions can be misleading, because the effort required to change two Java methods with the same number of revisions might not be the same. Also, number of revisions can be impacted by developers' commit habit [90]. One alternative is to use the sum of the change sizes (i.e., `git diff`) [73, 76] considering all the revisions the method underwent.

AdditionOnly: Does adding new lines require the same effort to delete existing lines? Probably not. Therefore, following prior work [76], we also use the sum of the number of *added lines only* as an indicator of change-proneness.

EditDistance: The number of added, and/or deleted lines (as considered in `git diff`) can be impacted by an individual's coding

style. It also does not differentiate between modifying a large line and a small line. We therefore used Levenshtein edit distance [49] as our final change-proneness indicator, as recommended by some prior studies [71, 73, 79]. Levenshtein edit distance counts the number of characters that are added, deleted, or modified, to convert one source code version into another.

3.3.2 Bug-proneness We consider a method as *bug-prone*, if it is associated with a bug-fixing commit. In accordance with earlier stdies [57, 67], a commit is considered as bug-fix if the its commit message contains at least one bug related keyword: *error*, *bug*, *fixes*, *fixing*, *fix*, *fixed*, *mistake*, *incorrect*, *fault*, *defect*, *flaw*. The level of bug-proneness of a method is the number of bug-fix commits associated with the method.

3.4 Size Calculation

Size is commonly referred to as the source lines of code (SLOC). Some previous studies were explicit about how they calculated SLOC (e.g., [6, 47, 62]), but some studies were not (e.g., [26, 76]). We calculated SLOC in three different ways, to evaluate if how size is calculated has any impact on its relationship with the maintenance indicators, such as change- and bug-proneness.

SLOCStandard source lines of code without comments and blank lines. This form of SLOC is widely used (e.g., [47, 66]).

SLOCAsItIs source lines of code with comments and blank lines.

SLOCPretty The same code snippet, when written by different developers, may produce different SLOC values. By using the PrettyPrinter function of the javaparser [39], we convert every source code to a common format producing the same SLOC value for the variants of a code snippet.

3.5 Preprocessing

Getters/setters: Getters and setters, also known as accessor methods, are often generated automatically and create noise in code metric studies [8, 31, 50]. Similar to earlier studies [1, 8], we filtered them out from our dataset, before calculating any code metrics. If a method name starts with `get`, has non-void return type, and does not have any parameter, then we identify this method as a *getter*. Similarly, if a method name starts with `set`, has one parameter and

a void return type, then we label this method as a *setter*. Table 1 shows the number of getters/setters for each project.

Controlling for Age: The Kendall's τ correlation coefficient between ages of methods and their number of revisions is 0.2 in our aggregated dataset. When considered individually, some projects show very high correlation (e.g., 0.51 for the checkstyle project). Therefore, when analyzing the relationship between SLOC and maintenance indicators, we should not include methods with different ages. Generally speaking, a ten year old method will have more chances to change than a newly introduced method, regardless of their code complexity. To perform age normalization, we followed a two step procedure. In step 1, we removed methods that are less than two years old. This does not, however, solve the problem completely, because we may still compare a 2 years old with a $2+x$ years old method. Therefore, in step 2, we excluded all revisions and bugs that happened after two years of a method's life. If we set the threshold more than two years, we lose many methods from our dataset. As presented in Table 1, with two year threshold, we still have 520,874 methods for our final analyses. If we set the threshold less than two years, we lose many change and bug information, because the window to observe changes becomes narrower.

Multiple Versions: A method can have multiple SLOCs while it evolves. Consider a method associated with 5 SLOCs in its lifetime: 20, 50, 50, 50, 22. The method was revised 4 times after its introduction, so the value of #revisions is 4. To calculate the correlation between SLOC and #revisions, what SLOC value should represent this method? If we take the introduction SLOC, we will map SLOC 20 to 4 revisions, which is inaccurate because the method was revised only once for SLOC 20. We solve this problem by applying a versioning technique, where the above method has three different versions. That way version 1 has SLOC 20 with 1 revision, version 2 has SLOC 50 with 3 revisions, and version 3 has SLOC 22 with 0 revision, because the method did not change afterwards. We apply this technique for other maintenance indicators as well.

3.6 Statistical Tests

Throughout this paper, we compare if different distributions are statistically different. We applied the Anderson-Darling normality test [83] to some of the randomly selected distributions and found that many of the distributions are not normally distributed. Therefore, non-parametric tests are selected for our statistical testing. For evaluating if two distributions are different, we use the Wilcoxon rank sum test (and Wilcoxon signed rank test for pairwise comparison). For calculating the correlation coefficient between two variables, we use the non-parametric Kendall's τ, instead of the Pearson's correlation. For calculating effect sizes, we apply Cliff's Delta instead of Cohen's d. In accordance with [34], we classify an effect size either as Negligible ($0 \leq |\delta| < 0.147$), Small ($0.147 \leq |\delta| < 0.330$), Medium ($0.330 \leq |\delta| < 0.474$), or Large ($0.474 \leq |\delta| \leq 1$). These non-parametric tests are widely used in software engineering research [19, 26, 37].

4 Results

In this section, we present our findings to our four research questions. We investigate whether method size is an important factor to understand change- and bug-proneness, what the upper-limit of method's size should be, whether developers should decompose

large methods into smaller ones, and if controlling for size controls other code quality factors.

### 4.1 RQ1: Is SLOC correlated with change- and bug-proneness at method-level?
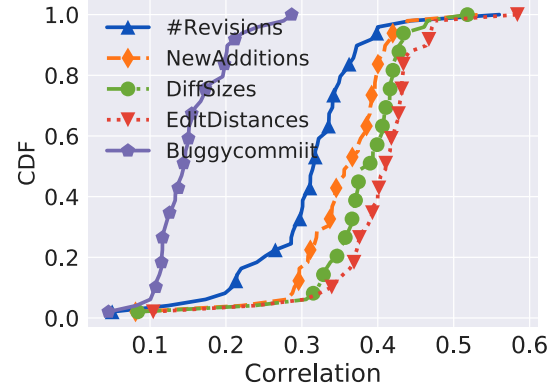
Figure 1: Cumulative distribution functions of the correlation between SLOC and the five maintenance indicators. Each line represents 49 points for our 49 different projects. For graph readability, number of markers has been reduced.

Figure 1 shows the distribution of Kendall's τ correlation coefficients between SLOC and the five maintenance indicators. Correlation coefficient for each project is shared with our pubic dataset as a table as well (`stats/table-2.pdf`). For all projects, the correlation coefficients are positive, indicating method size is an indicator of future maintenance. When compared among the four change-proneness indicators, SLOC performance is generally the lowest for #revisions, and highest for EditDistance. Correlations between SLOC and bug-proneness (i.e., #BuggyCommits) are significantly lower than the four change-proneness indicators. This implies that, at the method-level, code metrics are comparatively less helpful for bug prediction than for change prediction. This complements the recent findings of Pascarella *et al.* [62], who consider method-level bug prediction as an open research problem. We argue that it is unrealistic to expect very high correlation between a code metric and maintenance effort. A very high correlation would mean that maintenance effort can be estimated just by using one code metric. This is unrealistic because there are many factors that influence code maintenance, such as developer habits [82], application domain and platforms [88, 92], code clones [58], software architecture [4], test code quality [9, 78], and changes in requirements.

We also observe that SLOC performance is not similar across projects. For the docx4j project, SLOC correlates to change-proneness very weakly (e.g., correlation is only 0.05 with #revisions) while for the voldemort project the correlation is 0.56. This observation is true for bug-proneness as well. For the cucumber-jvm project, the correlation is only 0.05, but for the xerces2-j, the correlation is 0.29. These observations explain why cross project maintenance prediction is still a challenging problem [81, 92].

Table 2: P-values from Wilcoxon signed rank test (i.e., pairwise test). Most of the time SLOCPretty is significantly different than the other two ($p \leq 0.05$).

Type1	Type2	# Revisions	# Additions	DiffSize	EditDistance	Bugs
SLOCStandard	SLOCAsItIs	0.45	0.78	0.06	0.00	0.02
SLOCStandard	SLOCPretty	0.00	0.00	0.00	0.00	0.07
SLOCAsItIs	SLOCPretty	0.00	0.00	0.00	0.00	0.01
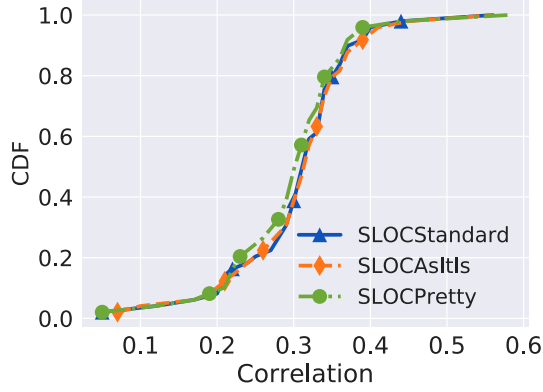
Figure 2: Cumulative distribution functions of the correlation between different SLOC types and number of revisions.

Unless otherwise mentioned, we have considered SLOC as the source lines of code without comments and blank lines (SLOCStandard), which we found to be the most common in the literature (e.g., [47, 66]). No previous study, however, has investigated if this is the best size measure to understand maintenance. With #revisions as the maintenance indicator, Figure 2 shows the performance comparison among the three different SLOC calculations (described in Section 3.4). Both SLOCStandard and SLOCAsItIs slightly outperform SLOCPretty. According to the Wilcoxon signed rank test (for pairwise testing), the performance between SLOCStandard and SLO-CAsItIs are not statistically different ($p > 0.05$). Their performance, however, are statistically different with SLOCPretty ($p \leq 0.05$). Table 2 shows the results for all the maintenance indicators.

We also calculated the Cliff's Delta effect sizes among the performance distributions. The differences are always negligible, except for two cases. These two cases are between SLOCPretty and SLO-CAsItIs: one for the DiffSize and the other for the EditDistance indicator. The signs of the Cliff's Delta values (+/-), however, suggest that SLOCPretty performs the worst whereas SLOCStandard performs the best, although with negligible effect sizes.

Summary: SLOC correlates with change- and bug-proneness at the method-level granularity. SLOC's correlation with bug-proneness is generally lower than change-proneness. The best way to calculate SLOC is to calculate source lines of code without comments and blank lines.

4.2 RQ2: What is the maintainable method size?

Our previous analysis shows that method SLOC is positively correlated with maintenance effort. This means developers should try to keep their method size as small as possible. *But how small should methods be?* Suggesting a very low upper-bound, such as *'keep your method size within 5 SLOC'*, can be impractical. In order to find a method size limit that is realistic and maintainable at the same time, we follow two steps. In step 1, we find a realistic method size limit by studying all the methods from the 49 Java projects in our dataset. In step 2, we empirically evaluate if this limit can indeed reduce maintenance effort by reducing change- and bug-proneness.

4.2.1 A realistic upper bound of method size Alves *et al.* [3] proposed a 6-steps systematic approach to find thresholds for any given code metric, such as McCabe. The methodology does not depend on intuition or expert opinion and is resilient to outlier projects. From the source code of a set of software projects, the methodology can automatically derive thresholds of a code metric in four sizes: small, medium, large, and very large. We apply this methodology to find these four thresholds for SLOC at the method-level granularity for our dataset of 49 projects. Although we refer to [3, 8, 91] for more details, for reproducibility, we here describe the exact process followed. To have better understanding, let us consider two projects each containing three methods. For the first project (P^1), SLOCs of the methods are 10, 20, and 10, whereas for the second project (P^2), the SLOCs are 20, 10, and 20.

Step 1 For each method M^m in project P^1, we calculate its SLOC: $P^1_{M^m_{sloc}}$. So for the first project, we have:

$$P^1_{M^1_{sloc}} = 10, P^1_{M^2_{sloc}} = 20, P^1_{M^3_{sloc}} = 10$$

Step 2 For each M^m in P^1, we calculate its weight ratio, $P^1_{M^m_{weight}} = P^1_{M^m_{sloc}}/\sum_{k=1}^{n} P^1_{M^k_{sloc}}$, where n is the number of methods in project P^1. Therefore:

$$P^1_{M^1_{weight}} = P^1_{M^1_{sloc}}/(P^1_{M^1_{sloc}} + P^1_{M^2_{sloc}} + P^1_{M^3_{sloc}}) = 0.25$$

Similarly, $P^1_{M^2_{weight}} = 0.5, P^1_{M^3_{weight}} = 0.25$

Step 3 All methods with identical SLOC are grouped together. For example, methods $P^p_{M^i}, P^p_{M^j}$ and $P^p_{M^k}$ are grouped together if

$$P^p_{M^i_{sloc}} = P^p_{M^j_{sloc}} = P^p_{M^k_{sloc}}.$$

The entity aggregation of this method group

$$= P^p_{M^i_{weight}} + P^p_{M^j_{weight}} + P^p_{M^k_{weight}}.$$

In our case with P^1, the entity aggregation for the first method

group with SLOC 10

$$= P^1_{M^1_{weight}} + P^1_{M^3_{weight}} = 0.5,$$

and for the second method group with SLOC 20$= P^1_{M^2_{weight}} = 0.5$

Each group's SLOC value is added to a list called *slocs*. The aggregated value is added to another list called *aggr*. So, we have *slocs* = [10, 20], and *aggr* = [0.5, 0.5]

Step 4 For each entry i in *aggr*, we calculate the normalized entity aggregation $aggr^i/\phi$, where ϕ is the number of projects. We add this normalized value to another list called *norms*. Therefore, *norms* = [0.25, 0.25].

Step 5 For each project $p = 1, 2, .., \phi$, we repeat step 1 to 4. This produces ϕ different lists of *slocs* (called *list_slocs*) and *norms* (called *list_norms*). In our case, *list_slocs* = [[10, 20], [20, 10]], and *list_norms* = [[0.25, 0.25], [0.4, 0.1]]. We used this two lists, to calculate another two lists called *x_axis* and *y_axis*, using an algorithm suggested by Alves *et al.* [3]. For reproducibility, the algorithm is shared with our dataset package (algorithm/algorithm.png).

Step 6 From the algorithm, x_axis = [10, 20], y_axis = [0.35, 0.65]. We now sort *x_axis* in ascending order (and *y_axis* accordingly) to produce a cumulative line chart. Figure 3 shows the cumulative line chart after following all the steps for all of the 49 projects. From the line chart, we get three critical values to categorize a method either as small, or medium, or large, or very large. According to the Alves *et al.*'s approach, the first critical value in Figure 3 is 24 (from the x-axis), because it covers 70% of the y-axis. The second and the third values are 36 and 63, because they cover 80% and 90% of the y-axis respectively.

Based on these three critical values, Table 3 explains how to determine if a method is small, medium, large, or very large. For example, if a method is 24 SLOC or fewer, it is a small method while a method with 64 SLOC or more is a very large method. We conclude that 24 SLOC is not only small but also realistic, because majority of the methods can be written within 24 SLOC (Figure 3). This observation is true even without getters/setters, because we have removed them from our dataset, as explained in Section 3.5.
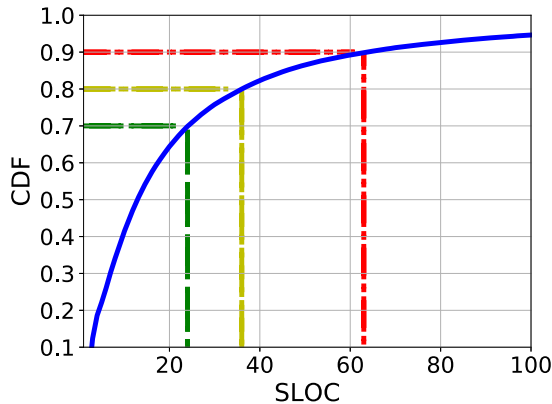

Figure 3: Cumulative line chart from Alves *et al.*'s approach.

Table 3: Determining the size of a method after applying Alves *et al.*'s approach in our dataset of 49 projects.

Size	Lower-bound	Upper-bound
Small	–	24
Medium	25	36
Large	37	63
Very Large	64	–

4.2.2 Evaluating the upper bound for maintenance We found that 24 SLOC is a reasonable upper bound of method size, but we need to evaluate if this bound makes any real difference with the other method sizes (medium, large, and very large) when compared against change- and bug-proneness. To understand how we evaluate this for the 49 different projects, let us consider Figure 4 as a demonstration for the elasticsearch project. In the first group of four box-plots, we compare the #revisions distributions, where each box-plot contains all the #revisions for all the methods with a specific size (e.g., the first box-plot shows the #revisions distribution for all the small methods with SLOC ≤ 24). Clearly, #revisions increases with the increase in method sizes.

To observe if these differences in #revisions are statistically different and what are the effect sizes of those differences, we apply three comparisons. These comparisons are between distributions of #revisions for i) small methods and medium methods, ii) medium methods and large methods, and iii) large methods and very large methods. According to the Wilcoxon rank sum test, in all the three cases, the differences are statistically different ($p \leq 0.05$). The Cliff's Delta effect size is *large* between small and medium methods, and *small* for the other two cases. These observations are true for the other four maintenance indicators as well.

Are these observations generalizable? According to our comparison approach, for each indicator (e.g., #revisions) we have a total of 147 comparisons (3 comparisons per project × 49 projects). Table 4 shows that for #revisions, in 82.98% of cases the comparisons are statistically different. This percent is the highest for the EditDistance, and the lowest for the #BuggyCommits. Many of our projects, however, do not have enough methods in each size category to make a reliable comparison. Therefore, we also consider only the top 20 projects, after ranking all the projects based on their number of methods. Table 4 shows that the percent of cases with significant statistical differences are much higher for the top 20 projects.

We also calculated the effect sizes of the differences, as presented and described in Table 5. For all the four change-proneness indicators, no project exhibits negligible effect size when compared small methods with medium size methods. In fact, most of the cases are within medium to large effect sizes. For example, for #revisions, 55.1% of the effect sizes are medium, and 34.69% are large. This observation, however, is not true when compared for the other size categories: **Medium-Large**, and **Large-Very Large**. For these two groups, most of the effect sizes are within negligible and small. These observations suggest that converting a medium method to a large method, or a large method to a very large method are not as harmful as converting a small method to medium method. Consistent with the observation from RQ1, for the bug-proneness indicator (i.e., #BuggyCommits), the observation is different than
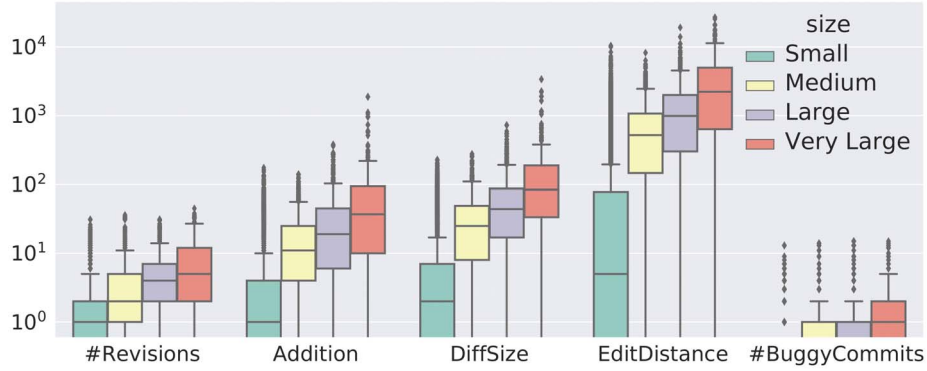
Figure 4: For the elasticsearch project, this graph compares the distributions of different maintenance indicators after grouping them according to the four different method sizes.

Table 4: Wilcoxon rank sum test to show the percentage of cases where the SLOC categorization exhibit statistically significantly different maintenance effort.

Sample Projects	# Revisions	Addition	DiffSize	EditDistance	#BuggyCommits
All	82.98%	86.52%	87.23%	87.94%	77.30%
Top 20	96.67%	96.67%	96.67%	96.67%	91.67%

Table 5: Cliff's Delta Effect sizes after comparing methods in different size categories (e.g., Small-Medium indicates comparison between small and medium size methods). N refers to Negligble, S refers to Small, M refers to Medium, and L refers to Large effect size. For example, the first column of the first row compares the #revisions distributions between small and medium size methods and shows the percent of differences with negligible, small, medium, and large effect sizes.

Indicator	Small-Medium				Medium-Large				Large-Very Large			
	N	S	M	L	N	S	M	L	N	S	M	L
# Revisions	0.00	10.20	55.10	34.69	35.42	58.33	2.08	4.17	29.55	59.09	6.82	4.55
Addition	0.00	6.12	20.41	73.47	29.17	60.42	8.33	2.08	27.27	59.09	11.36	2.27
DiffSize	0.00	6.12	6.12	87.76	22.92	66.67	4.17	6.25	20.45	52.27	22.73	4.55
EditDistance	0.00	4.08	6.12	89.8	18.75	60.42	16.67	4.17	22.73	45.45	25.0	6.82
#BuggyCommits	38.78	57.14	4.08	0.00	70.83	22.92	6.25	0.00	52.27	43.18	2.27	2.27

the observations with the four change-proneness indicators. Most of the differences are now within negligible and small effect sizes. And yet, the difference in the **Small-Medium** group is more meaningful than the others, because it has the lowest percent (38.78%) of negligible differences. We conclude that, a threshold < 24 would be less achievable (Figure 3), whereas a threshold > 24 would incur significantly higher maintenance effort (Table 5).

> **Summary:** Methods of 24 SLOC or fewer exhibit less change- and bug-proneness. Therefore, developers should strive to keep their methods within 24 SLOC.

4.3 RQ3: Should developers decompose large methods?

Some methods are inherently large, due to the complicated tasks they implement. Would refactoring these complex methods into smaller methods (≤ 24 SLOC) actually improve maintainability, or just distribute it? This is important, because decomposing large

methods would increase dependency and coupling score (more fan-in and fan-out) of a software project, which has been shown harmful for software maintenance [56]. To investigate this issue, we would require two versions of each software project. In one version, method size was not controlled, and in the other each method with SLOC > 24 was decomposed. Unfortunately, such a project does not exist. Therefore, *we consider that any group of small methods that can easily be merged to a larger method are the result of a large method decomposition.* We then compare the change- and bug-proneness of individual methods having SLOC > 24 with the summation of change- and bug-proneness of small methods (SLOC ≤ 24) that are candidate for merging.

Rules for merging: any method B *is a candidate to merge with any method* A*, if and only if i)* B *is called only by method* A*, ii) both* A *and* B *have SLOC ≤ 24, and iii) the SLOC limit for* A *will not apply if* A *is a result of previous merging.*

While the first two rules are intuitive, *rule iii* is to produce arbitrarily large SLOC so that we can compare with similar sized individual methods. To better understand the procedure, let us consider the call graph in Figure 5. For simplicity, we show #revisions as a method's attribute, although the procedure is the same for the other maintenance indicators. Here, methods M1, M2, M3, and M4 are candidate for decomposition, because their SLOCs are larger than 24. On the other hand, with our *rules for merging*, method M6 is a candidate to merge with M5, producing a total SLOC 40, and total #revisions 2. Similarly, we can merge M7 with M5, and M9 with M7. We can not merge M5 with M4, because M4 violates *rule ii* (SLOC > 24). M8 can not be merged with any methods because it is called by more than one methods (*rule i*). We, however, can merge M9 with M7 and then the resulted method with M5, because none of this methods has SLOC > 24 when considered individually (*rule iii*). Table 6 shows the two groups of methods with their SLOCs and #revisions. The *Selected* column is described later.
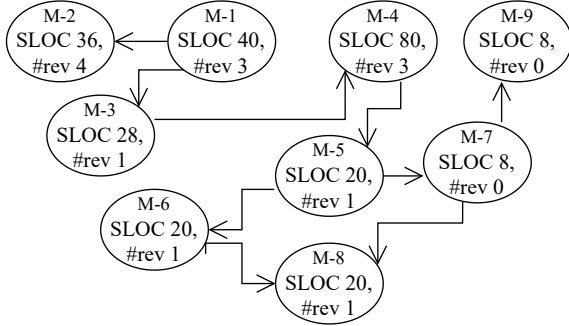
Figure 5: Example of a call graph.

Table 6: Grouping methods from Figure 5 either as individual or merged methods according to our *rules for merging*.

Individual Method	SLOC	#revisions	Selected
M1	40	3	Yes
M2	36	4	Yes
M3	28	1	Yes
M4	80	3	No
Merged Methods	**Total SLOC**	**Total #revisions**	**Selected**
M5, M6	40	2	Yes
M5, M7	28	1	Yes
M7, M9	16	0	No
M5, M7, M9	36	1	Yes

From Table 6, we can now compare the two distributions of #revisions for deciding which group, between the individual and merged methods, undergoes less #revisions. Unfortunately, the SLOC distributions of this two groups are different. Therefore, the comparison will not be accurate, as SLOC influences all the maintenance indicators (RQ1, and RQ2). This problem is depicted in Figure 6 for the commons-io project. The second group of box-plots suggests that #revisions is higher for merged methods (i.e., decomposition is harmful). The first group of box-plots, however, shows that merged
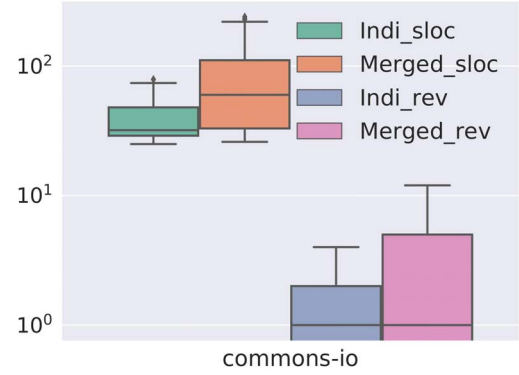

Figure 6: Comparing #revisions distributions between the two groups for the commons-io project. Ind~Individual, rev~#revisions. Here, SLOC distributions are not similar, leading to inaccurate comparison of maintenance indicators.

methods SLOC distribution is much higher than individual methods, leading to inaccurate comparison between the two distributions of #revisions. We solve this problem by removing entries that do not have a common SLOC, as presented in the *selected* column in Table 6. For example, the M4 method is not selected because its SLOC (80) does not match with any other methods in the second group of merged methods. For the same reason, the merged method M7, M9 is not selected. After removing these unmatched entries, we can now accurately compare the maintenance indicators of the individual and the merged methods.

We have used the JavaSymbolSolver [41] to produce the call graphs of our 49 projects. Producing the call graphs for every commits of a project is difficult and very time consuming. We will need to produce ~25,000 call graphs for the hadoop project alone, because of its ~25,000 commits. As a result, we selected only one commit per project for building the call graph and tracking the revision history of the project from that commit. Therefore, each project was checked out to two years before its current commit (current commit is presented in Table 1). This 2 years provides enough time to all the methods to undergo changes.

The JavaSymbolSolver, unfortunately, was not able to produce call graphs for some of our projects including elasticsearch, flink etc., mostly due to StackOverflowError exceptions. From the *issue* tracking system, we found this problem common for the JavaSymbolSolver [40]. Additionally, after removing entries with unmatched SLOCs, projects with very small number of methods, such as junit4, had too few samples to compare the maintenance indicators among the individual and the merged methods. After filtering out these two categories of projects from analysis, Table 7 shows the results for the remaining projects. Most of the time the *candidate for merge* methods were less expensive than the individual methods (i.e., more deep-blue cells than purple cells). More specifically, out of 115 cells, 84 of them are blue (73%). If we consider cells with non-negligible effect size, the percent of blue cell becomes 87%. This suggests that decomposing large methods will not increase maintenance effort of software systems.

Table 7: Distribution of different maintenance indicators of individual methods and merged methods are compared using the Cliff's Delta. Effect sizes are N~Negligible, and S~Small. A deep-blue cell means merged method has lower value for that specific indicator—suggesting decomposition is less expensive. For example, for hadoop, the cell for #revisions is deep-blue and the value is *N*. It means, according to the Cliff's Delta, #revisions is less for the merged methods with negligible effect size.

Project	# Samples	# Revisions	# Additions	DiffSize	EditDistance	Bugs
hadoop	1304	*N*	*N*	*N*	*N*	*N*
weka	1274	*N*	*N*	*N*	*N*	*N*
hibernate-orm	841	*N*	*N*	*N*	*N*	*N*
hbase	742	*N*	*N*	*N*	*N*	*N*
docx4j	498	*N*	*N*	*N*	*N*	*N*
hazelcast	493	*N*	*S*	*S*	*S*	*N*
netty	445	*N*	*N*	*N*	*N*	*N*
jgit	413	*S*	*S*	*S*	*S*	*S*
wicket	412	*N*	*N*	*N*	*N*	*N*
voldemort	184	*N*	*N*	*N*	*N*	*N*
openmrs-core	170	*N*	*N*	*N*	*N*	*N*
wildfly	163	*N*	*N*	*N*	*N*	*N*
mongo-java-driver	144	*S*	*S*	*S*	*S*	*N*
checkstyle	137	*S*	*S*	*S*	*S*	*N*
javaparser	105	*N*	*N*	*N*	*N*	*N*
hibernate-search	78	*N*	*N*	*N*	*N*	*N*
spring-boot	72	*N*	*N*	*N*	*N*	*N*
facebook-android-sdk	72	*N*	*N*	*N*	*N*	*N*
lombok	50	*N*	*N*	*N*	*N*	*N*
jna	45	*S*	*S*	*S*	*S*	*N*
hector	45	*S*	*S*	*S*	*S*	*N*
okhttp	41	*S*	*S*	*S*	*S*	*S*
commons-io	20	*N*	*N*	*S*	*N*	*N*

Summary: Developers should decompose large methods so that none of the methods are larger than 24 SLOC. A group of small methods, with sum SLOC x, are generally collectively less change- and bug-prone than an individual large method with SLOC x.

4.4 RQ4: Does controlling for size controls other quality metrics?

So far, we have studied the impact of size on change- and bug-proneness. Change- and bug-proneness, however, are not sufficient for understanding software maintenance as a whole. In this paper, we consider three more code quality factors that could potentially impact the maintenance of a software project. In particular, we investigate, if controlling size (i.e., keeping method size within 24 SLOC) also controls these other three code quality metrics that have been associated with software maintenance.

1) Testability: Developers and testers write test methods to evaluate whether a source method produces expected results. Writing test methods can be difficult for source methods with too many independent paths. Therefore, the popular McCabe metric is often used as an indication of the testability of a source method [12, 92]. The McCabe metric counts the number of independent paths in a code component with the formula: $1 + \#predicates$ [52]. McCabe, however, does not consider the number of control variables in a predicate. This is a limitation, because a predicate with more control variables should be considered more complex [43] and less testable

than a predicate with a single control variable. McClure [53], on the other hand, considers all the control variables and comparisons in a code metric, and can be used as an alternative of McCabe. We consider both McCabe and McClure as indications of the testability of a source method.

2) Readability: According to Martin Fowler, "Any fool can write code that a computer can understand. Good programmers write code that humans can understand." One of the most frequent activities in software development and maintenance is *code reading* [72]. Readability is a measure of how easy (or difficult) it is to read and understand a source code [13, 15, 42, 63], which has significant impact on code maintenance. In this paper, we have used the publicly available readability tool implemented by Buse *et al.* [15] to examine the relationship between method size and readability. This tool calculates a readability score for a given method from 0 (least readable) to 1 (completely readable code).

3) Maintainability Index: We also include the maintainability index (MI) metric, which is industrially popular as a measurement of code quality. MI is calculated as:

$$171 - 5.2 * ln(Halstead\,Volume) - 0.23 * (Cyclomatic\,Complexity) - 16.2 * ln(Lines\,of\,Code)$$

This is the evolved form of the original equation proposed by Oman and Hagemeister [59], and different forms of it have been adopted
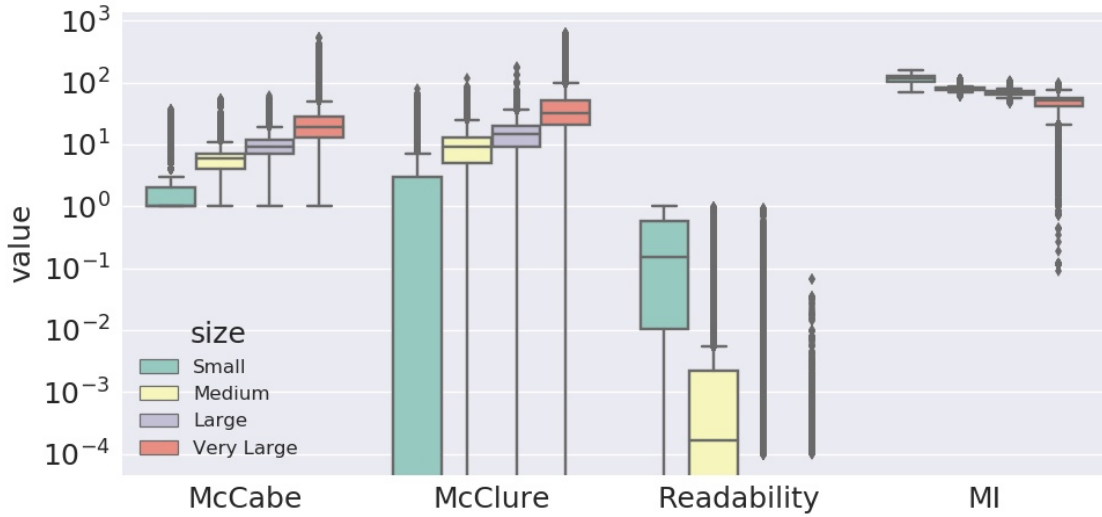
Figure 7: For the Elasticsearch project, this graph compares the distributions of different quality metrics after grouping them according to the four different method sizes. A high value for McCabe (and McClure) indicates a high maintenance effort, whereas a high readability (and maintenance index) value indicates low maintenance effort.

Table 8: Cliff's Delta Effect sizes after comparing methods in different size categories (e.g., Small-Medium indicates comparison between small and medium size methods). Effect sizes are N~Negligible, S~Small, M~Medium, and L~Large. For example, the first column of the first row compares the McCabe distributions between small and medium size methods and shows the percent of differences with different effect sizes.

Metrics	Small-Medium				Medium-Large				Large-Very Large			
	N	S	M	L	N	S	M	L	N	S	M	L
McCabe	0.00	2.04	0.00	97.96	6.12	4.08	16.33	73.47	0.00	2.22	8.89	88.89
McClure	0.00	2.04	0.00	97.96	2.04	4.08	59.18	34.69	0.00	8.89	17.78	73.33
Readability	0.00	2.04	0.00	97.96	0.00	14.29	40.82	44.9	44.44	44.44	8.89	2.22
MI	0.00	0.00	0.00	100.0	0.00	0.00	0.00	100.0	0.00	0.00	0.00	100.0

by popular tools such as Verifysoft technology [87], and Visual Studio [55].

Considering the Elasticsearch project in Figure 7, we can see that by controlling the method size, we can control the four above mentioned code quality factors in most cases. Similar to the RQ2, we compare these quality factors by grouping the method sizes. The Cliff's Delta effect size is large between all the compared method groups, except for the readability metric. For the readability metric, the effect sizes are large, small, and negligible between **Small-Medium**, **Medium-Large**, and **Large-Very Large** method groups, respectively. Table 8 shows the findings for all the 49 projects. Similar to RQ2, the difference in the **Small-Medium** group is more meaningful than the others; the effect size for all of the four quality factors are large in at 97.96% to 100% of the projects.

Summary: By controlling size we can control other complex code quality metrics, such as McCabe, McClure, Readability, and Maintainability index.

5 Discussion

Understanding code metrics and their relation with software maintenance is a forty years old problem [48]. While there is debate about the effectiveness of many of the famous code metrics (e.g., C&K), the research community is in complete agreement about the usefulness of size as an indicator of future maintenance. Therefore, research about size and maintenance is not new. It is obvious from all the previous studies that developers should keep the size of their code unit small, similar to our observation in RQ1. Unfortunately, there is no concrete evidence-based recommendations about how small the size should be. Although Visser *et al.* [89] recommended that developers should keep their method within 15 source lines of code, this recommendation was from intuition only, not based on evidence. Also, in RQ2, we have provided clear evidence that keeping method size always within 15 SLOC is less realistic.

This paper is unique in that context, because we provide evidence-based recommendation that *developers should keep their method size within 24 SLOC (RQ2)*. In RQ2, we also show that keeping method

size < 24 SLOC is less realistic, whereas method with size > 24 SLOC is less maintainable. Therefore, this 24 SLOC bound can be used for trade-offs between feasibility and maintainability. We also provide evidence that inherently large methods should be refactored to a group of helper methods, each within 24 SLOC (RQ3). Encouragingly, controlling for size, in most cases, controls other code quality factors, such as testability and readability (RQ4).

Our findings can be used for improving code metric tools such as *checkstyle*. For example, the current default limit of method size in *checkstyle* is 150 [17]. Our research shows that such a large method would be extremely maintenance prone, and this limit should be set to 24 only. Our methodology can be extended to find maintainable and feasible limit for other popular code metrics, such as McCabe, nested block depth, and maintainability index.

6 Threats to validity

Several threats, however, may impact the validity of our results.

External validity is hampered by our selection of software projects. Our observations with open-source software may not be consistent with closed-source projects. The selected 49 projects, however, are popular and have been widely used by other code metrics studies.

Internal validity is hampered by the statistical approaches. For example, the use of correlation coefficients might be inaccurate for unknown confounding factors. For finding the three critical values, we have relied on the Alves *et al.*'s approach. Other critical values may produce more useful results.

Construct validity is affected by our approach to define bug-proneness. Capturing bug related keywords from the commit messages for measuring bug-proneness is only ~80% accurate [78]. Also, the CodeShovel tool[28] we used for capturing method history is accurate only for 90% of the methods [29].

Conclusion validity is affected by the selected maintenance and quality indicators. Change-/bug-proneness, and the quality metrics we used may not be sufficient to understand software maintenance.

7 Conclusion

Significant research has been done to understand code metrics impact on software maintenance for building accurate predictive models. This paper investigated the relationship between method size and maintenance effort, because size has been frequently reported as the most common, easy-to-measure, and the most effective code metric for understanding maintenance. We established that developers should be careful about their method size for reducing maintenance effort, such as change- and bug-proneness. We also showed that controlling for method size, in most cases, controls other code quality metrics, such as testability, and readability. In general, developers should aim to keep their Java method within 24 source lines of code. They should also refactor inherently large methods to group of smaller methods, even if it increases the overall coupling of a system.

The methodology of our study can be extended for other programming languages, and for other metrics (e.g., McCabe, and C&K). Studies can also investigate other granularity, such as class/file and module levels. We hope such studies can help us better understand the relationship between code metrics and maintenance for building ever more accurate maintenance models.

References

[1] Nahla J. Abid, Bonita Sharif, Natalia Dragan, Hend Alrasheed, and Jonathan I. Maletic. 2019. Developer Reading Behavior While Summarizing Java Methods: Size and Context Matters. In *Proceedings of the 41st International Conference on Software Engineering* (Montreal, Quebec, Canada). 384–395.
[2] Fernando Brito Abreu and Rogério Carapuça. 1994. Object-oriented software engineering: Measuring and controlling the development process. In *Proceedings of the 4th international conference on software quality*, Vol. 186.
[3] T. L. Alves, C. Ypma, and J. Visser. 2010. Deriving metric thresholds from benchmark data. In *IEEE International Conference on Software Maintenance*. 1–10.
[4] Mauricio Finavaro Aniche, Christoph Treude, Andy Zaidman, Arie van Deursen, and Marco Aurélio Gerosa. 2016. SATT: Tailoring Code Metric Thresholds for Different Software Architectures. In *16th IEEE International Working Conference on Source Code Analysis and Manipulation,2016, Raleigh, NC, USA, October 2-3, 2016*. 41–50.
[5] V. Antinyan, M. Staron, J. Derehag, M. Runsten, E. Wikström, W. Meding, A. Henriksson, and J. Hansson. 2015. Identifying complex functions: By investigating various aspects of code complexity. In *2015 Science and Information Conference (SAI)*. 879–888.
[6] V. Antinyan, M. Staron, W. Meding, P. Österström, E. Wikstrom, J. Wranker, A. Henriksson, and J. Hansson. 2014. Identifying risky areas of software code in Agile/Lean software development: An industrial experience report. In *IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering*. 154–163.
[7] Vard Antinyan, Miroslaw Staron, and Anna Sandberg. 2017. Evaluating Code Complexity Triggers, Use of Complexity Measures and the Influence of Code Complexity on Maintenance Time. *Empirical Softw. Engg.* 22, 6 (Dec. 2017), 3057–3087.
[8] Francesca Arcelli Fontana, Vincenzo Ferme, Marco Zanoni, and Aiko Yamashita. 2015. Automatic Metric Thresholds Derivation for Code Smell Detection. In *2015 IEEE/ACM 6th International Workshop on Emerging Trends in Software Metrics*. 44–53.
[9] Dimitrios Athanasiou, Ariadi Nugroho, Joost Visser, and Andy Zaidman. 2014. Test Code Quality and Its Relation to Issue Handling Performance. *IEEE Trans. Software Eng.* 40, 11 (2014), 1100–1125.
[10] R. K. Bandi, V. K. Vaishnavi, and D. E. Turk. 2003. Predicting maintenance performance using object-oriented design complexity metrics. *IEEE Transactions on Software Engineering* 29, 1 (2003), 77–87.
[11] Jennifer Bauer, Janet Siegmund, Norman Peitek, Johannes C Hofmeister, and Sven Apel. 2019. Indentation: simply a matter of style or support for program comprehension?. In *IEEE/ACM 27th International Conference on Program Comprehension*. 154–164.
[12] Armin Beer and Michael Felderer. 2018. Measuring and Improving Testability of System Requirements in an Industrial Context by Applying the Goal Question Metric Approach. In *Proceedings of the 5th International Workshop on Requirements Engineering and Testing* (Gothenburg, Sweden) *(RET '18)*. 25–32.
[13] J. Börstler and B. Paech. 2016. The Role of Method Chains and Comments in Software Readability and Comprehension—An Experiment. *IEEE Transactions on Software Engineering* 42, 9 (2016), 886–898.
[14] J. M. Brittain. 1982. Pitfalls of user research, and some neglected areas. *Social Science Information Studies* 2, 3 (1982), 139–148.
[15] Raymond P. L. Buse and Westley R. Weimer. 2010. Learning a Metric for Code Readability. *IEEE Trans. Softw. Eng.* 36, 4 (July 2010), 546–558.
[16] Gemma Catolino, Fabio Palomba, Andrea De Lucia, Filomena Ferrucci, and Andy Zaidman. 2018. Enhancing change prediction models using developer-related factors. *Journal of Systems and Software* 143 (2018), 14–28.
[17] Checkstyle. 2022. Properties, Method Length. https://checkstyle.sourceforge.io/config.html. [Online; last accessed 06-Jan-2022].
[18] S. R. Chidamber and C. F. Kemerer. 1994. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering* 20, 6 (1994), 476–493.
[19] Shaiful Chowdhury, Stephanie Borle, Stephen Romansky, and Abram Hindle. 2019. GreenScaler: training software energy models with automatic test generation. *Empirical software engineering : an international journal* 24, 4 (2019), 1649–1692.
[20] L. Cruz, R. Abreu, J. Grundy, L. Li, and X. Xia. 2019. Do Energy-Oriented Changes Hinder Maintainability?. In *2019 IEEE International Conference on Software Maintenance and Evolution*. 29–40.
[21] B. Curtis, S. B. Sheppard, P. Milliman, M. A. Borst, and T. Love. 1979. Measuring the Psychological Complexity of Software Maintenance Tasks with the Halstead and McCabe Metrics. *IEEE Transactions on Software Engineering* SE-5, 2 (1979), 96–104.

[22] D. P. Darcy, C. F. Kemerer, S. A. Slaughter, and J. E. Tomayko. 2005. The structural complexity of software an experimental test. *IEEE Transactions on Software Engineering* 31, 11 (2005), 982–995.

[23] K. El Emam, S. Benlarbi, N. Goel, and S. N. Rai. 2001. The confounding effect of class size on the validity of object-oriented metrics. *IEEE Transactions on Software Engineering* 27, 7 (2001), 630–650.

[24] Emanuel Giger, Marco D'Ambros, Martin Pinzger, and Harald C. Gall. 2012. Method-Level Bug Prediction. In *Proceedings of the ACM-IEEE International Symposium on Empirical Software Engineering and Measurement* (Lund, Sweden) *(ESEM '12)*. 171–180.

[25] Yossi Gil and Gal Lalouche. 2016. When do Software Complexity Metrics Mean Nothing? – When Examined out of Context. *Journal of Object Technology* 15, 1 (Feb. 2016), 2:1–25.

[26] Yossi Gil and Gal Lalouche. 2017. On the Correlation between Size and Metric Validity. *Empirical Software Engineering* 22, 5 (Oct. 2017), 2585–2611.

[27] G.K. Gill and C.F. Kemerer. 1991. Cyclomatic complexity density and software maintenance productivity. *IEEE Transactions on Software Engineering* 17, 12 (1991), 1284–1288.

[28] Felix Grund, Shaiful Chowdhury, Nick C. Bradley, Braxton Hall, and Reid Holmes. 2021. CodeShovel: A Reusable and Available Tool for Extracting Source Code Histories. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. 221–222.

[29] Felix Grund, Shaiful Chowdhury, Nick C. Bradley, Braxton Hall, and Reid Holmes. 2021. CodeShovel: Constructing Method-Level Source Code Histories. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. 1510–1522.

[30] Tracy Hall, Min Zhang, David Bowes, and Yi Sun. 2014. Some Code Smells Have a Significant but Small Effect on Faults. *ACM Trans. Softw. Eng. Methodol.* 23, 4 (sep 2014).

[31] Ilja Heitlager, Tobias Kuipers, and Joost Visser. 2007. A Practical Model for Measuring Maintainability. In *Proceedings of the 6th International Conference on Quality of Information and Communications Technology*. 30–39.

[32] I. Herraiz, J. M. Gonzalez-Barahona, and G. Robles. 2007. Towards a Theoretical Model for Software Growth. In *Fourth International Workshop on Mining Software Repositories*. 21–21.

[33] K. Herzig and A. Zeller. 2013. The impact of tangled code changes. In *2013 10th Working Conference on Mining Software Repositories*. 121–130.

[34] M. Hess and J. Kromrey. 2004. Robust Confidence Intervals for Effect Sizes: A Comparative Study of Cohen's d and Cliff's Delta Under Non-normality and Heterogeneous Variances.

[35] A. Hindle, M. W. Godfrey, and R. C. Holt. 2008. Reading Beside the Lines: Indentation as a Proxy for Complexity Metric. In *16th IEEE International Conference on Program Comprehension*. 133–142.

[36] J. Hofmeister, J. Siegmund, and D. V. Holt. 2017. Shorter identifier names take longer to comprehend. In *IEEE 24th International Conference on Software Analysis, Evolution and Reengineering*. 217–227.

[37] Laura Inozemtseva and Reid Holmes. 2014. Coverage is Not Strongly Correlated with Test Suite Effectiveness. In *Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India). 435–445.

[38] Md Rakibul Islam and Minhaz F. Zibran. 2020. How Bugs Are Fixed: Exposing Bug-Fix Patterns with Edits and Nesting Levels. In *Proceedings of the 35th Annual ACM Symposium on Applied Computing* (Brno, Czech Republic). 1523–1531.

[39] Javaparser. 2022. Java 1-15 Parser and Abstract Syntax Tree for Java. https://github.com/javaparser/javaparser. [Online; last accessed 06-Jan-2022].

[40] JavaSymbolSolver. 2022. Issues. https://github.com/javaparser/javaparser/issues?q=stackoverflow. [Online; last accessed 06-Jan-2022].

[41] JavaSymbolSolver. 2022. Java 1-15 Parser and Abstract Syntax Tree for Java. https://github.com/javaparser/javasymbolsolver. [Online; last accessed 06-Jan-2022].

[42] J. Johnson, S. Lubo, N. Yedla, J. Aponte, and B. Sharif. 2019. An Empirical Study Assessing Source Code Readability in Comprehension. In *2019 IEEE International Conference on Software Maintenance and Evolution*. 513–523.

[43] D. Kafura and G. R. Reddy. 1987. The Use of Software Complexity Metrics in Software Maintenance. *IEEE Transactions on Software Engineering* SE-13, 3 (1987), 335–343.

[44] Foutse Khomh, Massimiliano Di Penta, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. 2012. An Exploratory Study of the Impact of Antipatterns on Class Change- and Fault-Proneness. *Empirical Softw. Engg.* 17, 3 (jun 2012), 243–275.

[45] Masanari Kondo, Daniel M. German, Osamu Mizuno, and Eun-Hye Choi. 2020. The impact of context metrics on just-in-time defect prediction. *Empirical software engineering* 25, 1 (2020), 890–939.

[46] Al Lake and Curtis R. Cook. 1994. *Use of Factor Analysis to Develop OOP Software Complexity Metrics.* Technical Report. USA.

[47] D. Landman, A. Serebrenik, and J. Vinju. 2014. Empirical Analysis of the Relationship between CC and SLOC in a Large Corpus of Java Methods. In *IEEE International Conference on Software Maintenance and Evolution*. 221–230.

[48] V. Lenarduzzi, A. Sillitti, and D. Taibi. 2017. Analyzing Forty Years of Software Maintenance Models. In *International Conference on Software Engineering Companion (ICSE-C)*. 146–148.

[49] Vladimir I Levenshtein. 1966. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, Vol. 10. 707–710.

[50] Huihui Liu, Xufang Gong, Li Liao, and Bixin Li. 2018. Evaluate How Cyclomatic Complexity Changes in the Context of Software Evolution. In *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, Vol. 02. 756–761.

[51] D. Matter, A. Kuhn, and O. Nierstrasz. 2009. Assigning bug reports using a vocabulary-based expertise model of developers. In *2009 6th IEEE International Working Conference on Mining Software Repositories*. 131–140.

[52] T. J. McCabe. 1976. A Complexity Measure. *IEEE Transactions on Software Engineering* SE-2, 4 (1976), 308–320.

[53] Carma L. McClure. 1978. A Model for Program Complexity Analysis. In *Proceedings of the 3rd International Conference on Software Engineering* (Atlanta, Georgia, USA). 149–157.

[54] T. Menzies, J. Greenwald, and A. Frank. 2007. Data Mining Static Code Attributes to Learn Defect Predictors. *IEEE Transactions on Software Engineering* 33, 1 (2007), 2–13.

[55] Microsoft. 2022. Code Metrics Maintainability Index. https://docs.microsoft.com/en-us/visualstudio/code-quality/code-metrics-maintainability-index-range-and-meaning?view=vs-2022. [Online; last accessed 06-Jan-2022].

[56] R. Mo, Y. Cai, R. Kazman, L. Xiao, and Q. Feng. 2016. Decoupling Level: A New Metric for Architectural Maintenance Complexity. In *2016 IEEE/ACM 38th International Conference on Software Engineering*. 499–510.

[57] Audris Mocku and Lawrence G. Votta. 2000. Identifying reasons for software changes using historic databases. In *Proceedings 2000 International Conference on Software Maintenance*. 120–130.

[58] A. Monden, D. Nakae, T. Kamiya, S. Sato, and K. Matsumoto. 2002. Software quality analysis by code clones in industrial legacy software. In *Proceedings IEEE Symposium on Software Metrics*. 87–94.

[59] P. Oman and J. Hagemeister. 1992. Metrics for assessing a software system's maintainability. In *Proceedings Conference on Software Maintenance 1992*. 337–344.

[60] Fabio Palomba, Andy Zaidman, Rocco Oliveto, and Andrea De Lucia. 2017. An Exploratory Study on the Relationship between Changes and Refactoring. In *Proceedings of the 25th International Conference on Program Comprehension* (Buenos Aires, Argentina). 176–185.

[61] J. Pantiuchina, M. Lanza, and G. Bavota. 2018. Improving Code: The (Mis) Perception of Quality Metrics. In *IEEE International Conference on Software Maintenance and Evolution*. 80–91.

[62] Luca Pascarella, Fabio Palomba, and Alberto Bacchelli. 2020. On the performance of method-level bug prediction: A negative result. *Journal of Systems and Software* 161 (2020).

[63] Daryl Posnett, Abram Hindle, and Premkumar Devanbu. 2011. A Simpler Model of Software Readability. In *Proceedings of the 8th Working Conference on Mining Software Repositories* (Waikiki, Honolulu, HI, USA). 73–82.

[64] Danijel Radjenović, Marjan Heričko, Richard Torkar, and Aleš Živkovič. 2013. Software fault prediction metrics: A systematic literature review. *Information and Software Technology* 55, 8 (2013), 1397 – 1418.

[65] Md Saidur Rahman and Chanchal K. Roy. 2017. On the Relationships Between Stability and Bug-Proneness of Code Clones: An Empirical Study. In *2017 IEEE 17th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 131–140.

[66] Paul Ralph and Ewan Tempero. 2018. Construct Validity in Software Engineering Research and Software Metrics. In *Proceedings of the 22nd International Conference on Evaluation and Assessment in Software Engineering 2018* (Christchurch, New Zealand). 13–23.

[67] Baishakhi Ray, Vincent Hellendoorn, Saheel Godhane, Zhaopeng Tu, Alberto Bacchelli, and Premkumar Devanbu. 2016. On the "Naturalness" of Buggy Code. In *Proceedings of the 38th International Conference on Software Engineering* (Austin, Texas) *(ICSE '16)*. 428–439.

[68] Baggen Robert, José P. Correia, Katrin Schill, and Joost Visser. 2012. Standardized code quality benchmarking for improving software maintainability. *Software Quality Journal* 20 (2012), 287–307.

[69] D. Romano and M. Pinzger. 2011. Using source code metrics to predict change-prone Java interfaces. In *2011 27th IEEE International Conference on Software Maintenance*. 303–312.

[70] Daniele Romano, Paulius Raila, Martin Pinzger, and Foutse Khomh. 2012. Analyzing the Impact of Antipatterns on Change-Proneness Using Fine-Grained Source Code Changes. In *2012 19th Working Conference on Reverse Engineering*. 437–446.

[71] S. Scalabrino, G. Bavota, C. Vendome, M. Linares-Vásquez, D. Poshyvanyk, and R. Oliveto. 2017. Automatically assessing code understandability: How far are we?. In *32nd IEEE/ACM International Conference on Automated Software Engineering*. 417–427.

[72] S. Scalabrino, M. Linares-Vásquez, D. Poshyvanyk, and R. Oliveto. 2016. Improving code readability models with textual features. In *IEEE 24th International Conference on Program Comprehension*. 1–10.

[73] Ingo Scholtes, Pavlin Mavrodiev, and Frank Schweitzer. 2016. From Aristotle to Ringelmann: a large-scale analysis of team productivity and coordination in Open Source Software projects. *Empirical software engineering : an international journal* 21, 2 (2016), 642–683.

[74] M. Shepperd. 1988. A critique of cyclomatic complexity as a software metric. *Software Engineering Journal* 3, 2 (1988), 30–36.

[75] Emad Shihab, Ahmed E. Hassan, Bram Adams, and Zhen Ming Jiang. 2012. An Industrial Study on the Risk of Software Changes. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering* (Cary, North Carolina).

[76] Y. Shin, A. Meneely, L. Williams, and J. A. Osborne. 2011. Evaluating Complexity, Code Churn, and Developer Activity Metrics as Indicators of Software Vulnerabilities. *IEEE Transactions on Software Engineering* 37, 6 (2011), 772–787.

[77] D. I. K. Sjøberg, A. Yamashita, B. C. D. Anda, A. Mockus, and T. Dybå. 2013. Quantifying the Effect of Code Smells on Maintenance Effort. *IEEE Transactions on Software Engineering* 39, 8 (2013), 1144–1156.

[78] D. Spadini, F. Palomba, A. Zaidman, M. Bruntink, and A. Bacchelli. 2018. On the Relation of Test Smells to Software Code Quality. In *2018 IEEE International Conference on Software Maintenance and Evolution*. 1–12.

[79] D. Ståhl, A. Martini, and T. Mårtensson. 2019. Big Bangs and Small Pops: On Critical Cyclomatic Complexity and Developer Integration Behavior. In *2019 IEEE/ACM 41st International Conference on Software Engineering: (ICSE-SEIP)*. 81–90.

[80] StackExchange. 2022. What is the ideal length of a method for you? https://softwareengineering.stackexchange.com/questions/133404/what-is-the-ideal-length-of-a-method-for-you. [Online; last accessed 06-Jan-2022].

[81] Chakkrit Tantithamthavorn and Ahmed E. Hassan. 2018. An Experience Report on Defect Modelling in Practice: Pitfalls and Challenges. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice* (Gothenburg, Sweden). 286–295.

[82] A. Terceiro, L. R. Rios, and C. Chavez. 2010. An Empirical Study on the Structural Complexity Introduced by Core and Peripheral Developers in Free Software Projects. In *Brazilian Symposium on Software Engineering*. 21–29.

[83] Henry C Thode. 2002. *Testing for normality*. Vol. 164. CRC press.

[84] Umesh Tiwari and Santosh Kumar. 2014. Cyclomatic Complexity Metric for Component Based Software. *SIGSOFT Softw. Eng. Notes* 39, 1 (Feb. 2014), 1–6.

[85] Ayşe Tosun, Ayşe Bener, Burak Turhan, and Tim Menzies. 2010. Practical considerations in deploying statistical methods for defect prediction: A case study within the Turkish telecommunications industry. *Information and Software Technology* 52, 11 (2010), 1242 – 1257.

[86] Alexander Trautsch, Steffen Herbold, and Jens Grabowski. 2020. Static source code metrics and static analysis warnings for fine-grained just-in-time defect prediction. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 127–138.

[87] VerifySoft. 2022. VerifySoft Maintainability Index. https://verifysoft.com/en_maintainability.html. [Online; last accessed 06-Jan-2022].

[88] Markos Viggiato, Johnatan Oliveira, Eduardo Figueiredo, Pooyan Jamshidi, and Christian Kästner. 2019. How Do Code Changes Evolve in Different Platforms? A Mining-based Investigation. In *2019 IEEE International Conference on Software Maintenance and Evolution*. 218–222.

[89] Joost Visser, Sylvan Rigal, Gijs Wijnholds, Pascal Van Eck, and Rob van der Leek. 2016. *Building Maintainable Software, C# Edition: Ten Guidelines for Future-Proof Code.* " O'Reilly Media, Inc.".

[90] Qingye Wang, Xin Xia, David Lo, and Shanping Li. 2019. Why is my code change abandoned? *Information and Software Technology* 110 (2019), 108 – 120.

[91] Kazuhiro Yamashita, Changyun Huang, Meiyappan Nagappan, Yasutaka Kamei, Audris Mockus, Ahmed E. Hassan, and Naoyasu Ubayashi. 2016. Thresholds for Size and Complexity Metrics: A Case Study from the Perspective of Defect Density. In *2016 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. 191–201.

[92] F. Zhang, A. Mockus, Y. Zou, F. Khomh, and A. E. Hassan. 2013. How Does Context Affect the Distribution of Software Maintainability Metrics?. In *IEEE International Conference on Software Maintenance*. 350–359.

[93] Yuming Zhou, Baowen Xu, and Hareton Leung. 2010. On the ability of complexity metrics to predict fault-prone classes in object-oriented systems. *Journal of Systems and Software* 83, 4 (2010), 660 – 674.

[94] Thomas Zimmermann, Rahul Premraj, and Andreas Zeller. 2007. Predicting Defects for Eclipse. In *Proceedings of the Third International Workshop on Predictor Models in Software Engineering*. 9.