Revisiting the debate

Are code metrics useful for measuring maintenance effort?

Chowdhury, Shaiful; Holmes, Reid; Zaidman, Andy; Kazman, Rick

DOI
10.1007/s10664-022-10193-8

Publication date
2022

Document Version
Final published version

Published in
Empirical Software Engineering

Empirical Software Engineering (2022) 27:158
https://doi.org/10.1007/s10664-022-10193-8

Revisiting the debate: Are code metrics useful for measuring maintenance effort?

Shaiful Chowdhury[1] · **Reid Holmes**[1] · **Andy Zaidman**[2] · **Rick Kazman**[3]

Accepted: 15 June 2022

Abstract
Evaluating and predicting software maintenance effort using source code metrics is one of the holy grails of software engineering. Unfortunately, previous research has provided contradictory evidence in this regard. The debate is still open: as a community we are not certain about the relationship between code metrics and maintenance impact. In this study we investigate whether source code metrics can indeed establish maintenance effort at the previously unexplored method level granularity. We consider ∼730K Java methods originating from 47 popular open source projects. After considering seven popular method level code metrics and using change proneness as a maintenance effort indicator, we demonstrate why past studies contradict one another while examining the same data. We also show that evaluation context is king. Therefore, future research should step away from trying to devise generic maintenance models and should develop models that account for the maintenance indicator being used and the size of the methods being analyzed. Ultimately, we show that future source code metrics can be applied reliably and that these metrics can provide insight into maintenance effort when they are applied in a judiciously context-sensitive manner.

Keywords Code metrics · Maintenance · McCabe · Code complexity

Communicated by: Mika Mäntylä

✉ Shaiful Chowdhury
shaifulc@cs.ubc.ca

Reid Holmes
rtholmes@cs.ubc.ca

Andy Zaidman
a.e.zaidman@tudelft.nl

Rick Kazman
kazman@hawaii.edu

[1] Department of Computer Science, University of British Columbia, Vancouver, Canada

[2] Department of Software Technology, Delft University of Technology, Delft, the Netherlands

[3] Department of Information Technology Management, University of Hawaii, Honolulu, Hawaii, USA

Published online: 17 August 2022

1 Introduction

The cost of software maintenance, which often exceeds the original cost of development (Börstler and Paech 2016), has long been a concern for the software industry (Kafura and Reddy 1987). This has led to considerable research estimating maintenance effort given the current state of a software project, to support project optimization and risk planning (e.g., Shin et al. 2011; Zhou et al. 2010; Ståhl et al. 2019; Cruz et al. 2019; Kondo et al. 2020; McClure 1978; Tosun et al. 2010). External software metrics—such as correctness, and performance—can indicate future maintenance effort, but they are difficult to collect (Gil and Lalouche 2017) and are often not available in early development phases. In contrast, source code metrics are easy to collect and are available throughout the software development life cycle. Therefore, a holy grail for the developer and the research community has been to predict future maintenance effort from code metrics (Gil and Lalouche 2017).

A number of code metrics (McCabe 1976; McClure 1978; Chidamber and Kemerer 1994; Lake and Cook 1994) have been used to predict maintenance indicators such as defect proneness, change proneness, and test difficulty. However, the true effectiveness of code metrics has been a subject of debate for the past forty years (e.g., Shepperd 1988; Gil and Lalouche 2016, 2017). While some studies showed that code metrics were good predictors (Johnson et al. 2019; Landman et al. 2014; Spadini et al. 2018; Bandi et al. 2003; Antinyan et al. 2014), in others the outcome was negative (Shepperd 1988; Scalabrino et al. 2017; Gil and Lalouche 2017). According to these critics, *other than program size* (El Emam et al. 2001; Gil and Lalouche 2017; Sjøberg et al. 2013), we do not have a single reliable code metric to estimate software maintenance effort (El Emam et al. 2001; Scalabrino et al. 2017; Gil and Lalouche 2017). In fact, size was found to be a good predictor of other code metrics (Herraiz et al. 2007), which is frustrating, because if size is the only valid metric, we can not prioritize maintenance activities between two components with similar sizes. Also, no good code metrics except size means that forty years of research (Lenarduzzi et al. 2017) on code metrics is potentially useless.

In this paper, we revisit the usefulness of code metrics so that we can inform both the research and developer communities as to whether code metrics are indeed good maintenance predictors, or if they should be abandoned. Also, we reproduce the previous contradictory claims as a means of guiding the research community on how to evaluate future code metrics reliably. For example, while some prior studies accounted for size (usually measured in Source Lines of Code, without comments and blank lines (Landman et al. 2014)) as a confounding factor for validating a metric, many did not. By using the complete history of ∼730K Java methods from 47 popular open source projects, along with seven source code metrics and four change proneness based maintenance indicators, we provide encouraging results. Our conclusion is that code metrics can in fact help estimate maintenance effort, such as change proneness, even when the confounding influence of size is eliminated. However, the impact of a code metric varies in different evaluation contexts. For example, *nested block depth* is not as good a predictor for large methods as it is for smaller methods, and metric performance can vary greatly based on the maintenance indicator used. We support our conclusion by answering the following research questions:

RQ1: Is the confounding effect of size a driving factor for the previous contradictory findings on the relationships between code metrics and maintenance effort?

Contribution 1: We show that size is indeed a significant factor in previous contradictory claims about the validity of code metrics. With our new method-level benchmark of

code metrics and change evolution, we reproduce three major prior observations: 1) Similar to some previous studies (e.g., Johnson et al. 2019; Tiwari and Kumar 2014; Subandri and Sarno 2017; Romano and Pinzger 2011), we first ignore size as a confounding factor, and show that code metrics are good maintenance predictors. 2) By dividing a metric value by size—a common (Suh and Neamtiu 2010; Shepperd 1988; Robert et al. 2012), but inaccurate approach (Gil and Lalouche 2017) for size normalization—we reproduce the claim that code metrics are good maintenance predictors. 3) We then show that the widely adopted size normalization approach fails to neutralize the size influence, and the maintenance impact of code metrics can still be explained by their correlation with size. This reproduces the criticism that without size influence there is no empirical evidence to support the validity of code metrics other than size itself (Gil and Lalouche 2017; Shepperd 1988).

RQ2: Why does the widely used size normalization approach not neutralize the size influence?

Contribution 2: Our expectation was similar to many other previous studies: a normalized metric (after dividing by size) should not have any correlation with size. To our surprise, we find that this is not the case. Some normalized metrics are negatively correlated with size while others are positively correlated with size. For example, normalized McCabe values are usually higher when the code size is small, thus producing a negative correlation between maintenance effort and size. For some others, the observation is opposite. We provide an explanation for why this unexpected observation is surprisingly common across all considered metrics.

RQ3: Can we apply simple regression analysis for observing the true (size neutralized) maintenance impact of code metrics (proposed in a recent study by Chen et al. 2020)?

Contribution 3: Our conclusion is encouraging. By a combination of bivariate (i.e., size∼maintenance) and multivariate (e.g., size + McCabe∼ maintenance) regression analysis we show that code metrics are indeed good maintenance predictors, even when their correlation with size is neutralized.

RQ4: Does the performance of code metrics vary based on the evaluation context (maintenance indicators and method size), and why?

Contribution 4: We show that evaluation context is a significant factor for code metric performance. Some code metrics perform well for small methods, but not for large methods. We show that these metrics lose variability when applied to large methods. Once they reach a threshold they lose predictive power. Other metrics, however, can increase monotonically (e.g., McCabe) and do not suffer from a lack of variability in the measurements. So their performance is not negatively impacted by code size. Also, a metric's performance varies greatly based on the maintenance indicator used. A metric can be good for estimating the number of revisions, but not good for estimating the size and the nature of code modifications.

These observations are novel because they clearly show that code metrics are useful as maintenance predictors, while explaining the apparent contradictions from prior studies. With context-based evaluations, we provide new ways to examine the effectiveness of existing and future code metrics, and how they should be used to build more accurate software maintenance models. To aid reproducibility, we provide a public replication package[1] consisting of a data set of ∼730K Java methods with their complete histories and the values for all computed metrics and maintenance indicators over time.

[1] https://github.com/shaifulcse/codemetrics-with-context-replication

1.1 Paper Organization

Section 2 discusses the potential root causes of the previous contradictory claims about code metrics, which helped design the methodology of this paper. Section 3 discusses the methodology. In Section 4, we reproduce the previous contradictory claims about code metrics. We also discuss the inaccuracy of the traditional size normalization approach. In Section 5, we show the true maintenance impact of code metrics by a combination of bivariate and multivariate regression analysis. We also demonstrate why different evaluation contexts should be considered before drawing any conclusion about code metrics. The significance of our findings and threats to validity are presented in Section 6. Section 7 concludes this paper with some potential future studies.

2 Related Work & Motivation

First we discuss the McCabe cyclomatic complexity, a metric for measuring the number of linearly independent paths through a component (McCabe 1976). This metric was proposed in 1976, and has been widely studied and adopted (Ebert et al. 2016; Pantiuchina et al. 2018). We can divide all the McCabe-related studies into two groups: studies that support its validity (e.g., Curtis et al. 1979; Landman et al. 2014; Tiwari and Kumar 2014; Zhou et al. 2010; Alfadel et al. 2017), and studies that do not (e.g., Weyuker 1988; Shepperd 1988; Gil and Lalouche 2017; Scalabrino et al. 2017). McCabe is not the only metric that has been debated. With strong empirical evidence, other widely adopted metrics, such as C&K (Chidamber and Kemerer 1994), readability (Buse and Weimer 2010) have been criticized (Gil and Lalouche 2017; Scalabrino et al. 2017). We identify the following factors that may influence the outcome of a code metric study, and thus support contradictory conclusions.

User studies are subjective: Much metrics research relies on user studies to understand the impact of metrics on maintenance indicators (Abid et al. 2019; Hofmeister et al. 2017; Scalabrino et al. 2016; Buse and Weimer 2010; Scalabrino et al. 2017; Antinyan et al. 2017; Bauer et al. 2019; Kafura and Reddy 1987; Curtis et al. 1979; Darcy et al. 2005). One peril related to user studies is that the outcome often depends on human subjects, and can be inconclusive or even contradictory (Brittain 1982). Also, in the context of code quality, user perception does not necessarily match with the true quality of software (Pantiuchina et al. 2018). Unsurprisingly, we observe contradictory results for similar maintenance indicators while code metrics were evaluated: both that code metrics are useful (Johnson et al. 2019), and that they are not useful (Scalabrino et al. 2017). *In this paper, we therefore focus on objective change measurements as maintenance indicators that we collect from real-world software projects.*

Size as a confounding factor: The most frequent criticism invalidating code metrics is that they are highly correlated with size (Ebert et al. 2016; Landman et al. 2014; Shepperd 1988; Yu and Mishra 2013; Kafura and Reddy 1987). Therefore, none of the metrics offer any new maintenance information when normalized against size (Gil and Lalouche 2017; El Emam et al. 2001).

To claim validity of a metric, we need to show that the metric has predictive power even after its dependency to size is neutralized. Despite this well and long established fact, several studies have ignored it (e.g., Johnson et al. 2019; Tiwari and Kumar 2014; Subandri and Sarno 2017; Romano and Pinzger 2011). In a recent study by Johnson et al. (2019) developers took less time to read code snippets that followed certain rules (e.g., reduced nesting

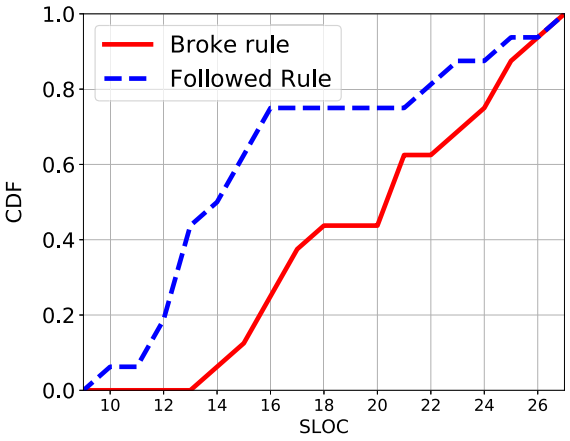
level) than those that did not follow such rules. The publicly available dataset enabled us to analyze the size distribution of the snippets. Figure 1a (cumulative distribution function of source lines of code) shows that the snippets that broke the rule were much larger than those that followed the rules. More lines of code would naturally take more time to read, so perhaps size made the difference in reading time, and not the reduced nesting level.

Some studies, however, have attempted to neutralize size while evaluating code metrics. For example, Spadini et al. (2018) evaluated the maintenance impact of test smells in three different size categories: small (SLOC $<$ 30), average (30 $<$ SLOC $<$ 60), and large (SLOC $>$ 60). Although this approach should reduce the confounding impact of size to some extent, analyzing all methods with SLOC $>$ 60 (for example) in one group can not eliminate the problem completely. A more common approach is to calculate metric density per lines of code (Suh and Neamtiu 2010; Shepperd 1988; Robert et al. 2012; Gil and Lalouche 2017)—i.e., $100 \times McCabe/Size$. Unfortunately, Gil and Lalouche (2017) argued that this approach is inaccurate and questions some of the previous claims of validity for different code metrics. *We argue that a metric is a valid maintenance indicator only when it correlates with maintenance after the confounding factor of size is neutralized, and traditional size normalization approach does not help in making such observation. We need a new approach to evaluate code metrics' effectiveness.*

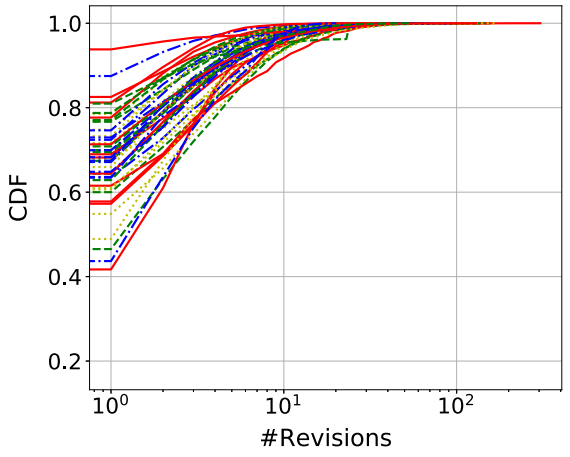
Aggregated analysis: Some studies were based on aggregated analyses (Gil and Lalouche 2017; Spadini et al. 2018; Pascarella et al. 2020). That is, they combined all metrics and maintenance indicators from all the studied projects. This is problematic for several reasons. Different external factors—e.g., code review policy (Wang et al. 2019), developer commit patterns (Herzig and Zeller 2013) and expertise (Matter et al. 2009)—cause code to evolve in projects differently. Figure 1b shows the distributions of revisions for all the methods in each of our 47 projects (described later); each line corresponds to one project. Evidently, these projects do not exhibit similar revision behavior. Combining them together may lead us to inaccurate conclusions. Figure 1c shows that the difference in distributions reduces after applying recommended log-normal transformation (Gil and Lalouche 2016), but the differences do not completely disappear. Even for code metrics, the distribution in their measurements greatly vary based on a project's domain, programming language, and life span (Zhang et al. 2013). Also, some projects are much bigger than others. This means that in aggregated analyses, results can be unduly influenced by few big projects.

These problems of aggregated analysis can be avoided by analyzing each project individually (Shin et al. 2011; Zhou et al. 2010; Kafura and Reddy 1987; Romano and Pinzger 2011). Individual project analysis, however, has been criticized for selection and publication bias (Gil and Lalouche 2017; Radjenović et al. 2013). As we also show in this paper, there are always outlier projects that exhibit unique behavior, which might seem normal if too few projects are studied. *The argument is thus to analyze each project separately, while studying a reasonably large number of projects with a systematic unbiased selection process.*

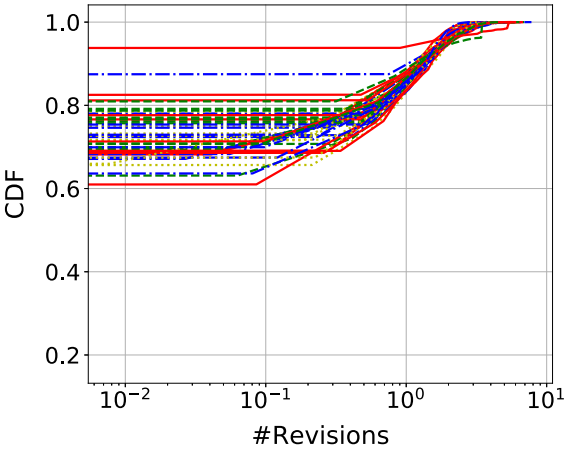
Granularity: Software maintenance studies have been conducted at different granularities that can influence observations (Landman et al. 2014); these include the system level (Suh and Neamtiu 2010; Kafura and Reddy 1987), class/file level (Palomba et al. 2017), snippet level (Hofmeister et al. 2017; Bauer et al. 2019), and even `git diff` level (Hindle et al. 2008). Understanding maintenance at the method level granularity from real software evolution data is difficult (Ying et al. 2004; Higo et al. 2020; Grund



(a) Confound factor



(b) Distribution of method revisions



(c) Distribution after log-normal transformation

Fig. 1 Figure (**a**) shows that size was not normalized in the study by Johnson et al. (2019). Code snippets that broke the rules are much larger than the code snippets that followed the rules, leading to inaccurate comparisons. Figure (**b**) shows that aggregated analysis is inaccurate because different projects exhibit different revision behaviors. Each line represents the revision distribution for a given project, and these lines are very different from each other. Figure (**c**) shows that these different revision patterns are not neutralized, even after applying a log-normal transformation, as suggested by Gil and Lalouche (2016)

et al. 2021b); it is harder to reliably generate method level histories than file level histories. Despite the difficulty, method level is the most desirable granularity (Pascarella et al. 2020; Menzies et al. 2007), because class/file level granularity is often too coarse-grained for practical use (Shihab et al. 2012; Giger et al. 2012; Pascarella et al. 2020). *We also argue that if we can estimate maintenance at method level granularity, we can extend this understanding to coarser levels of granularity—a class is generally a collection of methods.*

Maintenance Indicators: We consider software maintenance as a *construct* (Ralph and Tempero 2018), which is difficult to measure, but easier to estimate it through some reflective indicators. Different studies have focused on different indicators: human effort to read and understand code (Johnson et al. 2019; Sridhara et al. 2010), localizing bugs (Zimmermann et al. 2007; Islam and Zibran 2020; Pascarella et al. 2020; Menzies et al. 2007; Tosun et al. 2010), change proneness (Gil and Lalouche 2017), or developer activities (Shin et al. 2011). In this paper, we focus on four change proneness indicators of Java methods (justified later). We show that metric performance to understand maintenance can vary significantly based on the indicators used.

3 Methodology

This section describes our process for: i) selecting projects, ii) choosing code metrics and maintenance indicators, iii) collecting method-level history for analysis, iv) age normalization for methods with different ages, and v) selecting statistical approaches for analysis.

3.1 Project Selection

To reduce inaccuracies that may stem from aggregated analysis, we opted to analyze individual projects. To neutralize selection bias, we took the union of all GitHub Java projects used in four different software evolution studies (Gil and Lalouche 2017; Grund et al. 2021b; Spadini et al. 2018; Palomba et al. 2017), totalling 47 projects—mixing projects from different programming languages can significantly impact the outcome of code metrics studies (Zhang et al. 2013). As we show later, this project set is able to highlight code metric behaviors that are generic (true for most projects) and which behaviors are rare. Table 1 describes the dataset. The table suggests that only a small number of methods (e.g., 95^{th} percentile of revisions) undergoes a large number of revisions. That means we can significantly reduce the search space for maintenance optimization by identifying the top 5% high-churn methods. This paper investigates, if code metrics are indeed helpful for such identification.

We also note that the number of methods is significantly different across the projects. If we were to adopt aggregated analysis, results for small projects would be unnoticeable. The set of projects is clearly diverse. For example, even for the subset that was used in Gil and Lalouche (2017), the number of developers ranges between 16 and 197, and development duration varies from one to 13 years.

3.2 Code Metric Selection

In contrast to method-level granularity, many of the popular code metrics, such as C&K and depth of inheritance, work only at class or higher level granularities. Also, the objective of this paper is not to show which code metric is the best for estimating maintenance

Table 1 Description of the dataset used in this paper

Repository	# methods	# rev (avg)	# rev (med)	# rev (max)	# rev (95^{th} percentile)
hadoop	70,081	1.8	1.0	67.0	6.0
elasticsearch	62,190	3.5	2.0	121.0	12.0
flink	38,081	1.8	1.0	93.0	7.0
lucene-solr	37,133	1.5	1.0	145.0	6.0
docx4j	36,514	2.2	2.0	49.0	4.0
hbase	36,274	3.2	2.0	109.0	11.0
intellij-community	35,950	3.6	2.0	120.0	13.0
weka	35,639	1.7	1.0	86.0	5.0
hazelcast	35,265	2.7	1.0	109.0	10.0
spring-framework	26,634	2.4	1.0	60.0	8.0
hibernate-orm	24,800	2.5	2.0	70.0	7.0
eclipseJdt	22,124	3.0	1.0	133.0	12.0
guava	20,757	1.1	0.0	45.0	4.0
sonarqube	20,627	3.0	2.0	305.0	9.0
jclouds	20,358	1.6	1.0	59.0	5.0
wildfly	19,665	2.1	1.0	83.0	8.0
netty	16,908	2.0	1.0	75.0	9.0
cassandra	15,953	1.5	0.0	62.0	6.0
argouml	12,755	3.3	2.0	80.0	10.0
jetty	10,645	2.2	1.0	93.0	8.0
voldemort	10,601	1.7	0.0	65.0	8.0
spring-boot	10,374	2.6	2.0	59.0	9.0
wicket	10,058	4.9	3.0	63.0	14.0
ant	9,781	2.0	1.0	73.0	8.0
jgit	9,548	1.4	1.0	44.0	6.0
mongo-java-driver	9,467	3.3	2.0	57.0	13.0
pmd	8,992	3.2	2.0	91.0	10.0
xerces2-j	8,153	1.3	0.0	65.0	5.0
RxJava	8,145	3.7	3.0	22.0	10.0
openmrs-core	6,066	2.1	1.0	51.0	7.0
javaparser	5,862	3.2	1.0	84.0	14.0
hibernate-search	5,345	3.2	2.0	61.0	11.0
titan	4,590	2.2	1.0	42.0	8.0
checkstyle	3,340	3.8	2.0	72.0	13.0
commons-lang	2,948	3.6	3.0	34.0	9.0
lombok	2,684	2.0	1.0	43.0	7.0
atmosphere	2,659	2.4	0.0	87.0	11.0
jna	2,636	2.3	1.0	38.0	8.0
Essentials	2,390	3.0	1.0	46.0	14.0
junit5	2,085	2.7	1.0	59.0	11.0
hector	1,958	1.7	1.0	43.0	8.0

Table 1 (continued)

Repository	# methods	# rev (avg)	# rev (med)	# rev (max)	# rev (95^{th} percentile)
okhttp	1,953	4.7	3.0	54.0	16.0
mockito	1,498	4.1	3.0	62.0	13.0
cucumber-jvm	1,146	2.6	1.0	36.0	9.0
commons-io	1,145	3.1	3.0	24.0	8.0
vraptor4	926	1.6	1.0	24.0	6.0
junit4	874	3.1	2.0	70.0	11.0

In total, 733,577 Java methods were collected from 47 GitHub Java projects. For each project, we show the average (# rev (avg)), the median (# rev (med)), the maximum (# rev (max)), and the 95^{th} percentile of revisions, considering all the methods a project contains. The small average, and median number of revisions compared to the large 95^{th} percentile and maximum revisions suggest that most maintenance activities occur in small areas of code

effort, because there are many of them (Scalabrino et al. 2017). Instead, we focus on code metric validity: are they useful at all, and if so, are the underlying evaluation contexts important? Therefore, we focus only on seven widely adopted and widely studied metrics that are applicable at method-level granularity. We show that these seven metrics were sufficient to reproduce previous contradictory claims and to examine how code metrics should be reliably evaluated.

McCabe: The McCabe algorithm for measuring cyclomatic complexity is simply: $1 + \#predicates$ (McCabe 1976). There are, however, two forms: one counts logical && and ||, and the other ignores them. We only consider the latter form because considering them does not make any meaningful difference in McCabe's validity as a code metric (Landman et al. 2014).

McClure: A criticism of McCabe is that it does not consider the number of control variables in a predicate. If the outcome of a predicate depends on multiple control variables, it should be considered more complex than the one with a single control variable (Kafura and Reddy 1987). McClure differs in this regard (McClure 1978): it measures the sum of the total number of comparisons (thus includes && and ||) and the number of control variables in a component.

Nested Block Depth: Neither McCabe, nor McClure, considers nesting depth. To both of these metrics, two methods each with two loops (for example) are equally complex, even if one of them has nested loops and the other does not. Measuring Maximum Nested Block Depth (referred to as NBD) is a common solution (Johnson et al. 2019; Antinyan et al. 2015).

Proxy Indentation: Hindle et al. (2008) argued that a metric like McCabe is hard to calculate because one needs a language-specific parser. They found that it is similarly useful to use the level of indentation in a code component. Counting the raw number of leading spaces in each line is equally good as counting the number of logical spaces. Instead of calculating the max, sum, mean, or median, the authors found that standard deviation of those counts (referred to as IndentSTD) works as the best proxy for McCabe-like complexity.

FanOut: The aforementioned metrics, to some extent, measure similar complexities—mainly the number of conditional branches. Therefore, we add FanOut (total number

of method calls made by a given method) to our list. This metric provides an indication of how a particular method is dependent on other methods (coupling). Mo et al. observed that highly coupled systems are usually less maintainable (Mo et al. 2016). We also wanted to use FanIn or unique FanOut, but these two require a symbol solver that preprocesses a complete repository for each change commit a method has. It would be extremely time demanding for the ∼730K methods that we consider.

Readability: Unlike the aforementioned five metrics, readability is a composite metric that combines different code metrics to produce a single indirect maintenance index. For this we adopt the widely used Readability metric by Buse and Weimer (2010) which ranges from 0 (least readable) to 1 (completely readable code).

Maintainability Index: As another composite metric, we consider the popular maintainability index metric, which is calculated as:

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

This is the evolved form of the original equation proposed by Oman and Hagemeister (Oman and Hagemeister 1992). Different evolved forms have been adopted by popular tools such as Verifysoft technology[2] and Visual Studio.[3]

3.3 Maintenance Indicator Selection

Modeling maintenance effort is a difficult problem, because there are many different effort indicators that should be considered for building a comprehensive effort prediction model. A subset of these indicators include human effort to read and understand code (Börstler and Paech 2016; Buse and Weimer 2010; Johnson et al. 2019; Scalabrino et al. 2017), difficulty to modify a code (Mo et al. 2016), bug proneness (Giger et al. 2012; Rahman and Roy 2017), and change proneness (Catolino et al. 2018; Gil and Lalouche 2017; Romano and Pinzger 2011; Shihab et al. 2012). The objective of this paper is not to build an effort prediction model, but to answer if code metrics are at all useful for understanding maintenance effort, and how to evaluate these metrics reliably. In that vein, we focus on change proneness, as it is measurable without conducting user studies, reducing threats related to such studies. Also, the community unanimously agrees about the utility of change proneness as one of the most applicable maintenance effort indicators (Catolino et al. 2018; Gil and Lalouche 2017; Romano and Pinzger 2011; Shihab et al. 2012; Ying et al. 2004; Palomba et al. 2017; Mocku and Votta 2000; Khomh et al. 2012). While we considered bug proneness, we discarded this indicator to reduce threats to construct validity. From our dataset, bug proneness can be measured by capturing keywords from commit messages, such as *error*, *bug*, and *fixes* (Mocku and Votta 2000; Ray et al. 2016). Unfortunately, this approach has been criticized for low precision/recall (Gil and Lalouche 2017; Spadini et al. 2018), which is further complicated due to tangled changes. Developers often commit unrelated changes, which incorrectly labels bug-free code as *buggy* (Herzig and Zeller 2013). Additionally, change proneness is often highly correlated to bug proneness (Moser et al. 2008; Bell et al. 2011; Bavota et al. 2015; Rahman and Roy 2017; Pascarella et al. 2020). Therefore, if a code metric is a good predictor of change proneness, it is likely to be a good predictor of bug proneness as well. Ultimately we selected the following four change proneness indicators.

[2] https://verifysoft.com/en_maintainability.html: last accessed: December-28-2021

[3] https://docs.microsoft.com/en-us/visualstudio/code-quality/code-metrics-maintainability-index-range-and-meaning?view=vs-2022: last accessed: December-28-2021

#Revisions: Number of revisions of a component is considered as an indication of maintenance effort by many (Antinyan et al. 2014; Monden et al. 2002; Shin et al. 2011; Antinyan et al. 2015). The consensus is that a well designed less complex component should not need many revisions.

Diff size: Number of revisions does not disclose how large a change is. If two components are revised the same number of times, their maintenance effort is not necessarily the same. Also, the number of revisions can be influenced by developers' commit habit or culture (Wang et al. 2019). Therefore, some consider `git diff` size a more accurate maintenance indicator (Scholtes et al. 2016; Shin et al. 2011).

Additions only: Adding new lines is perhaps more difficult than deleting lines. This threat can be reduced by considering only the number of new lines added (Shin et al. 2011).

Edit Distance: Lines of changed code, as a metric, is affected by noise such as coding style; it does not distinguish modifications between large and small lines. Also, a simple automatic rename method refactoring may modify a large number of lines. Therefore, Levenshtein edit distance (Levenshtein 1966) is considered as a better maintenance indicator than number of lines (added and/or deleted) (Ståhl et al. 2019; Scalabrino et al. 2017; Scholtes et al. 2016). Levenshtein edit distance measures the number of characters $added + deleted + updated$ for converting one source code version into another.

3.4 Data Collection and Representation

We require a method's complete change history: how many times the method was changed, when the changes happened, and what was changed? There are only few tools that support history tracing at method level granularity: Historage (Hata et al. 2011), FinerGit (Higo et al. 2020), and CodeShovel (Grund et al. 2021a, b). Historage and FinerGit work similar to `Git`'s file tracking mechanism by converting each Java method to a file. However, we find that this approach does not scale well to larger projects. In contrast, CodeShovel tracks a method (even if the method's signature is changed) using string similarity and without any project preprocessing. Unlike the other tools, CodeShovel's accuracy was evaluated on both open source and closed source industry projects, with 99% precision and 90% recall.

After collecting the complete history of 733,577 Java methods from 47 selected projects, we collected the evolution of their code metrics (e.g., SLOC, McCabe), and change metrics (e.g., edit distance). To the best of our knowledge, there is no existing tool that provides measurements in this form, so we have implemented our own tool. We verified its correctness by randomly selecting and validating 200 Java methods. In addition, the accuracy of the tool was tested by an independent code metric researcher. A method, across its evolution history, can have different values for the same code metric (e.g., initially the McCabe was 5, but then it changed to 3, and then to 5 again). For a single method, we thus summed all the maintenance indicator values (e.g., sum of all edit distances) that a method had for each unique code metric value. For a given method, for example, if edit distance 10, 20, and 30 correspond to McCabe values 5, 3, and 5 respectively, McCabe value 5 is blamed for edit distance 40 (10+30), and McCabe 3 is blamed for edit distance 20. This is how we mapped code metrics value with different maintenance indicators to study the relationships between them.

Why did we use sum instead of other descriptive statistics, such as mean? Let us consider the history for two real methods from the Checkstyle project present in our dataset. The method *visitToken* (in MagicNumberCheck.java, with CodeShovel method ID: visitToken__ast-DetailAST) was revised 18 times with edit distances: 17, 425, 106, 437, 133, 96,

41, 86, 2, 48, 2, 90, 29, 272, 3, 5, 126, and 65. The method *hasJavadocInlineTags* (in SingleLineJavadocCheck.java, with CodeShovel method ID: hasJavadocInlineTags__javadocRootDetailNode) was revised three times, and the edit distances are: 4, 422, and 2. A natural question is to ask which method is most change prone. While it is obvious that the *visitToken* method is more change-prone than the method *hasJavadocInlineTags*, different statistics can provide different interpretations. Specifically, the sum of edit distances suggests that *visitToken* is more change-prone (the sum edit distance for method *visitToken* is 1983, while for *hasJavadocInlineTags* is 428). But the mean edit distance suggests the opposite: for method *visitToken* the mean edit distance is 110.16, and for method *hasJavadocInlineTags* the mean edit distance is 142.66. This is contrary to what one would reasonably expect, looking at the raw data. For this reason, we believe using the metric's mean is more likely to be misleading.

3.5 Age Normalization

It is inaccurate to compare the change history of two differently-aged methods. An older method is more likely to have more revisions than a newer method (Lehman et al. 1997; Yu and Mishra 2013). For the rest of the analysis, we consider methods that are at least two years old: reasonably enough time to undergo their initial changes. However, this approach does not completely neutralize the time effect; for instance, we should not compare a two year with a year ten method. We neutralize this by considering changes that happen only within the first two years of these filtered samples. This is like time traveling to each of the methods change history when they were two years old. But why two years? Figure 2 (cumulative distribution functions with day) shows that more than 80% of our methods (total 602,550 methods) are older than two years (Age). Among all the revisions in whole dataset (All changes in the graph), ∼60% of them happened within the first two years. If we consider the interval time of subsequent revisions, around 86% of changes happened within the first two years. If we increase the age threshold value, we lose more methods. If we decrease it, we lose more change history, so it is a trade-off. Note that if we set the threshold to one or three years, the major conclusions of this paper remain the same.

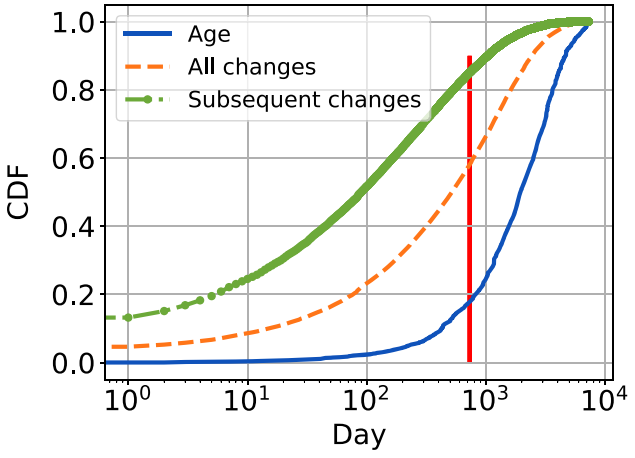


Fig. 2 Two years is a good threshold for age normalization. We only lose ∼20% of the methods, and yet retain ∼60% of the revisions that happened within our whole dataset

3.6 Correlation and Statistical Significance

To apply Pearson's formula for calculating correlation coefficients between code metrics and maintenance indicators we need to establish that each metric, for each change proneness indicator, for each project is normally distributed. After applying the Anderson-Darling normality test (Thode 2002) for some of the randomly selected projects, we found that they are not normally distributed. Therefore, we opted to use Kendall's τ correlation coefficient. Unlike Pearson's correlation coefficient, Kendall's τ does not assume any distribution of the data (non-parametric), and is less affected by outliers, which the community has chosen to use for these kinds of analysis (Inozemtseva and Holmes 2014; Gil and Lalouche 2017; Chowdhury et al. 2019). Unless otherwise stated, all results in this paper are statistically significant (p-value < 0.05). When necessary, we also use the Wilcoxon rank-sum test to test if the performance distributions of the code metrics are statistically different, and if so, we report how large the differences are (Cliff's Delta effect size). Similar to Kendall's τ, these two tests are non-parametric and do not assume any distribution of the data (Sheskin 2020; Romano et al. 2006).

As we consider each project separately, we present the results as distributions. Therefore, we use the Cumulative Distribution Function (CDF) for the visual representation of our results. We considered using XY-plots, but CDF better conveys our findings. As CDF is a monotonic function, comparing multiple lines (because of multiple code metrics) is easier than XY-plots' zigzag-patterns.

4 Results: Looking into the Past

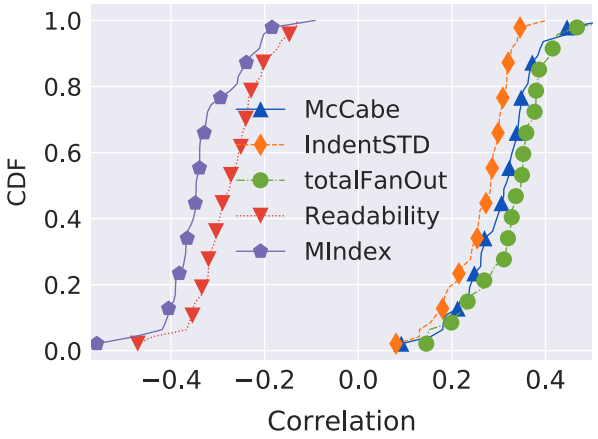
In this section, we reproduce previous claims about the relationship between code metrics and software maintenance (RQ1). We show that the debate about source code metric effectiveness stems from improperly considering, or normalizing for, size as a confounding factor. We show that the most commonly used normalization approach fails to neutralize the size effect in practice. We then explain why size normalization is difficult and remains an open research problem (RQ2).

4.1 (RQ1) Metrics are (not) Useful

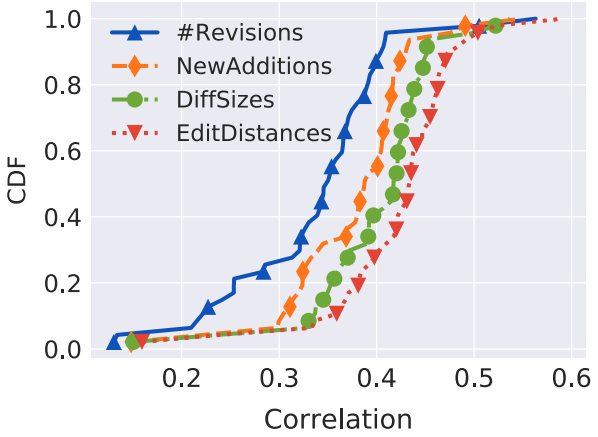
Figure 3a shows the cumulative distribution functions (CDF) of the correlation coefficients between the selected code metrics and number of revisions (each line represents a particular metric and shows the distribution across all the 47 projects). Results are similar for McCabe, McClure, and NBD, so we show only McCabe to maintain graph readability.

Evidently, all the seven code metrics are correlated with maintenance measures, which means that they are potentially good maintenance predictors. This approach aligns with a group of prior studies (e.g., Johnson et al. 2019; Tiwari and Kumar 2014; Subandri and Sarno 2017) that did not attempt to control for size as a confounding factor.

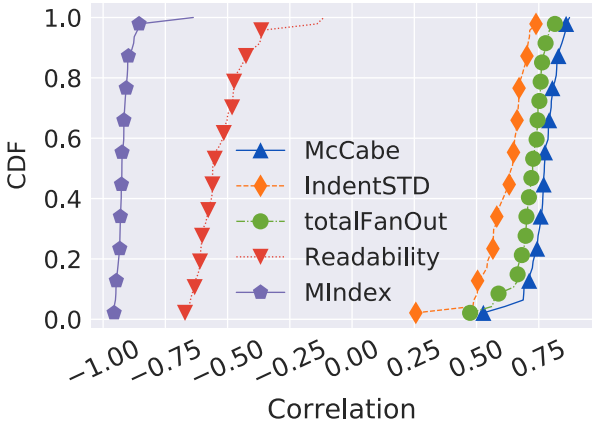
Correlating a metric with maintenance alone does not make a metric valid or useful (Gil and Lalouche 2017; Shepperd 1988). The arguments supporting this are: (1) size is a great predictor of maintenance, and (2) many code metrics are highly correlated with size. So a metric's correlation with maintenance could simply be due to its correlation with size. Figure 3b shows the correlation between SLOC and the four maintenance indicators for all 47 projects (supporting argument 1). Figure 3c shows that all the metrics are correlated with size (supporting argument 2). We observe that the direction and strength of the correlation



(a) Code metrics and #Revisions.



(b) SLOC and change indicators.



(c) Code metrics and SLOC.

Fig. 3 Figure (**a**) shows that all the code metrics are significantly correlated with number of revisions in each project (observations are similar for other maintenance indicators). Figure (**b**) shows that SLOC is positively correlated with all the maintenance indicators. Figure (**c**) shows that code metrics are correlated with SLOC. For graph readability, the number of marks in each line is fewer than the actual number of data points

between a code metric and maintenance is similar to the metric's correlation with size. For example, Readability is negatively correlated with both size and maintenance (larger size means less readable and thus less maintainable). For IndentSTD the correlation is somewhat lower (compared to McCabe and FanOut) with size and thus lower with maintenance. These observations align well with the criticism that when the influence of size is considered, we do not have any empirical evidence to support the usefulness of code metrics (Shepperd 1988). From this we conclude that without size normalization we do not know the true effectiveness of code metrics.

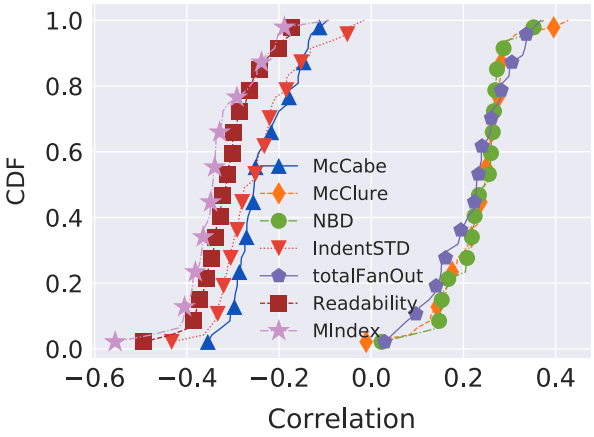
The most common approach for size normalization takes the density of a metric and divides its measure in a component by the size of the component (Suh and Neamtiu 2010; Shepperd 1988; Robert et al. 2012; Gil and Lalouche 2017). For example, $100 \times McCabe/Size$ gives the McCabe value per 100 lines of code, so we should have a normalized McCabe measure completely independent of size. The hypothesis is that, if we still see correlation between a metric and maintenance, we can argue for the validity of the metric. Figure 4a shows the distributions (for 47 projects) of correlation coefficients for all the normalized code metrics with the number of revisions (results are similar for other maintenance indicators). Evidently, all the metrics are still correlated with maintenance. This supports the assumption of code metrics validity after the size influence is neutralized.

To our surprise, we find that this commonly practiced size normalization approach is inaccurate. Figure 4b shows the correlation distributions between the normalized metrics and size for all 47 projects. Although we were expecting the correlation to be close to zero, this is not the case. We also note that the direction of the correlation between size and a metric still dictates the direction of the correlation between maintenance and that metric. For example, McCabe is negatively correlated with both size and number of revisions. We later found that the same observation was made by Gil and Lalouche (2017), although their granularity level was different (file-level instead of method-level). They concluded that *size is the only valid code metric* because maintenance impact of other code metrics can directly be explained by their correlations to size. Despite our similar observation, we see hope if we carefully examine Fig. 4a and b. For example, the correlations with revisions are similar for FanOut, NBD, and McClure, but not as similar to size. For Gil et al., size explains everything (file-level), but for us it does not (method-level).

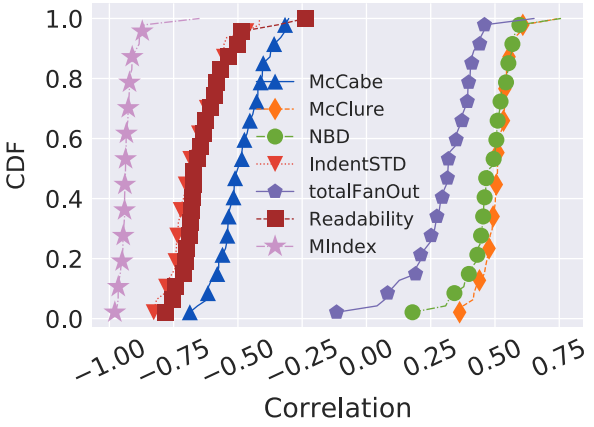
This difference of observations can be explained by the findings of Landman et al. (2014), who have studied the correlation between McCabe and size at different granularity levels. In their study, the strong correlation between McCabe and size is true only for large code units, but dwindles significantly at the method level granularity. The authors, however, did not examine McCabe's impact on maintenance. In this paper, with the help of bivariate and multivariate regression analysis, we show that code metrics are indeed good maintenance predictors, even when their relations with size are neutralized. But before examining this, we first explain why the widely adopted and believed size normalization fails (RQ2).

4.2 (RQ2) Size Normalization is Sensitive to Size

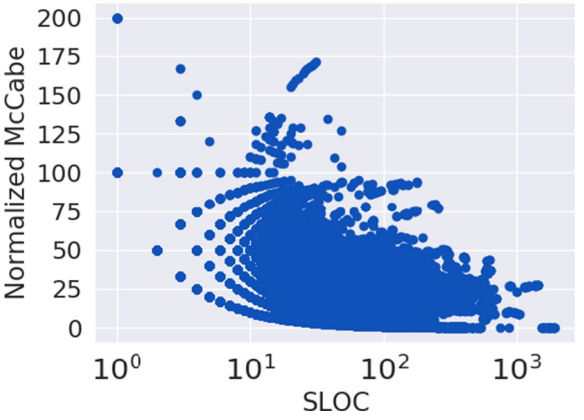
To see why size normalization does not work, we take a deeper look into McCabe complexity ($1+\#predicates$). Why is normalized McCabe negatively correlated with size (and thus with maintenance)? Of course, a negative correlation here indicates that the lower the size the higher the normalized value (i.e., density per 100 lines of code). Interestingly, we find that the '1' in the McCabe formula ($1+\#predicates$) is a major issue. Consider a simple 3 line Java method, which just returns the sum of two numbers. The McCabe is already 1, and the normalized value is 0.33 (1/3). The effect of '1', however, diminishes as method size



(a) Normalized metrics and #Revisions.



(b) Normalized metrics and SLOC.



(c) Normalized McCabe against SLOC.

Fig. 4 Figure (**a**) shows that all the normalized code metrics are still significantly correlated with number of revisions in each project. However, Figure (**b**) suggests that even the normalized code metrics are correlated with SLOC. Figure (**c**) demonstrates why the normalized McCabe is negatively correlated with SLOC (and thus with change)

grows. Figure 4c depicts normalized McCabe score against size for all the 602,550 methods. Clearly, the normalized McCabe score decreases with the increase in size because of the high density in small methods. We find that if we eliminate the '1' from the formula, the size influence in normalized McCabe is reduced, but does not go away completely. The graph shows that the normalized score for small methods can even exceed 100. As an example, consider the method from the *Elasticsearch* project, shown in Fig. 5. This method is written as a one line method (size = 1), and because of the conditional operator, the plain McCabe is 2 (1+#predicates). The normalized McCabe is thus 200 ($100\times(2/1)$). Note that this data point with 200 normalized McCabe score represents 56 methods, not just one.

As we show later, most methods in our dataset are small (SLOC $\leq$21), and therefore the overall correlation is significantly impacted by the high McCabe density of this large number of small methods. The problem is, Fig. 4b shows that different code metrics suffer differently from this size normalization approach. Size normalization thus remains an open research problem: we need to develop an approach that not only eliminates the influence of size, but also does not normalize in a way that hides the effectiveness of code metrics.

Summary: Previous approaches that supports the usefulness of code metrics either did not consider the size influence or normalized it inaccurately. A new way is required to evaluate the true effectiveness of code metrics by completely eliminating the effect of size.

5 Results: Evaluating Metrics with Regression and Contexts

The problems of performing a size-decoupled metric evaluation with a traditional size normalization approach led us to a study by Chen et al. (2020). The authors investigated why different mutation testing studies claimed differently (Inozemtseva and Holmes 2014; Gopinath et al. 2014; Just et al. 2014; Papadakis et al. 2018) about the relationship between test suite size, test adequacy criteria (e.g., coverage), and test effectiveness (fault detection). Although the context of their study is different than ours, the outcome is similar: two different studies control for test suite size, while evaluating the relationship between test adequacy criteria and test effectiveness, producing two different conclusions. For highly correlated variables (code metrics and size in our case), the authors suggested that regression analysis can be useful. Encouraged by their hypothesis, we designed our approach as follows.

1. With a bivariate regression analysis between size and a maintenance indicator (e.g., revisions), we calculate the goodness of fit score of the regression model.
2. In the same model, we then add one of the code metrics (e.g., McCabe) as the second independent variable and asked whether this multivariate regression model (size+McCabe $\sim$ #revisions) improves the goodness of fit score with statistical significance (p-value < 0.05 for the coefficient of McCabe)?
3. We take the difference between the two fitness scores and convert it to a percent improvement to show the distribution across the 47 different projects.

```
static int normalizeIndex(final byte[] ar,
final int index) { return index >= 0 ?
index : index + array.length; }
```

Fig. 5 A sample method from Elasticsearch

4. We repeat steps 1 to 3 for all the maintenance indicators and code metrics.

We argue that this approach shows the true maintenance impact of code metrics because it correctly eliminates the size influence.

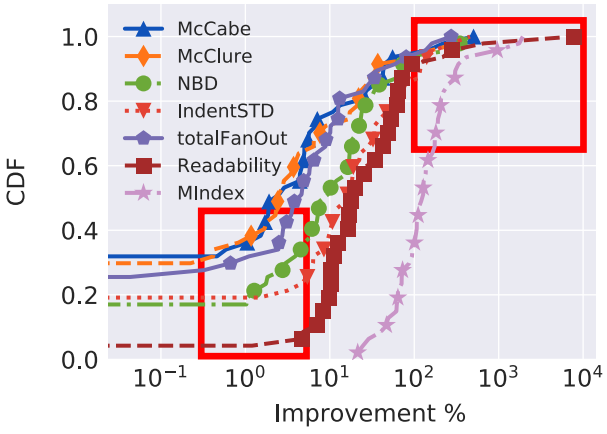
5.1 (RQ3) Regression Analysis for Code Metric Evaluation

Figure 6 shows the CDFs of percent of improvement in fitness scores with the multivariate regression models for three of the maintenance indicators. We excluded the result for the `diff size` indicator, because the observation is the same—all the metrics improve the fitness scores for all of the indicators. Figure 6a, for example, shows that the Readability metric improves the fitness accuracy by at least 10% for more than 80% of the projects (and at least 100% for 20% of the projects). Maintainability Index performs even better. This clearly refutes the claim that code metrics are not useful after size influence is neutralized (Gil and Lalouche 2017; El Emam et al. 2001). Except for the Maintainability Index, the performance of other metrics, however, are not the same across all maintenance indicators. Readability is the second best metric for estimating the number of revisions. For estimating change size (e.g., edit distance), however, NBD and IndentSTD outperform Readability. Also, results in Fig. 6 are dominated by methods that are small in size, because most of our methods are small. We need to evaluate if code metrics perform differently when evaluated for large methods only, and if so, what factors influence their performance. These are the questions we investigate in RQ4.

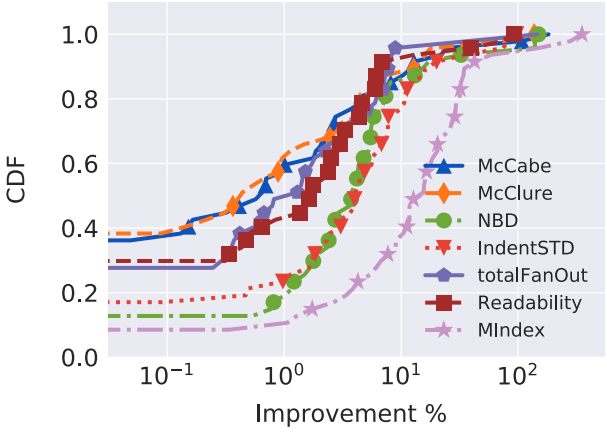
5.2 (RQ4) Evaluating Code Metrics with Contexts

To investigate whether the performance of a code metric depends on the method size, we need to first define a threshold for separating large methods from small ones. Instead of defining such a threshold from intuition or from expert opinion, we followed the 6-step systematic approach proposed by Alves et al. (2010). The main objective of the approach is to find critical values for identifying low risk (small size), medium risk (medium size), high risk (large size), and very high risk (very large size) code components in terms of maintenance from a given set of projects. These critical values are robust, i.e., they are not impacted by outlier projects or methods. We refer to Alves et al. (2010) for more detail. The first 5 steps of Alves et al.'s approach deliver 3 critical values that are derived from Fig. 7. The first critical value shows that SLOC is ≤ 21 for 70% of the Y-axis. The second (32) and the third (58) critical values represent 80% and 90% of the Y-axis respectively. In step 6, we can now find the range to define a method's size category: small (SLOC ≤ 21), medium ($21 <$ SLOC ≤ 32), large ($32 <$ SLOC ≤ 58), and very large (SLOC > 58). Clearly, the results in Fig. 6 are dominated by methods with SLOC ≤ 32 (80% of the Y-axis), limiting our understanding of metric performance for large methods. Here, we evaluate metrics for large and very large methods only (SLOC > 32)—we refer to both groups as "large" for simplicity.

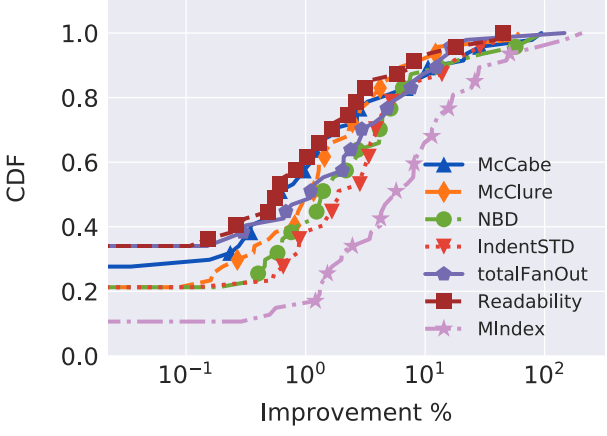
Figure 8 shows that code metrics can be used to understand maintenance effort for large methods. However, their ranks in performance are not the same when compared with methods from all sizes (Fig. 6, dominated by small SLOCs). We make the following observations while comparing Figs. 6 and 8: 1) Code metrics are useful for estimating maintenance effort for both large and small methods. 2) FanOut's performance is pretty consistent across different method sizes. This indicates that developers should be careful about coupling (i.e., dependency on other methods) for all methods. 3) Readability effectiveness drops



(a) # revisions



(b) New additions



(c) Edit distance

Fig. 6 Cumulative distribution functions of percent of improvement in goodness of fit scores. All seven metrics improve the prediction accuracy of the regression models when they are added with size. The two boxed areas in Figure (**a**) show why selecting only a few projects can be inaccurate for providing a generalizable observation about code metric usefulness. By selecting one boxed group only, we can underestimate (or overestimate) the effectiveness of code metrics

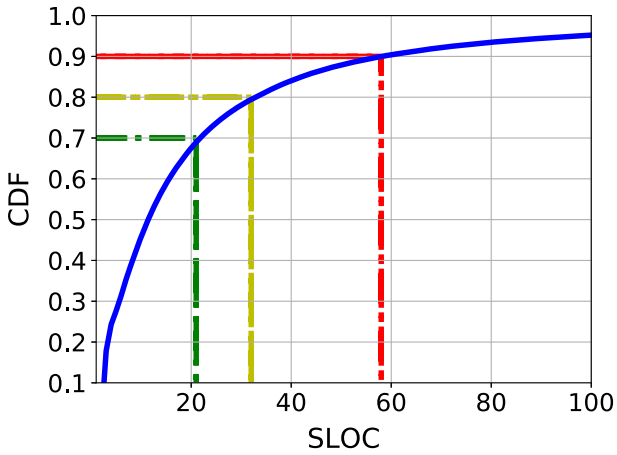
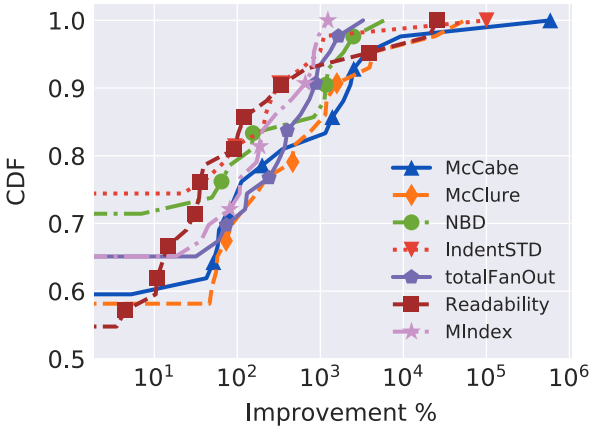


Fig. 7 Finding the critical values to determine small, medium, large, and very large methods, as proposed by Alves et al. (2010)

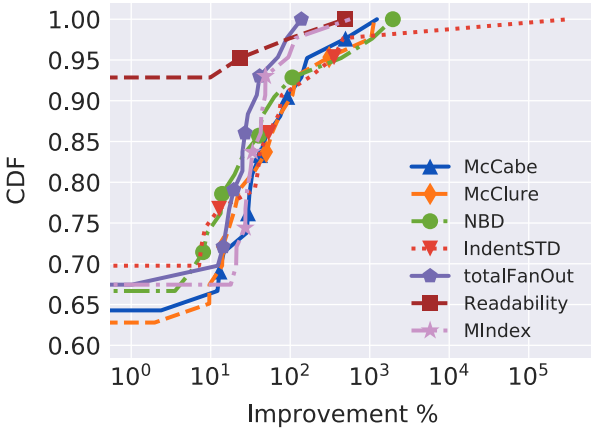
significantly when considering for large methods only. For edit distance for example (Fig. 8c), Readability was able to improve the maintenance effort prediction accuracy only for 4 projects. 4) For edit distance, NBD and IndentSTD outperform all other metrics (except for the Maintainability Index) when the evaluation was dominated by small methods (Fig. 6c). Surprisingly, their performance drops significantly for large methods. Although one may initially assume these observations as random noise, next we show that most of these performance variations are indeed explicable. We, therefore, need to consider the evaluation contexts to truly understand the usefulness of code metrics in estimating software maintenance effort.

Insight into the inconsistency of code metric performance: Source code metrics will never model maintenance effort with 100% accuracy, because there are other factors that influence how a code component evolves over time: developer habits (Terceiro et al. 2010), application domain and platforms (Zhang et al. 2013; Viggiato et al. 2019), code clones (Monden et al. 2002), software architecture (Aniche et al. 2016), and test code quality (Spadini et al. 2018; Athanasiou et al. 2014). Despite this difficulty, we take a deeper look into the following questions: i) Why is Readability's performance so poor for large methods although it is excellent for all methods (i.e., dominated by small methods)? ii) Why do NBD and IndentSTD perform so well for estimating edit distance when considered for all methods? And can we deduce a common phenomenon that explains the inconsistent performance of source code metrics?

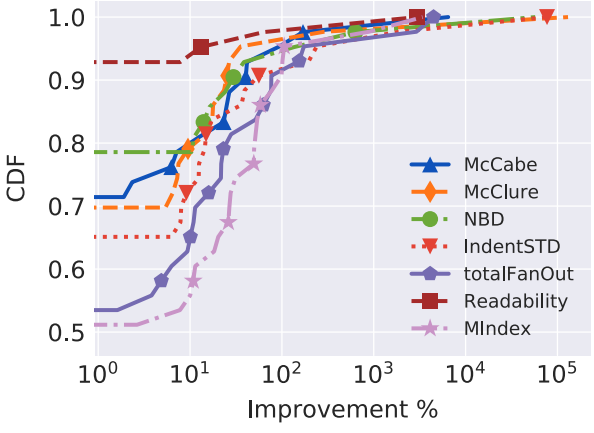
i) Readability: Figure 6a shows that Readability is the second best metric for estimating the number of revisions when the evaluation is dominated by a large number of small methods. According to the Wilcoxon rank-sum test, the performance distribution of Readability is statistically different from others. According to the Cliff's Delta test, Readability's performance has negligible effect size with IndentSTD, small effect size with NBD, and large effect size with all the others. Although its performance drops and becomes similar to the others when other maintenance indicators are used, Readability performs extremely poorly when evaluated for large methods only. For example, for edit distance and large methods, Readability is outperformed by all with large to small effect sizes. The scatter diagram in Fig. 9 indicates that the Readability metric by Buse and



(a) # revisions



(b) New additions



(c) Edit distance

Fig. 8 Percent of improvement in goodness of fit scores for large and very large methods only. Notice that all the metrics fail to improve the fitness score for a fraction of the projects. This is because many of these projects do not have enough large methods to produce regression coefficients with statistical significance (p-values are >= 0.05)

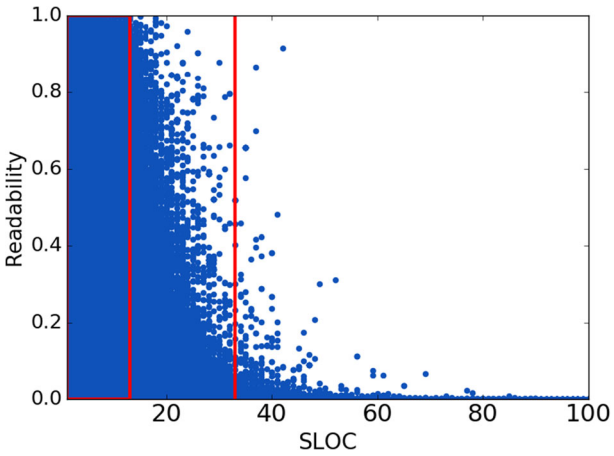


Fig. 9 Readability score against SLOC for all the methods in our dataset, based on the approach by Buse and Weimer (2010). SLOCs>100 are discarded for graph readability

Weimer (2010) can only distinguish between how readable two methods are if the methods are small in size; to this model, all the large methods are similarly less readable. Interestingly, we find that the readability model was based on small code snippets only (maximum SLOC was 11 Posnett et al. 2011). Evidently, the model does not scale for large methods. The graph shows that the variability in the readability measurements starts dwindling after SLOC 11 (first box), and almost diminishes after SLOC 32 (second bar). This clearly explains why the Readability metric performs poorly for the large methods.
ii) Nested Block Depth and IndentSTD: Figure 6c suggests that NBD and IndentSTD outperform all other metrics (except Maintainablity Index), when edit distance is the maintenance indicator. For example, NBD's performance distribution is statistically different than others with non-negligible effect sizes. Levenshtein edit distance (Levenshtein 1966) counts even the number of white spaces when it captures how many characters have been added, deleted, or edited to convert one method version into another. Although initially it seems like unimportant information to capture, it can actually indicate if a modification was done inside a nested block (by capturing the leading spaces) or outside. This is a strength of edit distance as a maintenance indicator, because modification inside nested blocks are considered more bug prone (Islam and Zibran 2020; Hindle et al. 2008). This observation explains the superior performance of NBD and IndentSTD for this maintenance indicator; among the seven code metrics, only these two capture nesting information of a method.

But why do the performance of NBD and IndentSTD drops so significantly when evaluated for large methods only? We observe that NBD and IndentSTD lack *variability* in large methods. Let us consider NBD with two large methods. One method has 2 nested loops, each with depth 3. The other one has one nested loop only, but with the same depth. NBD as a metric fails to distinguish between these two methods— NBD is 3 for both. Some metrics such as FanOut, McCabe, McClure, and Maintainability Index do not have this limitation. Their value can change monotonically with the increase in SLOC. These four metrics, in contrast to the others, do not exhibit noticeable performance degradation for large methods.

To verify if our observation about measurement variability generalizes to all the metrics, let us consider Fig. 10. Each metric measurement is represented by two box plots. The first shows the measurement distributions for all methods, and the second only for large methods. The measurement distribution of the Maintainability Index is much taller than the other metrics, because of the large default constant (171) used in its equation. For this metric and Readability, the values for large methods are smaller than small methods, because large methods are less maintainable, and less readable.

For the other five metrics, however, the second boxplot is always taller than the first one, because the measurements are naturally higher in large methods. When we compare the first boxplots for these five metrics, they are not extremely different. However, the second boxplots are noticeably different from each other, and show that some metrics (FanOut, McCabe, and McClure) have much higher variability than others (e.g., NBD, and IndentSTD) when only the large methods are considered. These observations explain the size dependent performance of metrics like NBD, and IndentSTD. Interestingly, a 1988 study by Weyuker (1988) emphasized variability in measurements to be a desired property of a code metric, stating "a measure which rates all programs as equally complex is not really a measure".

Summary: Code metrics are useful maintenance predictors, even after the size influence is completely neutralized. Their usefulness, however, depends greatly on the context in which they are applied. Some metrics are only good for particular maintenance indicators, and some only for small methods, because they reach a threshold and cannot discriminate further after the threshold measurement is reached.

6 Discussion

In this paper, we studied and reproduced the early contradictory claims about the relationship between maintenance impact and code metrics. We first focused on the methodological aspects that have influenced the outcomes of many previous studies. This investigation outlines some fundamental challenges that must be understood for accurately understanding a code metric's validity. For example, if we can not neutralize the project's difference in

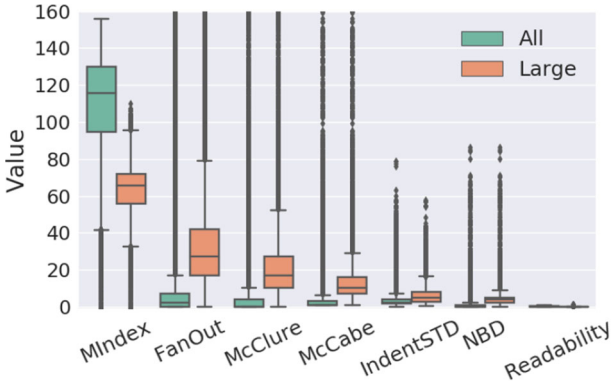


Fig. 10 Comparing variability in measurements between all methods and large methods for all the code metrics

change evolution (Fig. 1b), we should not rely on aggregated analysis. At the same time, enough projects should be considered to characterize outlying observations. While answering **RQ1**, and **RQ2** (*the confounding effect of size on the relationships between code metrics and maintenance effort*) we have established that the commonly used size normalization approach fails to neutralize the influence of size and should not be used in practice.

Using regression analysis for the size neutralized metrics evaluation (**RQ3**), we showed that we can use code metrics to prioritize our effort for reducing maintenance effort. This is encouraging for the research and development communities, because it refutes the claim that *size is the only valid code metric* (Gil and Lalouche 2017; El Emam et al. 2001; Herraiz et al. 2007), and it suggests that as a community we can continue researching new source code metrics that can provide greater insight into our software systems. The utility of a code metric, however, greatly depends on the evaluation context in which it is applied. A metric, due to the lack of variability in the measurements, may become less useful when applied to large methods (**RQ4**). Additionally, a metric's performance is impacted by the underlying maintenance indicators used.

The varying performance of code metrics suggests that building context-aware maintenance models would be more effective than trying to derive a single generic model applicable to all systems. Software maintenance models have been studied for the last forty years (Lenarduzzi et al. 2017), but considering their accuracy, there remains room for improvement (Pascarella et al. 2020). Along with other existing recommendations, such as parameter optimization (Tantithamthavorn et al. 2016), we provide convincing evidence that the community should also focus on building ensemble (Kotu and Deshpande 2015) maintenance models instead of generic models that are commonly built (e.g., Pascarella et al. 2020; Coleman et al. 1994; Nagappan and Ball 2005). The envisioned approach is not to blindly apply a mixture of different machine learning algorithms (a form of ensemble modeling Alsolai et al. 2018), but to focus more on a mixture of models where each model is trained on a selected set of code metrics, a bounded method size, and a particular maintenance indicator. For example, while Readability, NBD and IndentSTD should be used for small methods-based models, they should be excluded for other models that are better able to forecast maintenance indicators for large methods. This way we can build multiple base models based on the contextual code metrics evaluation, and can then aggregate the prediction of each base model that produces one final maintenance prediction for a given method (Kotu and Deshpande 2015).

6.1 Threats to Validity

Several threats may impact the observations in this study.

Construct validity is hampered by the maintenance indicators we used. Change proneness is not the only indicator of maintenance effort. Also, the indicators we used may not be sufficient to understand the true change proneness. Number of revisions, for example, can be impacted by the commit habits of contributors (Ståhl et al. 2019). Some may commit more frequently than others. A less revised code, which is difficult to understand, and structurally complex to make a change, may require more effort than a more revised code. We mitigated this threat, at least to some extent, by using all the four change proneness indicators that we commonly found from the literature. The accuracy of collecting the complete change history in our measurements can be criticized, because we solely relied on CodeShovel (Grund et al. 2021b). Considering the run-time performance and accuracy, however, CodeShovel is the state-of-the-art tool for tracing method history.

A metric's validity can be evaluated in multiple ways. For example, by applying measurement theory as proposed by Fenton and Kitchenham (1991). However, in this paper, we consider a code metric as a valid/useful metric only if it can offer extra insight into maintenance after the influence of size is neutralized. While this definition of validity can be argued, we found it common in previous studies (Gil and Lalouche 2017; El Emam et al. 2001; Sjøberg et al. 2013).

External validity can be limited by the representativeness of the Java open source projects we used. Our results might not generalize to closed source projects. Also, we only focused on seven selected code metrics. Although these seven metrics were sufficient to demonstrate the problems and a potential solution for reliable evaluation of code metric performance, more code metrics should be investigated in the future.

Internal validity can be criticized by the statistical tests we used; however, these tests are well-established and well-recognized that seem appropriate for our context. We have used the sum value for the change indicators; other descriptive statistics, such as mean, and median, can be explored in the future. The selected metrics of this study, although popular, can be correlated to each other, which may hinder the observations for a large set of independent code metrics.

Conclusion validity can be hampered because of our dependence on regression analysis. We mitigated this threat by relying on regression coefficients that were statistically significant (i.e., p-value $<= 0.05$).

7 Conclusion & Future Work

In this paper we set out to investigate whether code metrics can help us gain insight into maintenance effort, considering four code churn measures as maintenance effort indicators. While this question has been investigated before—and contradictory results have been presented—our approach was to reduce the level of granularity of our analysis to that of methods, and investigate the influence of size.

The key take away of our study is that *context is king*. Code metrics are useful predictors of maintenance effort, even after normalizing for size. However, their utility for predicting maintenance effort depends greatly on the context of how they are applied based on the type of maintenance indicators that are used and the size of the methods being examined.

This study presents a call-to-arms to the research community to investigate maintenance models that are context-aware, both in terms of method sizes and maintenance indicators. Implicitly, this is also a stringent warning for software engineering practitioners to not blindly follow metrics without taking context into account.

As our focus was on the previously unexplored method level granularity, we could not investigate the usefulness of some widely used class level code metrics (e.g., depth of inheritance). It would be interesting future work to see if the famous class level code metrics indeed help estimating future maintenance effort. We also plan to study the relationship between code metrics and bug-proneness with dedicated dataset reporting manually curated bugs.

Declarations

Conflict of Interests The authors declare that they have no conflict of interest.

References

Abid NJ, Sharif B, Dragan N, Alrasheed H, Maletic JI (2019) Developer reading behavior while summarizing java methods: Size and context matters. In: Proceedings of the 41st international conference on software engineering, pp 384–395

Alfadel M, Kobilica A, Hassine J (2017) Evaluation of halstead and cyclomatic complexity metrics in measuring defect density. In: 2017 9th IEEE-GCC conference and exhibition, pp 1–9

Alsolai H, Roper M, Nassar D (2018) Predicting software maintainability in object-oriented systems using ensemble techniques. In: 2018 IEEE International conference on software maintenance and evolution, pp 716–721

Alves TL, Ypma C, Visser J (2010) Deriving metric thresholds from benchmark data. In: IEEE International conference on software maintenance, pp 1–10

Aniche MF, Treude C, Zaidman A, van Deursen A, Gerosa MA (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, pp 41–50

Antinyan V, Staron M, Derehag J, Runsten M, Wikström E, Meding W, Henriksson A, Hansson J (2015) Identifying complex functions: By investigating various aspects of code complexity. In: 2015 Science and information conference (SAI), pp 879–888

Antinyan V, Staron M, Meding W, Österström P, Wikstrom E, Wranker J, Henriksson A, Hansson J (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, pp 154–163

Antinyan V, Staron M, Sandberg A (2017) Evaluating code complexity triggers, use of complexity measures and the influence of code complexity on maintenance time. Empirical Softw Engg 22(6):3057–3087

Athanasiou D, Nugroho A, Visser J, Zaidman A (2014) Test code quality and its relation to issue handling performance. IEEE Trans Software Eng 40(11):1100–1125

Bandi RK, Vaishnavi VK, Turk DE (2003) Predicting maintenance performance using object-oriented design complexity metrics. IEEE Trans Softw Eng 29(1):77–87

Bauer J, Siegmund J, Peitek N, Hofmeister JC, Apel S (2019) Indentation: Simply a matter of style or support for program comprehension? In: IEEE/ACM 27th International conference on program comprehension, pp 154–164

Bavota G, Linares-Vásquez M, Bernal-Cárdenas CE, Penta MD, Oliveto R, Poshyvanyk D (2015) The impact of api change- and fault-proneness on the user ratings of Android apps. IEEE Trans Softw Eng 41(4):384–407

Bell RM, Ostrand TJ, Weyuker EJ (2011) Does measuring code change improve fault prediction? In: Proceedings of the 7th international conference on predictive models in software engineering, Promise '11

Börstler J, Paech B (2016) The role of method chains and comments in software readability and comprehension–an experiment. IEEE Trans Softw Eng 42(9):886–898

Brittain JM (1982) Pitfalls of user research, and some neglected areas. Soc Sci Inf Stud 2(3):139–148

Buse RPL, Weimer WR (2010) Learning a metric for code readability. IEEE Trans Softw Eng 36(4):546–558

Catolino G, Palomba F, De Lucia A, Ferrucci F, Zaidman A (2018) Enhancing change prediction models using developer-related factors. J Syst Softw 143:14–28

Chen YT, Gopinath R, Tadakamalla A, Ernst MD, Holmes R, Fraser G, Ammann P, Just R (2020) Revisiting the relationship between fault detection, test adequacy criteria, and test set size. In: 2020 35th IEEE/ACM International conference on automated software engineering (ASE), pp 237–249

Chidamber SR, Kemerer CF (1994) A metrics suite for object oriented design. IEEE Trans Softw Eng 20(6):476–493

Chowdhury S, Borle S, Romansky S, Hindle A (2019) Greenscaler: Training software energy models with automatic test generation. Emp Softw Eng: Int J 24(4):1649–1692

Coleman D, Ash D, Lowther B, Oman P (1994) Using metrics to evaluate software system maintainability. Computer 27(8):44–49

Cruz L, Abreu R, Grundy J, Li L, Xia X (2019) Do energy-oriented changes hinder maintainability? In: 2019 IEEE International conference on software maintenance and evolution, pp 29–40

Curtis B, Sheppard SB, Milliman P, Borst MA, Love T (1979) Measuring the psychological complexity of software maintenance tasks with the halstead and mccabe metrics. IEEE Trans Softw Eng SE-5(2):96–104

Darcy DP, Kemerer CF, Slaughter SA, Tomayko JE (2005) The structural complexity of software an experimental test. IEEE Trans Softw Eng 31(11):982–995

Ebert C, Cain J, Antoniol G, Counsell S, Laplante P (2016) Cyclomatic complexity. IEEE Softw 33(6):27–29
El Emam K, Benlarbi S, Goel N, Rai SN (2001) The confounding effect of class size on the validity of object-oriented metrics. IEEE Trans Softw Eng 27(7):630–650
Fenton N, Kitchenham B (1991) Validating software measures. Softw Test Verif Reliab 1(2):27–42
Giger E, D'Ambros M, Pinzger M, Gall HC (2012) Method-level bug prediction. In: Proceedings of the 2012 ACM-IEEE international symposium on empirical software engineering and measurement, pp 171–180
Gil Y, Lalouche G (2016) When do software complexity metrics mean nothing? — when examined out of context. J Obj Technol 15(1):2:1–25
Gil Y, Lalouche G (2017) On the correlation between size and metric validity. Empir Softw Eng 22(5):2585–2611
Gopinath R, Jensen C, Groce A (2014) Code coverage for suite evaluation by developers. In: Proceedings of the 36th international conference on software engineering, pp 72–82
Grund F, Chowdhury S, Bradley NC, Hall B, Holmes R (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), pp 221–222
Grund F, Chowdhury S, Bradley NC, Hall B, Holmes R (2021) Codeshovel: Constructing method-level source code histories. In: 2021 IEEE/ACM 43rd international conference on software engineering (ICSE), pp 1510–1522
Hata H, Mizuno O, Kikuno T (2011) Historage: Fine-grained version control system for java. In: Proc. International workshop on principles of software evolution and ERCIM workshop on software evolution, pp 96–100
Herraiz I, Gonzalez-Barahona JM, Robles G (2007) Towards a theoretical model for software growth. In: Fourth international workshop on mining software repositories, pp 21–21
Herzig K, Zeller A (2013) The impact of tangled code changes. In: 2013 10th Working conference on mining software repositories, pp 121–130
Higo Y, Hayashi S, Kusumoto S (2020) On tracking java methods with git mechanisms. J Syst Softw 165:110571
Hindle A, Godfrey MW, Holt RC (2008) Reading beside the lines: Indentation as a proxy for complexity metric. In: 16th IEEE International conference on program comprehension, pp 133–142
Hofmeister J, Siegmund J, Holt DV (2017) Shorter identifier names take longer to comprehend. In: IEEE 24th International conference on software analysis, evolution and reengineering, pp 217–227
Inozemtseva L, Holmes R (2014) Coverage is not strongly correlated with test suite effectiveness. In: Proceedings of the 36th international conference on software engineering, pp 435–445
Islam MR, Zibran MF (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, pp 1523–1531
Johnson J, Lubo S, Yedla N, Aponte J, Sharif B (2019) An empirical study assessing source code readability in comprehension. In: 2019 IEEE International conference on software maintenance and evolution, pp 513–523
Just R, Jalali D, Inozemtseva L, Ernst MD, Holmes R, Fraser G (2014) Are mutants a valid substitute for real faults in software testing? In: Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering, pp 654–665
Kafura D, Reddy GR (1987) The use of software complexity metrics in software maintenance. IEEE Trans Softw Eng SE-13(3):335–343
Khomh F, Penta MD, Guéhéneuc Y-G, Antoniol G (2012) An exploratory study of the impact of antipatterns on class change- and fault-proneness. Emp Softw Eng: Int J 17(3):243–275
Kondo M, German DM, Mizuno O, Choi E-H (2020) The impact of context metrics on just-in-time defect prediction. Emp Softw Eng 25(1):890–939
Kotu V, Deshpande B (2015) Chapter 2 - data mining process. In: Kotu V, Deshpande B (eds) Predictive analytics and data mining. Morgan Kaufmann, Boston, pp 17–36
Lake A, Cook CR: **Use of factor analysis to develop oop software complexity metrics**. USA, 1994
Landman D, Serebrenik A, Vinju J (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, pp 221–230
Lehman MM, Ramil JF, Wernick PD, Perry DE, Turski WM (1997) Metrics and laws of software evolution-the nineties view. In: International software metrics symposium, pp 20–32
Lenarduzzi V, Sillitti A, Taibi D (2017) Analyzing forty years of software maintenance models. In: International conference on software engineering companion (ICSE-C), pp 146–148
Levenshtein VI (1966) Binary codes capable of correcting deletions, insertions, and reversals. In: Soviet physics doklady, vol 10, pp 707–710

Matter D, Kuhn A, Nierstrasz O (2009) Assigning bug reports using a vocabulary-based expertise model of developers. In: 2009 6th IEEE International working conference on mining software repositories, pp 131–140
McCabe TJ (1976) A complexity measure. IEEE Trans Softw Eng SE-2(4):308–320
McClure CL (1978) A model for program complexity analysis. In: Proceedings of the 3rd international conference on software engineering, pp 149–157
Menzies T, Greenwald J, Frank A (2007) Data mining static code attributes to learn defect predictors. IEEE Trans Softw Eng 33(1):2–13
Mo R, Cai Y, Kazman R, Xiao L, Feng Q (2016) Decoupling level: A new metric for architectural maintenance complexity. In: 2016 IEEE/ACM 38th international conference on software engineering, pp 499–510
Mocku A, Votta LG (2000) Identifying reasons for software changes using historic databases. In: Proceedings 2000 International conference on software maintenance, pp 120–130
Monden A, Nakae D, Kamiya T, Sato S, Matsumoto K (2002) Software quality analysis by code clones in industrial legacy software. In: Proceedings IEEE symposium on software metrics, pp 87–94
Moser R, Pedrycz W, Succi G (2008) Analysis of the reliability of a subset of change metrics for defect prediction. In: Proceedings of the Second ACM-IEEE international symposium on empirical software engineering and measurement, ESEM '08, pp 309–311
Nagappan N, Ball T (2005) Use of relative code churn measures to predict system defect density. In: Proceedings. 27th International conference on software engineering, pp 284–292
Oman P, Hagemeister J (1992) Metrics for assessing a software system's maintainability. In: Proceedings conference on software maintenance 1992, pp 337–344
Palomba F, Zaidman A, Oliveto R, De Lucia A (2017) An exploratory study on the relationship between changes and refactoring. In: Proceedings of the 25th international conference on program comprehension, pp 176–185
Pantiuchina J, Lanza M, Bavota G (2018) Improving code: The (mis) perception of quality metrics. In: IEEE International conference on software maintenance and evolution, pp 80–91
Papadakis M, Shin D, Yoo S, Bae D-H (2018) Are mutation scores correlated with real fault detection? A large scale empirical study on the relationship between mutants and real faults. In: Proceedings of the 40th international conference on software engineering, pp 537–548
Pascarella L, Palomba F, Bacchelli A (2020) On the performance of method-level bug prediction: A negative result. J Syst Softw, 161
Posnett D, Hindle A, Devanbu P (2011) A simpler model of software readability. In: Proceedings of the 8th working conference on mining software repositories, pp 73–82
Radjenović D, Heričko M, Torkar R, Živkovič A (2013) Software fault prediction metrics: A systematic literature review. Inf Softw Technol 55(8):1397–1418
Rahman MS, Roy CK (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), pp 131–140
Ralph P, Tempero E (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, pp 13–23
Ray B, Hellendoorn V, Godhane S, Tu Z, Bacchelli A, Devanbu P (2016) On the "naturalness" of buggy code. In: Proceedings of the 38th international conference on software engineering. ICSE '16, pp 428–439
Robert B, Correia JP, Schill K, Visser J (2012) Standardized code quality benchmarking for improving software maintainability. Softw Qual J 20:287–307
Romano D, Pinzger M (2011) Using source code metrics to predict change-prone java interfaces. In: 2011 27th IEEE International conference on software maintenance, pp 303–312
Romano J, Kromrey JD, Coraggio J, Skowronek J (2006) Appropriate statistics for ordinal level data: Should we really be using t-test and cohen'sd for evaluating group differences on the nsse and other surveys. In: Annual meeting of the Florida association of institutional research, pp 1–33
Scalabrino S, Bavota G, Vendome C, Linares-Vásquez M, Poshyvanyk D, Oliveto R (2017) Automatically assessing code understandability: How far are we? In: 32nd IEEE/ACM International conference on automated software engineering, pp 417–427
Scalabrino S, Linares-Vásquez M, Poshyvanyk D, Oliveto R (2016) Improving code readability models with textual features. In: IEEE 24th International conference on program comprehension, pp 1–10
Scholtes I, Mavrodiev P, Schweitzer F (2016) From aristotle to ringelmann: A large-scale analysis of team productivity and coordination in open source software projects. Emp Softw Eng: Int J 21(2):642–683
Shepperd M (1988) A critique of cyclomatic complexity as a software metric. Softw Eng J 3(2):30–36
Sheskin DJ (2020) Handbook of parametric and nonparametric statistical procedures. CRC Press

Shihab E, Hassan AE, Adams B, Jiang ZM (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

Shin Y, Meneely A, Williams L, Osborne JA (2011) Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. IEEE Trans Softw Eng 37(6):772–787

Sjøberg DIK, Yamashita A, Anda BCD, Mockus A, Dybå T (2013) Quantifying the effect of code smells on maintenance effort. IEEE Trans Softw Eng 39(8):1144–1156

Spadini D, Palomba F, Zaidman A, Bruntink M, Bacchelli A (2018) On the relation of test smells to software code quality. In: 2018 IEEE International conference on software maintenance and evolution, pp 1–12

Sridhara G, Hill E, Muppaneni D, Pollock L, Vijay-Shanker K (2010) Towards automatically generating summary comments for java methods. In: Proceedings of the IEEE/ACM International conference on automated software engineering, pp 43–52

Ståhl D, Martini A, Mårtensson T (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), pp 81–90

Subandri MA, Sarno R (2017) Cyclomatic complexity for determining product complexity level in cocomo ii. Proced Comput Sci 124:478–486. 4th Information systems international conference 2017, ISICO 2017, 6-8 November 2017, Bali, Indonesia

Suh SD, Neamtiu I (2010) Studying software evolution for taming software complexity. In: 21st Australian software engineering conference, pp 3–12

Tantithamthavorn C, McIntosh S, Hassan AE, Matsumoto K (2016) Automated parameter optimization of classification techniques for defect prediction models. In: IEEE/ACM 38th International conference on software engineering, pp 321–332

Terceiro A, Rios LR, Chavez C (2010) An empirical study on the structural complexity introduced by core and peripheral developers in free software projects. In: Brazilian symposium on software engineering, pp 21–29

Thode HC (2002) Testing for normality, vol 164. CRC press

Tiwari U, Kumar S (2014) Cyclomatic complexity metric for component based software. SIGSOFT Softw Eng Notes 39(1):1–6

Tosun A, Bener A, Turhan B, Menzies T (2010) Practical considerations in deploying statistical methods for defect prediction: A case study within the turkish telecommunications industry. Inf Softw Technol 52(11):1242–1257

Viggiato M, Oliveira J, Figueiredo E, Jamshidi P, Kästner C (2019) How do code changes evolve in different platforms? A mining-based investigation. In: 2019 IEEE International conference on software maintenance and evolution, pp 218–222

Wang Q, Xia X, Lo D, Li S (2019) Why is my code change abandoned? Inf Softw Technol 110:108–120

Weyuker EJ (1988) Evaluating software complexity measures. IEEE Trans Softw Eng 14(9):1357–1365

Ying ATT, Murphy GC, Ng R, Chu-Carroll MC (2004) Predicting source code changes by mining change history. IEEE Trans Softw Eng 30(9):574–586

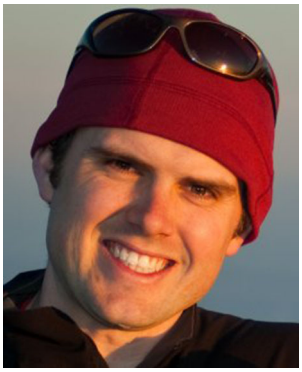
Yu L, Mishra A (2013) An empirical study of Lehman's law on software quality evolution

Zhang F, Mockus A, Zou Y, Khomh F, Hassan AE (2013) How does context affect the distribution of software maintainability metrics? In: IEEE International conference on software maintenance, pp 350–359

Zhou Y, Xu B, Leung H (2010) On the ability of complexity metrics to predict fault-prone classes in object-oriented systems. J Syst Softw 83(4):660–674

Zimmermann T, Premraj R, Zeller A (2007) Predicting defects for eclipse. In: Proceedings of the third international workshop on predictor models in software engineering, p 9

Shaiful Chowdhury is now a postdoctoral fellow at the University of Calgary. This paper was completed when he was an NSERC Postdoctoral Fellow at the University of British Columbia, Canada. He received his Ph.D from the University of Alberta, Canada which won the outstanding PhD thesis award. He received his MSc and BSc degrees in Computer Science from University of Saskatchewan, Canada and University of Chittagong, Bangladesh respectively. Shaiful's research interest includes software maintenance, software energy modeling and efficiency, and mining software repositories. Among many other awards, Shaiful won an ACM SIGSOFT DISTINGUISHED paper award (at ICSE 2021), and the Early Achievement Award in PhD (Computing Science) at the University of Alberta. He also received the mining challenge paper award at MSR 2015.



Reid Holmes is an Associate Professor in the Department of Computer Science at the University of British Columbia. His research interests include understanding how software engineers build and maintain complex systems. This understanding is generally translated into tools and techniques that can be validated in practice. He was previously an Assistant Professor at the University of Waterloo and a postdoctoral fellow at the University of Washington. He earned his Ph.D. at the University of Calgary, and his M.Sc. and B.Sc. at the University of British Columbia.

Andy Zaidman is currently a full professor in software engineering at Delft University of Technology, the Netherlands. He has received the MSc and PhD degrees in computer science from the University of Antwerp, Belgium, in 2002 and 2006, respectively. His main research interests include software evolution, software quality, mining software repositories, and software testing. He is an active member of the research community and involved in the organization of numerous conferences (WCRE'08, WCRE'09, VISSOFT'14 and MSR'18). In 2013 he was the laureate of a prestigious Vidi career grant from the Dutch science foundation NWO, while in 2019 he was awarded the Vici career grant, the most prestigious career grant from the Dutch science foundation NWO.

Rick Kazman is a Professor at the University of Hawaii and a Visiting Researcher at the Software Engineering Institute of Carnegie Mellon University. His primary research interests are software architecture, design and analysis tools, software visualization, and software engineering economics. Kazman has been involved in the creation of several highly influential methods and tools for architecture analysis, including the ATAM (Architecture Tradeoff Analysis Method), the CBAM (Cost-Benefit Analysis Method) and the Dali and Titan tools.