POLITECNICO DI TORINO
Repository ISTITUZIONALE

A Tool-Based Perspective on Software Code Maintainability Metrics: A Systematic Literature Review

Original
A Tool-Based Perspective on Software Code Maintainability Metrics: A Systematic Literature Review / Ardito, L., Coppola, R., Barbato, L., Verga, D.. - In: SCIENTIFIC PROGRAMMING. - ISSN 1058-9244. - ELETTRONICO. - 2020 (8840389):(2020), pp. 1-26. [10.1155/2020/8840389]

Availability:
This version is available at: 11583/2842312 since: 2020-10-22T16:51:57Z

Publisher:
Hindawi

Published
DOI:10.1155/2020/8840389

Terms of use:

This article is made available under terms and conditions as specified in the corresponding bibliographic description in the repository

Publisher copyright

(Article begins on next page)

15 October 2026

Hindawi
Scientific Programming
Volume 2020, Article ID 8840389, 26 pages
https://doi.org/10.1155/2020/8840389

Review Article

A Tool-Based Perspective on Software Code Maintainability Metrics: A Systematic Literature Review

Luca Ardito,[1] **Riccardo Coppola**,[1] **Luca Barbato**,[2] **and Diego Verga**[1]

[1]*Politecnico di Torino Department of Control and Computer Engineering Turin, Turin, Italy*
[2]*Luminem, Turin, Italy*

Correspondence should be addressed to Luca Ardito; luca.ardito@polito.it

Received 19 March 2020; Accepted 17 July 2020; Published 4 August 2020

Academic Editor: Daniela Briola

Copyright © 2020 Luca Ardito et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

Software maintainability is a crucial property of software projects. It can be defined as the ease with which a software system or component can be modified to be corrected, improved, or adapted to its environment. The software engineering literature proposes many models and metrics to predict the maintainability of a software project statically. However, there is no common accordance with the most dependable metrics or metric suites to evaluate such nonfunctional property. The goals of the present manuscript are as follows: (i) providing an overview of the most popular maintainability metrics according to the related literature; (ii) finding what tools are available to evaluate software maintainability; and (iii) linking the most popular metrics with the available tools and the most common programming languages. To this end, we performed a systematic literature review, following Kitchenham's SLR guidelines, on the most relevant scientific digital libraries. The SLR outcome provided us with 174 software metrics, among which we identified a set of 15 most commonly mentioned ones, and 19 metric computation tools available to practitioners. We found optimal sets of at most five tools to cover all the most commonly mentioned metrics. The results also highlight missing tool coverage for some metrics on commonly used programming languages and minimal coverage of metrics for newer or less popular programming languages. We consider these results valuable for researchers and practitioners who want to find the best selection of tools to evaluate the maintainability of their projects or to bridge the discussed coverage gaps for newer programming languages.

1. Introduction

Nowadays, software security and resilience have become increasingly important, given how pervasive the software is. Effective tools and programming languages can

(i) discover mistakes earlier
(ii) reduce the odds of their occurrence
(iii) make a large class of common errors impossible by restricting at compile time what the programmer can do

Several best practices are consolidated in software engineering, e.g., continuous integration, testing with code coverage measurement, and language sanitization. All these techniques allow the application of code analysis tools automatically, which can provide a significant enhancement of the source code quality and allow software developers to efficiently detect vulnerabilities and faults [1]. However, the lack of comprehensive tooling may render it challenging to apply the same code analysis strategies to software projects developed with different languages or for different domains.

The literature defines software maintainability as the ease with which a software system or component can be modified to correct faults, improve performance or other attributes, or adapt to a changing environment [2]. Thus, maintainability is a highly significant factor in the economic success of software products. Several studies have described models and frameworks, based on software metrics, to predict or infer the maintainability of a software project [3–5]. However, although many different metrics have been proposed by the scientific literature over the course of the last 40 years, the available models are very language- and domain-

specific, and there is still no accordance in the industry and academia about a universal set of metrics to adopt to evaluate software maintainability [6].

This work aims at answering the primary need of identifying evaluation frameworks for different programming languages, either affirmed or newly emerged, e.g., the Rust programming language, developed by Mozilla Research as a language similar in characteristics to C++, but with better code maintainability, memory safety, and performance [7, 8].

Thus, the first goal of this paper is to find which are the most commonly mentioned metrics in the state-of-the-art literature. We focused on static metrics since the analysis of dynamic metrics (i.e., metrics collected during the execution of adequately instrumented software [9]) was out of the scope of this work.

The second goal of the paper is to determine which tools are more commonly used in the literature to calculate source code metrics. Based on the mostly used tools, we then define an optimal selections able to compute the most popular metrics for a set of programming languages.

To pursue both goals we

(i) applied the systematic literature review (SLR) methodology on a set of scientific libraries

(ii) performed a thorough analysis of all the primary studies, available in the literature, about the topic of software metrics for maintainability

Hence, this manuscript provides the following contributions to researchers and practitioners:

(i) The definition of the most mentioned metrics that can be used to measure software maintainability for software projects

(ii) Details about closed-source and open-source tools that can be leveraged by practitioners to evaluate the quality of their software projects

(iii) Optimal sets of open-source tools that can be leveraged to

investigate the computation of software metrics for maintainability
adopt them in evaluation frameworks
adapt them to other programming languages that are currently not supported

The remainder of the manuscript is structured as follows:

(i) Section 2 describes the approach we adopted to conduct our SLR

(ii) Section 3 presents a discussion of the results obtained by applying such approach

(iii) Section 4 discusses the threats to the validity of the present study

(iv) Section 5 provides a comparison of this study with existing related work in the literature

(v) Section 6 concludes the paper and provides directions for future research

2. Research Method

In this section, we outline the method that we utilized to realize this study. We performed a systematic literature review (from now on, SLR), following the guidelines provided by Barbara and Charters [10] to structure the work and report it in an organized and replicable manner.

An SLR is considered one of the key research methodologies of evidence-based software engineering (EBSE) [11]. The methodology has gained significant attention from software engineering researchers in recent years [12]. SLRs all include three fundamental phases: (i) planning the review (which includes specifying its goals and research questions); (ii) conducting the review (which includes querying article repositories, selecting the studies, and performing data extraction); and (iii) reporting the review.

All those steps have been undertaken during this research and are detailed in the following sections of this paper.

2.1. Planning. According to Barbara and Charters guidelines, the planning phase of an SLR involves the identification of the need for the review (hence the definition of its goals), the definition of the research questions that will guide the review, and the development of the review protocol we will use.

2.1.1. Goals. The need for the review, as said in the introduction section, came from the need to improve the software maintainability, in terms of clarity of its source code, while implementing complex algorithms. Our primary objective was to identify a dependable set of metrics widely used in the literature and computed for software usage with available tools.

The objectives of our research are defined by using the Goal-Question-Metric paradigm by van Solingen et al. [13]. Specifically, we based our research on the following goals:

(i) Goal 1: have an overview of the most used metrics in the literature in the last few years

(ii) Goal 2: find what tools have been used in (or described by) the literature about maintainability metrics

(iii) Goal 3: find a mapping between the most common metrics and the tools able to compute them

2.1.2. Research Questions. Based on the goals defined above, our study entailed answering the research questions defined in the following:

(i) RQ1.1: what are the metrics used to evaluate code maintainability available in the literature?

Our aim for this research question is to determine what metrics are present in the literature and how popular they are in manuscripts about code maintainability.

(ii) RQ1.2: which of the metrics we found are the most popular in the literature?

This research question aims at characterizing the different metrics obtained from answering RQ1.1 based on their popularity and adoption.

(iii) RQ2.1: what tools are available to perform code evaluation?

The expected result of this research question is a list of tools, both closed source and open source, along with the metrics they can calculate.

(iv) RQ2.2: what is the ideal selection of tools able to apply the most popular metrics for the most supported programming languages?

This research question entails measuring the coverage provided by the set of the most popular metrics for each language and providing the optimal set of tools that can compute those metrics.

2.1.3. Selected Digital Libraries. The search strategy involves the selection of the search resources and the identification of the search terms. For this SLR, we used the following digital libraries:

(i) ACM Digital Library
(ii) IEEE Xplore
(iii) Scopus
(iv) Web of Science

2.1.4. Search Strings. The formulation of the search strings is crucial for the definition of the search strategy of the SLR. According to the guidelines defined by Kitchenham et al., the first operation in defining the search string involved an analysis of the main keywords used in the RQs, their synonyms, and other possible spellings of such words.

In this phase, all the researchers collaboratively selected several pilot studies. The selected pilot studies are presented in Table 1 and are related to the target research domain.

These studies are selected to be used to verify the goodness of the research queries: the researchers should review the queries if the pilot studies are not present after the refining phase.

The starting keywords identified were *software*, *maintainability*, and *metrics*. The search string "software maintainability metric" was hence used to perform the first search on the selected digital libraries. Our results include articles published between 2000 and 2019.

This first search pointed out that adding the *code* synonym of the keyword *software* added a large numbers of papers to the results.

Also, the following keywords were excluded from the search to reduce the number of unfitting papers from the results:

(i) *Defect* and *fault*, to avoid considering manuscripts more related to the topic of verification and validation, error-proneness, and software reliability prediction, than to code maintainability
(ii) *Co-change*, to avoid considering manuscripts more related to the topic of code evolution
(iii) *Policy-driven* and *design*, to avoid considering manuscripts more related to the definition and usage of metrics used to design software, instead of evaluating existing code

Table 2 reports the search queries before and after excluding the keywords listed above, for each of the chosen digital libraries.

2.1.5. Inclusion and Exclusion Criteria. The final phase of the study selection uses the studies obtained by applying the final search queries detailed below.

The following are the inclusion criteria used for the study selection:

IC1: studies written in a language comprehensible by the authors

IC2: studies presenting a new metric accurately

IC3: studies that present, analyze, or compare known metrics or tools

IC4: detailed primary studies

On the other hand, in the following are defined the exclusion criteria:

EC1: studies written in a language not directly comprehensible by the authors, i.e., not written in English, Italian, Spanish, or Portuguese

EC2: studies that present a novel metric, but not do not describe it accurately

EC3: studies that do not describe or use metrics or tools

EC4: secondary studies (e.g., systematic literature reviews, surveys, and mappings)

2.2. Conducting. After defining the review protocol in the planning phase, the conducting phase involves its actual application, the selection of papers by application of the search strategy, and the extraction of relevant data from the selected primary studies.

2.2.1. Study Search. This phase consisted of gathering all the studies by applying the search strings formulated and discussed in Section 2.1.4 to the selected digital libraries. To this end, we leveraged the *Publish or Perish (PoP)* tool [17]. To aid the replicability of the study, we report that we performed the last search iterations at the end of October 2019. After the application of the queries and the removal of the duplicate papers on the four considered digital libraries, 801 unique papers were gathered (see Table 3). The result of this phase is a list of possible papers that must be subject to the application of exclusion and inclusion criteria. This action allows having a final verdict for their selection as primary studies for our SLR. We exported the mined papers in a CSV file with basic information about each extracted manuscript.

Table 1: Pilot studies.

ID	Authors	Title	Year
[14]	Ostberg and Wagner	On automatically collectable metrics for software maintainability evaluation	2014
[15]	Ludwig et al.	Compiling static software metrics for reliability and maintainability from GitHub repositories	2017
[5]	Kaur et al.	Software maintainability prediction by data mining of software code metrics	2014
[6]	Sarwar et al.	A comparative study of MI tools: defining the roadmap to MI tool standardization	2008
[16]	Liu et al.	Evaluate how cyclomatic complexity changes in the context of software evolution	2018

Table 2: Search strings for the selected digital libraries.

Library	Before refinement	After refinement
ACM Digital Library	+("software" "code")+(metrics)+(maintainability)	+("software" "code")+(metrics)+(maintainability)-(defect)-(fault)-(co-change)-(policy-driven)
IEEE Xplore	(((code OR software) AND metrics) AND maintainability)	(((((((((code OR software) AND metrics) AND maintainability) NOT defect) NOT fault) NOT co-change) NOT policy-driven) NOT design)
Scopus	(code OR software) AND metrics AND maintainability	code OR software AND metrics AND maintainability AND NOT fault AND NOT defect AND NOT co-change AND NOT policy-driven AND NOT design
Web of Science	(code OR software) AND metrics AND maintainability	code OR software AND metrics AND maintainability NOT fault NOT defect NOT co-change NOT policy-driven NOT design

Table 3: Number of manuscripts collected from the selected digital libraries.

Library	Before refinement	After refinement	Without duplicates
ACM Digital Library	497	152	—
IEEE Xplore	443	215	—
Scopus	848	381	—
Web of Science	599	300	—
Total	2387	1048	801

2.2.2. Study Selection. The authors of this SLR carried the paper selection process independently. To analyze the papers, we used a 5-point Likert scale, instead of dividing them between the fitting and unfitting. We performed the following assignation:

(i) One point to the papers that matched exclusion criteria and did not match any inclusion criteria

(ii) Two points to papers that matched some exclusion criteria and some inclusion criteria

(iii) Three points to papers that did not match any criteria (neither exclusion or inclusion)

(iv) Four points to papers that matched some, but not all, inclusion criteria

(v) Five points to papers that matched all inclusion criteria

We analyzed the studies in two different steps: first, the title and abstract for finding immediate compliance of the paper to the inclusion and exclusion criteria. For papers that received 3 points after reading the title and abstract, the full text was read, with particular attention to possible usage or definition or metrics throughout the body of the article. At the end of the second read, none of the uncertain studies were evaluated as fitting with our research needs, and hence, no other primary study was added to our final pool.

During this phase, we also applied the process of snowballing. Snowballing refers to using the reference list of the included papers to identify additional papers [18]. The application of snowballing, for this specific SLR, did not lead to any additional paper to take into consideration.

2.2.3. Data Extraction. In this phase, we read each identified primary studies again, to mine relevant data for addressing the formulated RQs. We have created a spreadsheet form to be compiled for each of the considered papers, and that contained the data of interest subdivided by the RQ they concurred to answer. The data extraction phase, again, was performed by all the authors of the papers in an independent manner.

For each paper, we collected some basic context information:

(i) Year of publication

(ii) Number of times the paper was viewed fully and number of citations

(iii) Authors and location of the authors

To answer RQ1.1, we needed to inspect the set of primary studies to understand which metrics they defined or mentioned. Hence, for each paper, we extracted the following data:

(i) The list of metrics and metric suites utilized in each paper

(ii) The programming languages and the family of programming language (e.g., C-like and object oriented) for which the used or proposed metrics can be computed

To answer RQ1.2, we wanted to give an additional classification of the metrics, other than the number of mentions. We took in consideration the opinion of the authors on each of the metrics studied in their papers. This allowed us to evaluate if a metric is considered useful or not in most papers. This analysis allowed us to take into consideration the popularity of the metrics by counting the difference between positive and negative citations by authors.

To answer RQ2.1, we needed to inspect the primary studies to understand which tools they presented or used to compute the metrics that were adopted. For each paper that mentioned tools, we hence gathered the following information:

(i) The list of tools described, used, or cited by each paper

(ii) When possible, the list of metrics that can be calculated by each tool

(iii) The list of programming languages on which the tool can operate

(iv) The type of the tool, i.e., the fact that the tool is open source or not

Finally, to answer RQ2.2, we had to correlate the information gathered for the previous research questions. We achieved this by finding the tool or tools covering the metrics that proved to be the most popular among selected primary studies.

2.2.4. Data Synthesis and Reporting. In this phase, we elaborated the data extracted and synthesized previously to obtain a response for each of the research questions we had. Having all the data we needed, in the shape of a form per paper analyzed, we proceeded with the data synthesis.

We gathered all the metric suites and the metrics we found in tables, keeping track of the papers mentioning them. We computed aggregate measures on the popularity value assigned to each metric.

3. Results

This section describes the results obtained to answer the research questions described in Section 2.1.2. The appendices of this paper report the complete tables with the extracted data to improve the readability of this manuscript.

At the end of this phase, we collected a final set of 43 primary studies for the subsequent phase of our SLR. Figure 1 reports the distribution over the considered time frame of the selected papers, and Figure 2 indicates the distribution of authors of related studies over the world. We report the selected papers in Table 4. The statistic seems to suggest that the interest in software maintainability metrics had grown since 2008 and has increased in the latest years since 2016 (see barplot in Figure 1).

3.1. RQ1.1: Available Metrics. The papers selected as primary studies for our SLR cited a total of 174 different metrics. We report all the metrics in Table 5 in the appendix. The table reports

(i) the metric suite (empty if the metric is not part of any specific suite)

(ii) the metric name (acronym, if existing, and a full explanation, if available)

(iii) the list of papers that mention the metric. The last two columns, respectively, report

(iv) the total number of papers mentioning the metric (i.e., the number of studies in the third column)

(v) the score we gave to each metric

We computed the score in the following way:

(i) +1 if the study used (or defined) the metric or the authors of the study expressed a positive opinion about it

(ii) −1 if the paper criticized the metric

By examining the last two columns of the metrics table, it can be seen that the last two columns are most of the times identical. This is because the majority of the papers we found just utilize the metrics without commenting them, neither positively or negatively.

It is immediately evident that some suites and metrics are taken into consideration much more often than others. More than 75% of the metrics are mentioned by just a single paper. The boxplots in Figure 3 show, in red, the distribution of the total number of mentions and the score for all the considered metrics. It is evident, from the boxplots, that the difference between the two distribution is rather limited, confirming the vast majority of neutral or positive opinions when the metrics are referenced in a research paper. Since only 24.7% of the metrics are used by more than one of our selected studies, the median values of both the measured indicators, "TOT" and "Score", are equal to 1 if the whole set of metrics is considered.

In general, however, it is worth underlining that a low score does not necessarily mean that the metric is of lesser quality but instead that it is less known in the related literature. Another interesting thing to point out is that we did not find a particular metric that received many negative scores.

3.2. RQ1.2: Most Mentioned Metrics. Since our analysis was aimed at finding the most popular metrics, to extract a set of them to be declined to different languages, we were interested in finding metrics mentioned by multiple papers. In Table 6 we report metrics that were used by at least two papers among the selected primary studies. This operation allowed us to reduce the noise caused by metrics that were

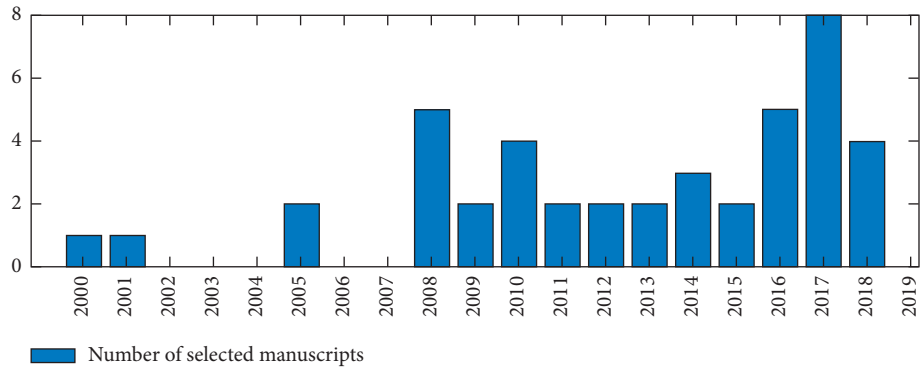

Figure 1: Distribution of papers per year, between 2000 and 2019.

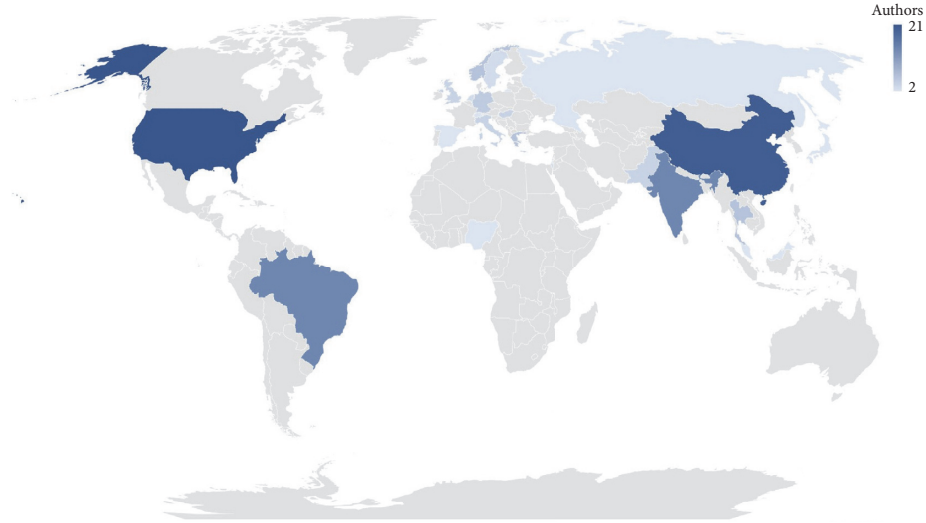

Figure 2: Number of authors per nationality of afference.

mentioned only once (possibly in the papers where they were originally defined). After applying this filter, only 43 metrics (the 24.7% of the original set of 174) remained. The boxplots in Figure 3 show, in green, the distributions of the total number of mentions and the measured score for this set of metrics. On these distributions, the rounded median value for the total number of mention is 3, and for the score is 3.

Since our final aim in answering RQ1.2 was to find a set of most popular metrics for the maintainability of source code, we resorted on selecting, on the complete set of 43 metrics mentioned in at least two papers, those whose score was above the median.

With this additional filtering, we obtained a set of 13 metrics and 2 metric suites, which are reported in Table 7. Two suites were included in their completeness (namely, the Chidamber and Kemerer suite and the Halstead suite) because all of their metrics had a number of total mentions and score higher or equal to the median. For them, the table reports the lower number of mentions and score among those of the contained metrics. Instead, for the Li and Henry suite, only the MPC (message passing coupling) metric obtained a number of mention and score above the median and hence was included in our set of selected most popular metrics. A brief description of the selected most popular metrics is reported in the following. The metrics are listed in alphabetical order:

(i) *CC (McCabe's Cyclomatic Complexity).* It is developed by McCabe in 1976 [56] and is a metric meant to calculate the complexity of code by examining the control flow graph of the program, i.e., counting its independent execution paths based on the flow graph [14]. The assumption is that the complexity of the code is correlated to the number of execution paths of its flow graph. It is also proved that there exists a linear correlation between the CC and the LOC metrics, as found by Jay and Hale. Such relationship is independent from the used programming language and code paradigms [57].

Each node in the flow graph corresponds to a block of code in the program where the flow is sequential; the arcs correspond to branches that can be taken by the control flow during the execution of the program. Based on those building blocks, the CC of a source code is defined as $M = e\ n + 2p$

Table 4: Selected studies.

ID	Authors	Title	Year	Score
[19]	K´ad´ar et al.	A code refactoring dataset and its assessment regarding software maintainability	2016	4
[6]	Sarwar et al.	A comparative study of MI tools: defining the roadmap to MI tools standardization	2008	4
[9]	Tahir and Ahmad	An AOP-based approach for collecting software maintainability dynamic metrics	2010	4
[20]	Gil et al.	An empirical investigation of changes in some software properties over time	2012	4
[21]	Jain et al.	An empirical investigation of evolutionary algorithm for software maintainability prediction	2016	4
[22]	Curtis et al.	An evaluation of the internal quality of business applications: does size matter?	2011	4
[23]	Chhillar and Gahlot	An evolution of software metrics: a review	2017	4
[24]	Tian et al.	AODE for source code metrics for improved software maintainability	2008	5
[25]	Kaur et al.	A proposed new model for maintainability index of open-source software	2014	4
[26]	Barbosa and Hirama	Assessment of software maintainability evolution using C&K metrics	2013	5
[27]	Misra et al.	A suite of object-oriented cognitive complexity metrics	2018	5
[28]	Rongviriyapanish et al.	Changeability prediction model for Java class based on multiple layer perceptron neural network	2016	4
[29]	Arshad and Tjortjis	Clustering software metric values extracted from C# code for maintainability assessment	2016	4
[30]	Pizka	Code normal forms	2005	4
[15]	Ludwig et al.	Compiling static software metrics for reliability and maintainability from GitHub repositories	2017	5
[31]	Mamun et al.	Correlations of software code metrics: an empirical study	2017	5
[32]	Alves et al.	Deriving metric thresholds from benchmark data	2010	4
[33]	Matsushita and Sasano	Detecting code clones with gaps by function applications	2017	4
[34]	Silva et al.	Detecting modularity flaws of evolving code: what the history can reveal?	2010	4
[16]	Liu et al.	Evaluate how cyclomatic complexity changes in the context of software evolution	2018	5
[35]	Ch´avez et al.	How does refactoring affect internal quality attributes? a multiproject study	2017	4
[36]	Ma et al.	How multiple-dependency structure of classes affects their functions: a statistical perspective	2010	4
[37]	Wahler et al.	Improving code maintainability: A case study on the impact of refactoring	2016	4
[38]	Kaur and Singh	Improving the quality of software by refactoring	2017	4
[39]	Yan et al.	Learning to aggregate: an automated aggregation method for software quality model	2017	4
[40]	Chatzidimitriou et al.	npm-miner: an infrastructure for measuring the quality of the npm registry	2018	4
[41]	Bohnet and ollner	Monitoring code quality and development activity by software maps	2011	4
[14]	Ostberg and Wagner	On automatically collectable metrics for software maintainability evaluation	2014	4
[42]	Narayanan Prasanth et al.	Prediction of maintainability using software complexity analysis: an extended FRT	2008	4
[43]	Wang et al.	Predicting object-oriented software maintainability using projection pursuit regression	2009	4
[44]	Sjøberg et al.	Questioning software maintenance metrics: a comparative case study	2012	5
[45]	Hindle et al.	Reading beside the lines: indentation as a proxy for complexity metric	2008	4
[46]	Lee and Chang	Reusability and maintainability metrics for object-oriented software	2000	4
[47]	Sinha et al.	Software complexity measurement using multiple criteria	2013	4
[5]	Kaur et al.	Software maintainability prediction by data mining of software code metrics	2014	5
[48]	Vytovtov and Markov	Source code quality classification based on software metrics	2017	4
[49]	Gold et al.	Spatial complexity metrics: an investigation of utility	2005	4
[50]	Ludwig et al.	Static software metrics for reliability and maintainability	2018	5
[51]	Saboe	The use of software quality metrics in the materiel release process experience report	2001	4
[52]	Yamashita et al.	Using concept mapping for maintainability assessments	2009	5
[53]	Threm et al.	Using normalized compression distance to measure the evolutionary stability of software systems	2015	4
[54]	Gon¸calves et al.	Using TDD for developing object-oriented software—A Case study	2015	4
[55]	Jermakovics et al.	Visualizing software evolution with Lagrein	2008	4

where n is the number of nodes of the graph, e is the number of edges of the graph, and p is the number of connected components, i.e., the number of exits from the program logic [6].

(ii) *CE (Efferent Coupling)*. It is a metric that measures how many data types the analyzed class utilizes, apart from itself. The metric takes into consideration the known type inheritance, the interfaces implemented by the class, the types of the parameters of its methods, the types of the declared attributes, and the types of the used exceptions.

(iii) *CHANGE (Number of Lines Changed in the Class)*. It is a change metric, which measures how many lines of code are changed between two versions of the same class of code. This metric is hence not defined on a single version of the software project, but it is tailored to analyze the evolution of the source code. The assumption between the usage of

Table 5: Metric studies (complete).

Metric suite	Metric	Papers using it	TOT	Score
—	Aggregate stability	[53]	1	1
—	AODE, aggregating one dependence estimators	[24]	1	1
Aspect-based metrics	DCP, degree of crosscutting per pointcut	[23]	1	1
Aspect-based metrics	NAA, number of advices per aspect	[23]	1	1
Aspect-based metrics	Number of aspects	[23]	1	1
Aspect-based metrics	NPA, number of pointcuts per aspect	[23]	1	1
Aspect-based metrics	RAD, response for advice	[23]	1	1
—	Avg CC, average cyclomatic complexity	[5, 54]	2	2
—	Avg. LOC per method	[54]	1	1
—	Bandwidth	[24]	1	1
—	Base classes (IFANIN)	[35]	1	1
—	Branching complexity (Sneed Metric)	[35]	1	1
—	Branch stability	[53]	1	1
—	Bug patterns	[14]	1	1
—	CA, afferent coupling	[5, 21]	2	2
—	CAM, cohesion among methods of a class	[21]	1	1
—	CBM, coupling between methods	[21]	1	1
—	CC, McCabe's cyclomatic complexity	[14, 24, 44, 51], [6, 16, 32, 44, 48], [23, 29, 31, 35, 47]	14	12
—	CE, efferent coupling	[5, 21, 25]	3	3
—	CHANGE, number changed in the class	[5, 25, 26, 43]	4	4
—	CHC, class coupling complexity	[46]	1	1
Chidamber and Kemerer	CBO, coupling between objects	[5, 14, 15, 25, 26], [21, 27, 32, 44], [20, 28, 36, 39], [23, 29, 35, 47, 50]	18	16
Chidamber and Kemerer	DIT, depth of inheritance tree	[5, 14, 26, 43, 44], [20, 21, 27, 32], [23, 28, 29, 39], [35, 47]	15	13
Chidamber and Kemerer	LCOM, lack of cohesion in methods	[5, 14, 26, 43, 44], [20, 27, 28, 32], [23, 36, 39, 46, 47]	14	12
Chidamber and Kemerer	NOC, number of children	[5, 14, 26, 43, 44], [20, 27, 32, 39], [23, 29, 35, 47]	13	11
Chidamber and Kemerer	RFC, response for class	[5, 15, 19, 26, 43], [14, 21, 27, 44], [20, 29, 32, 39], [23, 46, 47, 50]	17	15
Chidamber and Kemerer	WMC, weighted methods per class	[5, 21, 26, 43, 44], [16, 20, 27, 32], [23, 29, 39, 47]	13	11
—	CI, clone instances	[19]	1	1
—	CLOC, comment lines of code	[5, 24, 25, 28, 44], [35]	6	6
—	Clone coverage	[14]	1	1
—	Cocol's metric	[48]	1	1
Code smells	Feature Envy (# per KLOC of code)	[44]	1	1
Code smells	God Class (# per KLOC of code)	[44]	1	1
—	Code-to-comment ratio	[14, 15]	2	2
—	Complexity average by class	[31]	1	1
—	Complexity average by file	[31]	1	1
—	Complexity average by function	[31]	1	1
—	CONS, number of constructors	[5, 25]	3	1
—	Coupling and cohesion	[14]	1	1
—	Coupling dispersion	[35]	1	1
—	Coupling intensity	[35]	1	1
—	CPC, class coupling complexity	[46]	1	1
—	CSA, class size (attributes)	[5, 25]	2	2
—	CSO, class size (operations)	[5, 25]	2	2
—	CSOA, class size (operations + attributes)	[5, 25]	2	2
—	Cyclic, number of cyclic dependencies	[25]	1	1
—	Cyclomatic complexity in classes	[31]	1	1
—	Cyclomatic complexity in functions	[31]	1	1

Table 5: Continued.

Metric suite	Metric	Papers using it	TOT	Score
—	DAM, data access metric (Card Metric)	[21, 47]	2	2
—	Data complexity (Chapin Metric)	[35]	1	1
—	Dataflow	[14]	1	1
—	Dcy, number of dependencies	[5]	1	1
—	Dcy*, number of transitive dependencies	[5]	1	1
—	Decisional complexity (McClure)	[35]	1	1
—	Divergent change	[34]	1	1
—	Dominator tree metrics	[20]	1	1
—	Dpt, number of dependents	[5]	1	1
Dynamic metrics	DCBO, dynamic coupling between objects	[9]	1	1
Dynamic metrics	MTBF, mean time between failure	[9]	1	1
Dynamic metrics	MTTF, the mean time to failure	[9]	1	1
—	ECC, external class complexity (CIC + ICP)	[46]	1	1
—	Essential complexity	[35]	1	1
—	Fan-in	[35]	1	1
—	Fan-out	[35]	1	1
Halstead	Halstead bugs (B)	[5, 14, 23, 25, 48], [47]	6	4
Halstead	Halstead difficulty (D)	[5, 14, 25, 44, 51], [23, 47, 48]	8	6
Halstead	Halstead effort (E)	[5, 14, 25, 44, 51], [23, 47, 48]	8	6
Halstead	Halstead length (N)	[5, 14, 24, 25, 51], [23, 44, 47, 48]	9	7
Halstead	Halstead vocabulary (n)	[5, 14, 25, 44, 51], [23, 47, 48]	8	6
Halstead	Halstead volume (V)	[5, 14, 25, 44, 51], [6, 23, 44, 47, 48]	10	8
History sensitive metrics	pLOC	[40]	1	1
History sensitive metrics	rdocLOC	[40]	1	1
History sensitive metrics	rniLOC	[40]	1	1
History sensitive metrics	rpdLOC	[40]	1	1
History sensitive metrics	rpiLOC	[40]	1	1
History sensitive metrics	TL	[40]	1	1
—	I, instability	[28]	1	1
—	IC, inheritance coupling	[21]	1	1
—	ICC, internal class complexity (CAC + CMC)	[46]	1	1
—	Indentation as proxy for complexity metric	[44]	1	1
—	Inner*, number of inner classes	[5]	1	1
—	Jensen's Nf	[24]	1	1
—	JLOC, JavaDoc lines of code	[5, 25, 28]	3	3
—	Kaur's metric	[25]	1	1
—	LCOM2, lack of cohesion in methods	[21, 29, 35]	3	3
—	LCOM3, lack of cohesion of methods	[21, 35]	2	2
—	Level, level order	[5]	1	1
—	Level*, level order	[5]	1	1
Li and Henry (L&H)	DAC	[39, 43]	2	2
Li and Henry (L&H)	DIT, depth of inheritance tree	[25]	1	1
Li and Henry (L&H)	LCOM, lack of cohesion in methods	[25]	1	1
Li and Henry (L&H)	MPC, message passing coupling	[25, 39, 43, 46]	4	4
Li and Henry (L&H)	NOC, number of children	[25]	1	1
Li and Henry (L&H)	NOM	[39]	1	1
Li and Henry (L&H)	RFC, response for a class	[25]	1	1

Table 5: Continued.

Metric suite	Metric	Papers using it	TOT	Score
Li and Henry (L&H)	SIZE2	[39]	1	1
Li and Henry (L&H)	SLOC, source lines of code	[25]	1	1
Li and Henry (L&H)	WMC, weighted method per class	[25]	1	1
—	LOC, lines of code	[15, 24–26], [14, 44, 51], [6, 21, 23, 31, 32], [35, 50]	15	11
—	Logic	[14]	1	1
—	MC, method coupling	[46]	1	1
—	MFA, measure of functional abstraction	[21]	1	1
—	MI, maintainability index	[6, 14, 25, 44, 51], [30]	6	4
Misra's metrics	AAC, average attributes per class	[27]	1	1
Misra's metrics	AC, attribute complexity	[27]	1	1
Misra's metrics	ACC, average class complexity	[27]	1	1
Misra's metrics	ACF, average coupling factor	[27]	1	1
Misra's metrics	AMC, average method complexity	[27]	1	1
Misra's metrics	AMCC, average method complexity per class	[27]	1	1
Misra's metrics	CLC, class complexity	[27]	1	1
Misra's metrics	CC, code complexity	[27]	1	1
Misra's metrics	CWC, coupling weight for a class	[27]	1	1
Misra's metrics	MC, method complexity	[27]	1	1
Misra's metrics	OMMIC, coupling	[44]	1	1
—	MOA, measure of aggregation	[21, 28]	2	2
Mood's metrics	AHF, attribute hiding factor	[23]	1	1
Mood's metrics	AIF, attribute inheritance factor	[23]	1	1
Mood's metrics	CF, coupling factor	[23]	1	1
Mood's metrics	MHF, method hiding factor	[23]	1	1
Mood's metrics	MIF, method inheritance factor	[23]	1	1
Mood's metrics	PF, polymorphism factor	[23]	1	1
—	NAA, number of attributes added	[25]	1	1
Narsimhan's metrics	AID, average interaction density	[23]	1	1
Narsimhan's metrics	IID, incoming interaction density	[23]	1	1
Narsimhan's metrics	OID, outgoing interaction density	[23]	1	1
—	Nesting depth	[14]	1	1
—	Nesting (Max Nest)	[35]	1	1
—	NIM, instance methods	[35]	1	1
—	NIV, instance variables	[35]	1	1
—	NOAC, number of operations added	[5, 25]	2	2
—	NOI, number of outgoing invocations	[19]	1	1
—	NOOC, number of operations overridden	[5]	1	1
—	NOM, number of methods	[26, 28, 31, 43]	4	4
—	Noncommenting lines of code (lines only containing space, tab, and CR are ignored)	[31]	1	1
—	Noncommenting lines of new code	[31]	1	1
—	NOP, number of polymorphic methods	[28]	1	1
—	NOPA, number of public attributes	[35]	1	1
—	NPATH	[32]	1	1
—	NPM, number of public methods	[5, 21, 28, 29]	4	4
—	Number of attributes added	[5]	1	1
—	Number of code characters	[24]	1	1
—	Number of classes (including nested classes, interfaces, enums, and annotations)	[31, 35]	2	2
—	Number of commands	[5, 25]	2	2
—	Number of comment characters	[24]	1	1
—	Number of directories	[31]	1	1
—	Number of files	[31, 33]	2	2

Table 5: Continued.

Metric suite	Metric	Papers using it	TOT	Score
—	Number of God Classes	[34]	1	1
—	Number of queries	[5, 25]	2	2
—	Ocmax, maximum operation complexity	[25]	1	1
—	OSmax, maximum operation size	[25]	1	1
—	Override ratio	[35]	1	1
—	Paths	[35]	1	1
—	PDcy, number of package dependencies	[5, 25]	2	2
—	RAM, RAM + CPU memory usage	[48]	1	1
—	RCI, ratio of cohesion interactions	[46]	1	1
—	Shotgun Surgery	[34]	1	1
SM, structural measures	OMMIC, coupling	[44]	1	1
SM, structural Measures	TCC, tight class cohesion	[35, 44]	2	2
SM, structural Measures	WMC1, size of classes	[44]	1	1
—	Structure stability	[53]	1	1
—	Spatial complexity metrics	[49]	1	1
—	STAT, number of statements	[5, 25, 31, 35]	4	4
—	TCLOC, total comment lines of code	[19]	1	1
—	Test results and coverage	[14]	1	1
—	TLLOC, total logical lines of code	[19]	1	1
—	TNOS, total number of statements	[19]	1	1
—	Token count	[23]	1	1
—	Total number of characters	[24]	1	1
—	Version distance	[53]	1	1
—	Version stability	[53]	1	1
—	Vytovtov's metric	[48]	1	1
—	Welker and Oman	[25]	1	1
—	WMC, McCabe's weighted method count	[14, 15, 25, 38], [28, 36, 50]	7	7
—	WMCU, McCabe's weighted method count-unweighted	[15, 50]	2	2
—	WOC, weight of classes	[35]	1	1

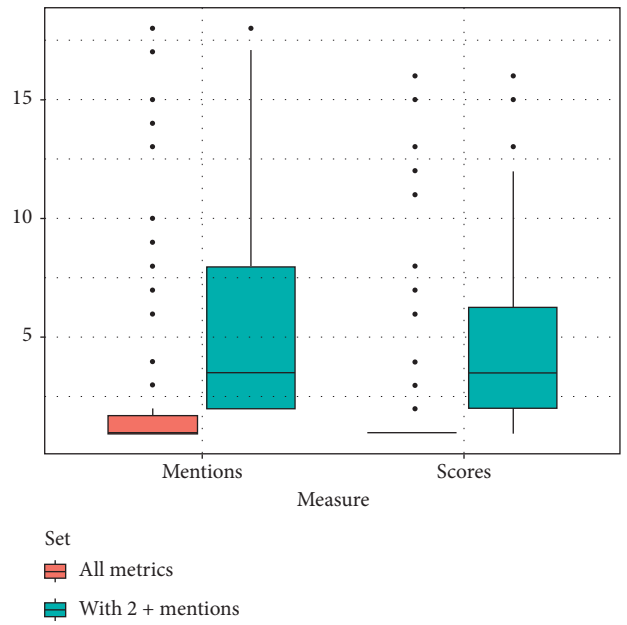

Figure 3: Distributions of total number of mentions and the measured score for the metrics.

this metric is that if a class is continuously modified, it can be a sign that it is hardly maintainable.

Generally, three types of changes *can* be made to a line of code: additions, deletions, or modifications. In the literature, there is typically accordance about how to count the operations of modifications, which typically counts two times as the additions or deletions (the modification is considered as a deletion followed by an addition). Most of the times, comments, and blanks are not considered in the computation of the changed LOCs during the evolution of software code.

(iv) *C&K (Chidamber and Kemerer Suite)*. It is one of the best-known sets of metrics, which was introduced in 1994 [58]. This suite has been designed keeping into consideration the object-oriented approach. It is composed of 6 metrics, listed as follows:

WMC, weighted method per class, defined in the same way as McCabe's WMC (weighted method count, described below) but applied to a class, i.e., it gives the complexity of that particular class by

Table 6: Metrics found in the selected set of primary studies, with the number of mentions and score higher or equal to 2.

Metric suite	Metric	Mentioned by	TOT	Score
—	Avg CC, average cyclomatic complexity	[5, 54]	2	2
—	CA, afferent coupling	[5, 21]	2	2
—	CC, McCabe's cyclomatic complexity	[14, 24, 51], S13, [6, 16, 32, 45, 48], [23, 29, 31, 35, 47]	14	12
—	CE, efferent coupling	[5, 21, 25]	3	3
—	CHANGE, number of lines changed in the class	[5, 25, 26, 43]	4	4
Chidamber and Kemerer	CBO, coupling between objects	[5, 14, 15, 21, 25–27, 32, 44], [20, 23, 28, 29, 35, 36, 39, 47, 50]	18	16
Chidamber and Kemerer	DIT, depth of inheritance tree	[5, 14, 26, 43], S13, [20, 21, 27, 32], [23, 28, 29, 35, 39, 47]	15	13
Chidamber and Kemerer	LCOM, lack of cohesion in methods	[5, 14, 26, 43], S13, [20, 27, 28, 32], [23, 36, 39, 46, 47]	14	12
Chidamber and Kemerer	NOC, number of children	[5, 14, 26, 43], S13, [20, 27, 32, 39], [23, 29, 35, 47]	13	11
Chidamber and Kemerer	RFC, response for class	[5, 14, 15, 19, 21, 26, 27, 43, 44], [20, 23, 29, 32, 39, 46, 47, 50]	17	15
Chidamber and Kemerer	WMC, weighted methods per class	[5, 26, 43], S13, [16, 20, 21, 27, 32], [23, 29, 39, 47]	13	11
—	CLOC, comment lines of code	[5, 24, 25], S13, [28, 35]	6	6
—	Code-to-comment ratio	[14, 15]	2	2
—	CSA, class size (attributes)	[5, 25]	2	2
—	CSO, class size (operations)	[5, 25]	2	2
—	CSOA, class size (operations+attributes)	[5, 25]	2	2
—	DAM, data access metric (card metric)	[21, 47]	2	2
Halstead	Halstead bugs (B)	[5, 14, 23, 25, 47, 48]	6	4
Halstead	Halstead difficulty (D)	[5, 14, 23, 25, 45, 47, 48, 51]	8	6
Halstead	Halstead effort (E)	[5, 14, 23, 25, 45, 47, 48, 51]	8	6
Halstead	Halstead length (N)	[5, 14, 23–25, 45, 47, 48, 51]	9	7
Halstead	Halstead vocabulary (n)	[5, 14, 23, 25, 45, 47, 48, 51]	8	6
Halstead	Halstead volume (V)	[5, 14, 25, 51], S13, [6, 23, 45, 48], [47]	10	8
—	JLOC, JavaDoc lines of code	[5, 25, 28]	3	3
—	LCOM2, lack of cohesion of methods	[21, 29, 35]	3	3
—	LCOM3, lack of cohesion of methods	[21, 35]	2	2
Li and Henry (L&H)	DAC	[39, 43]	2	2
Li and Henry (L&H)	MPC, message passing coupling	[25, 39, 43, 46]	4	4
—	LOC, lines of code	[14, 15, 21, 24–26, 44, 51], [6, 23, 31, 32, 35, 50]	15	11
—	MI, maintainability index	[6, 14, 25, 30, 44, 51]	6	4
—	MOA, measure of aggregation	[21, 28]	2	2
—	NOAC, number of operations added	[5, 25]	2	2
—	NOM, number of methods	[26, 28, 31, 43]	4	4
—	NPM, number of public methods	[5, 21, 28, 29]	4	4
—	Number of classes (including nested classes, interfaces, enums, and annotations)	[31, 35]	2	2
—	Number of commands	[5, 25]	2	2
—	Number of files	[31, 33]	2	2
—	Number of queries	[5, 25]	2	2
—	PDcy, number of package dependencies	[5, 25]	2	2
SM, structural measures	TCC, tight class cohesion	S13, [35]	2	2
—	STAT, number of statements	[5, 25, 31, 35]	4	4
—	WMC, McCabe's weighted method count	[14, 15, 25, 28, 36, 38, 50]	7	7
—	WMCU, McCabe's weighted method count-unweighted	[15, 50]	2	2

adding together the CC of all the methods within that same class [58].

DIT, depth of inheritance tree, defined as the length of the maximal path from the leaf node to the root of the inheritance tree of the classes of the analyzed software.

Inheritance helps to reuse the code; therefore, it increases the maintainability. The side effect of

Table 7: Metrics (suites) with citation count and score above the median.

Metric	Total mentions	Score
CC, McCabe's cyclomatic complexity	14	12
CE, efferent coupling	3	3
CHANGE, number of lines changed in class	4	4
C&K, Chidamber and Kemerer suite	13+	11+
CLOC, comment lines of code	6	6
Halstead's suite	6+	4+
JLOC, JavaDoc lines of code	3	3
LOC, lines of code	14	11
LCOM2, lack of cohesion in methods	3	3
MI, maintainability index	6	4
MPC, message passing coupling	4	4
NOM, number of methods	4	4
NPM, number of public methods	4	4
STAT, number of statements	4	4
WMC, McCabe's weighted method count	7	7

inheritance is that classes deeper within the hierarchy tend to have increasingly complex behaviour, making them difficult to maintain.
Having one, two, or even three levels of inheritance can help the maintainability, but increasing the value further is deemed detrimental.
NOC, number of children, is the number of immediate subclasses of the analyzed class. As the NOC increases, maintainability of the code increases.
CBO, coupling between objects, is the number of classes with which the analyzed class is coupled. Two classes are considered coupled when methods declared in one class use methods or instance variables defined by the other class. Thus, this metric gives us an idea on how much interlaced the classes are to each other and hence how much influence the maintenance of a single class has on other ones.
RFC, response for class, is defined as the set of methods that can potentially be executed in response to a message received by an object of that class. Also, in this case, the greater is the returned value, the greater is the complexity of the class.
LCOM, lack of cohesion in methods, is defined as the subtraction between the number of method pairs having no attributes in common, and the number of method pairs having common attributes. Several other versions of the metrics have been provided in the literature. High values of LCOM metric value provide a measure of the relative disparate nature of methods in the class.

(v) *CLOC (Comment Line of Code)*. It is the metric which gives the number of lines of code which contain textual comments. Empty lines of comments are not counted. In contrast to the LOC metric, the higher the value CLOC returns, the more the comments there are in the analyzed code; therefore, the code should be easier to understand and to maintain.

The literature has also proposed a metric that puts in relation between CLOC and LOC, and it is called the code-to-comment ratio.

(vi) *The Halstead Suite*. It is introduced in 1977 [59] and is a set of statically computed metrics, which tries to assess the efforts required to maintain the analyzed code, the quality of the program, and the number of errors in the implementation.

To compute the metrics of the Halstead suite, the following indicators must be computed from the source code: n_1, i.e., the number of distinct operators; n_2, i.e., the number of distinct operands; N_1, i.e., the total number of operators; and N_2, i.e., the total number of operands. Operands are the objects that are manipulated, and operators are all the symbols that represent specific actions.

Operators and operands are the two types of components that form all the expressions. The following metrics are part of the Halstead suite:

Length (N): $N = N_1 + N_2$, i.e.,
where N_1 is the total number of occurrences of operators and N_2 is *the* total number of occurrences of operands.
Vocabulary (n): $n = n_1 + n_2$, i.e., where n_1 is the total number of distinct operators and n_2 is the number of distinct operands in the program. By definition, the Vocabulary constitutes a lower bound for the Length, since each distinct operator and operand has at least an occurrence.
Volume (V): $V = N \log_2 n$, i.e., the size, in bits, of the space used to store the program (note that this varies according to the specific implementation of the program).
Difficulty (D): $D = n_1/2 \cdot N_2/n_2$, which represents the difficulty to understand the code.
Effort (E): $E = D\ V$, which represents effort necessary to understand a class.
Bugs (B): $B = E^{(2/3)}/3000$, which tries to give an esteem of the number of bugs present during the implementation of the code.
Time (T): $T = E/18$, which gives an esteem of the time needed to implement that code.

(vii) *JLOC, (JavaDoc Lines of Code)*. It is a metric specific for Java code, which is defined as the number of lines of code to which JavaDoc comments are associated. It is similar to other metrics discussed in the literature that measure the number of comments in the source code. In general, a high value for the JLOC metrics is deemed positive, since it suggests better documentation of the code and hence a better changeability and maintainability. This metric is specific to the Java programming language. Similar

Table 8: All tools found in the selected set of primary studies.

Tool name	Papers using it	Source
Columbus quality model	[15]	Not found
Analyst4j	[6]	Not found
Lachnesis	[6]	Not found
Metrics	[6]	Not found
CCEvaluator	[16]	Not found
Lagrein	[55]	Not found
Code Crawler	[55]	Not clear what program they used
RAMOOS, reconfigurable automated metrics for object-oriented software	[46]	Reconfigurable
Baker (Baker, 1993)	[33, 41]	Not found/not clear what program they used
Kamiya (Kamiya et al., 2002)	[33]	Not found/not clear what program they used
Li (Li and Thompson, 2010)	[33]	Not found/not clear what program they used
Baxter (Baxter et al., 1998)	[33]	Not found/not clear what program they used
Brown (Brown and Thompson, 2010)	[33]	Not found/not clear what program they used
Koschke (Koschke et al., 2006)	[33]	Not found/not clear what program they used
Higo (Higo and Kusumoto, 2009)	[33]	Not found/not clear what program they used
Mayrand (Mayrand et al., 1996)	[33]	Not found/not clear what program they used
Elva (Elva and Leavens, 2012)	[33]	Not found/not clear what program they used
Murakami (Murakami et al., 2014)	[33]	Not found/not clear what program they used
Higo (Higo et al., 2007)	[33]	Not found/not clear what program they used
Closed-source tools		
Codacy	[15]	https://www.codacy.com
Visual Studio	[48]	https://docs.microsoft.com/en-us/visualstudio/code-quality/code-metrics-values
Understand	[6, 50]	https://scitools.com/feature/metrics
JHawk	[6]	http://www.virtualmachinery.com/jhawkprod.htm
CMT++/CMTJava	[6]	https://www.verifysoft.com/en cmtx.html
CAST's Application Intelligence Platform	[22]	https://www.castsoftware.com/products/application-intelligence-platform
Open-source tools		
CKJM	[5, 21, 25, 26, 43]	https://www.spinellis.gr/sw/ckjm
MetricsReloaded (IntelliJ IDEA plugin)	[5]	https://github.com/BasLeijdekkers/MetricsReloaded
CodeMetrics (IntelliJ IDEA plugin)	[5]	https://github.com/kisstkondoros/codemetrics-idea
Ref-Finder (Eclipse plugin)	[19, 38]	https://sites.google.com/site/reffindertool
Squale	[15]	http://www.squale.org
Quamoco Benchmark for software quality	[15]	https://github.com/wagnerst/quamoco
CBR Insight	[15]	https://github.com/StottlerHenkeAssociates
Halstead Metrics Tool	[38]	https://sourceforge.net/p/halsteadmetricstool
SonarQube and CodeAnalyzers, by SonarSource	[6, 15, 31]	https://www.sonarsource.com
JSInspect	[40]	https://www.npmjs.com/package/jsinspect
Escomplex	[40]	https://github.com/escomplex/escomplex
Eslint	[40]	https://eslint.org
CCFinder (code clones finder), now called CCFinderX	[33]	http://www.ccfinder.net/ccfinderxos.html

documentation generators are available for JavaScript (JSDoc) and PHP (PHPDocumentor); however, we were not able to gather evidence from the manuscripts about the applicability of the JLOC metric to them, so we deemed it applicable only for source code written in Java.

(viii) *LOC (Lines of Code)*. It is a widely used metric which is often used for its simplicity. It gives an immediate measure of the size of the source code. Among the most popular metrics, the LOC metric was the only one to have two negative mentions in other works in the literature. These comments are related to the fact that there appears to be no single, universally adopted definition of how this metric is computed [14]. Some works consider the count of all the lines in a file, and others (the majority) remove blank lines from such computation; if there is more than one instruction in a single line or a single instruction is divided into different rows, there is ambiguity about considering the number of lines (physical lines) or the actual number of instructions involved (logical lines). Thus, it is of the utmost importance that the tools to calculate the metrics specify exactly how they calculate the values they return (or that they are open source, hence allowing an analysis of the tool source code for deriving such information).

Although LOC seems to be poorly related to the maintenance effort [14] and there is more than one way to calculate it, this metric is used within the

maintainability index, and it seems to be correlated with many of different metric measures [60]. The assumption is that the bigger the LOC metric, the less maintainable the analyzed code is.

(ix) *LCOM2 (Lack of Cohesion in Methods)*. It is an evolution of the LCOM metric, which is part of the Chidamber and Kemerer suite. LCOM2 equals the percentage of methods that do not access a specific attribute averaged over all attributes in the class. If the number of methods or attributes is zero, LCOM2 is undefined and displayed as zero. A low value of LCOM2 indicates high cohesion and a well-designed class.

(x) *MI (Maintainability Index)*. It is a composite metric, proposed as a way to assess the maintainability of a software system. There are different definitions of this metric, which was firstly introduced by Oman and Hagemeister in 1992 [61]. There are two different formulae to calculate the MI, one utilizing only three different metrics, Halstead volume (HV), cyclomatic complexity (CC), and the number of lines of code (LOC), while the other takes in consideration also the number of comments. Despite being quite popular, Ostberg and Wagner express their doubts about the effectiveness of this metric, claiming it does not give information about the maintainability of the code, since it is based on metrics considered not suited for that task, and the result of the metric itself is not intuitive [14]. In contrast, Sarwar et al. state that MI proved to be very efficient in improving software maintainability and cost-effectiveness [6].

The *3-metric* equation is as follows: $\mathrm{MI} = 171 - 5.2 \cdot \ln(avgV) - 0.23 \cdot avgCC - 16.2 \cdot \ln(avgLOC)$.

The *4-metric* equation is as follows: $\mathrm{MI}_{\sqrt{}} = 171 - 5.2 \ln(avgV) - 0.23\ avgCC + -16.2\ \ln(avgLOC) + 50 \sin(2.4\ perCM)$.

In both equations, the following symbols are adopted: *avgV* is the average Halstead volume for the source code files; *avgLOC* is the average LOC metric; *avgCC* is the average cyclomatic complexity; *perCM* is the percentage of LOC containing comments.

A returned value above 85 means that the code is easily maintainable; a value from 85 to 65 indicates that the code is not so easy to maintain; below 65, the code is difficult to maintain. The returned value can reach zero, and even become negative, especially for large projects.

(xi) *MPC (Message Passing Coupling)*. It is a metric from the Li and Henry suite (the only metric of that suite to have a score above the rounded median), and it is defined as the number of send statements defined in a class [62], i.e., the number of method calls in a class.

(xii) *NOM (Number of Methods Counts)*. It is the number of methods in a given class/source file, with the assumption that the higher the number of methods, the lower the maintainability of the code.

(xiii) *NPM (Number of Public Methods)*. It returns the number of all the methods in a class that are declared as public.

(xiv) *STAT (Number of Statements)*. It counts the number of statements in a method. Different variations of the metric have been proposed in the literature, which differ on the decision of counting statements also in named inner classes, interfaces, and anonymous inner classes. For instance, Kaur et al., in their study for software maintainability prediction, count the number of statements only in anonymous inner classes [5].

(xv) *WMC (McCabe's Weighted Method Count)*. It is a measure of complexity that sums the complexity of all the methods implemented in the analyzed code. The complexity of each method is calculated using McCabe's cyclomatic complexity, which is also present among the most cited metrics and discussed above. A simplified variant of this metric, called WMC-unweighted, simply counts each method as if it had unitary complexity; this variant corresponds to the NOM (number of methods) metric.

3.3. RQ2.1: Available Tools. In Table 8, we report all the tools that were identified while reading the papers. The columns report, respectively, as follows: the name of the tool, as it is presented in the studies; the studies using it; a web source where the tool can be downloaded. In the upmost section of the table, we reported papers from which we cannot find the used tool (i.e., a tool was mentioned but no download pointer was provided, indicating that the tool has never been made public and/or it had been discontinued), or for which no information about the used tool was provided. For the latter, we have indicated the studies in the table with the respective author's name.

In the second and third section of the table, we have divided the tools according to their release nature, i.e., we discriminated between open-source and commercial tools. The table reports information about a total of 38 tools: 19 were not found; 6 were closed source; and 13 were open source.

The majority of the tools we found are mentioned by only one study; three are cited by two studies, and only one, CKJM, is quoted by five papers.

It is immediately evident that the open-source tools are more than two times in number than the closed-source ones. This result may be unrelated to the quality of the tools themselves but instead be justified by the fact that open-source tools are better suited for academic usage since they provide the possibility of checking the algorithms and possibly modify or integrate them to analyze their performance.

	CAST's AIP	Codacy	CMT++/CMTJava	Jhawk	Understand	Visual studio	CKJM	MetricsReloaded	CodeMetrics	Squale	Quamoco benchmark	CBR insight	Halstead metrics tool	SonarQube - codeAnalyzer	Jsinspect	Escomplex	Eslint	CCFinderX	Ref-finder tool
Ada					X							X							
APAB														X					
Apex		X			X							X		X					
Assembly					X							X							
C	X	X	X		X	X		X		X		X	X	X				X	
C++	X	X	X		X	X				X		X	X	X				X	
C#	X	X	X		X							X		X				X	
Cobol	X				X					X		X		X				X	
CSS														X					
Fortran	X				X							X							
GO														X					
HTML														X					
Java	X	X	X	X	X		X	X	X		X	X	X	X				X	X
JavaScript	X	X			X							X		X	X	X	X		
Jovial					X							X							
Json		X			X							X							
JSP	X	X			X							X							
Kotlin		X			X							X		X					
Markdown		X			X							X							
Objective C														X					
PHP	X	X			X							X		X					
Python	X	X			X							X		X					
RPG														X					
Ruby		X			X							X		X					
Scala		X			X							X		X					
SQL														X					
Switf														X					
T-SQL														X					
TypeScript														X					
VB6	X	X			X							X							
VB.NET	X	X			X							X							
Velocity		X			X							X							
VisualForce		X			X							X							
XML		X			X							X		X					

Figure 4: Programming languages supported by each tool.

For each of the tools that we were able to identify, we give a brief description in the following; the details about their supported languages and metrics can be found after the descriptions of the tools.

3.3.1. Closed-Source Tools. Six closed-source tools can be found in the analyzed primary studies, three of which are mentioned in the same paper. The tools described hereafter are listed in alphabetical order and not in any order of importance.

(i) *CAST's Application Intelligence Platform*. This tool analyzes all the source code of an application, to measure a set of nonfunctional properties such as performance, robustness, security, transferability, and changeability (which is strictly tied to maintainability). This last nonfunctional property is measured based on cyclomatic complexity, coupling, duplicated code, and modification of indexes in groups [63]. The tool produces as output a set of violation of typical architectural and design patterns and best practices, which are aggregated in formats specific for both the management and the developers.

(ii) *CMT++/CMTJava*. CMT is a tool specifically made to estimate the overall maintainability of code done in C, C++, C#, or Java, and to identify the less maintainable parts of it. It is possible to compute many of the discussed metrics with the tool: McCabe's cyclomatic number, Halstead's software science metrics, lines of code, and others. CMT also allows computing the maintainability index (MI). The tool can work in command line mode or with a GUI.

(iii) *Codacy*. It is a free tool for open-source projects and can be self-hosted, otherwise a license must be purchased to use it. This tool aims at improving the code quality, to augment the code coverage and to prevent security issues. Its main focus is on identifying bugs and undefined behaviours rather than calculating metrics. It provides a set of statistics about the analyzed code: error-proneness, code style, code complexity, unused code, and security.

(iv) *JHawk*. The tool is tailored to only analyze code written in Java, but it can calculate a vast variety of different metrics. JHawk is not new on the market since its first release was introduced more than ten years ago. At the time of writing this article, the last

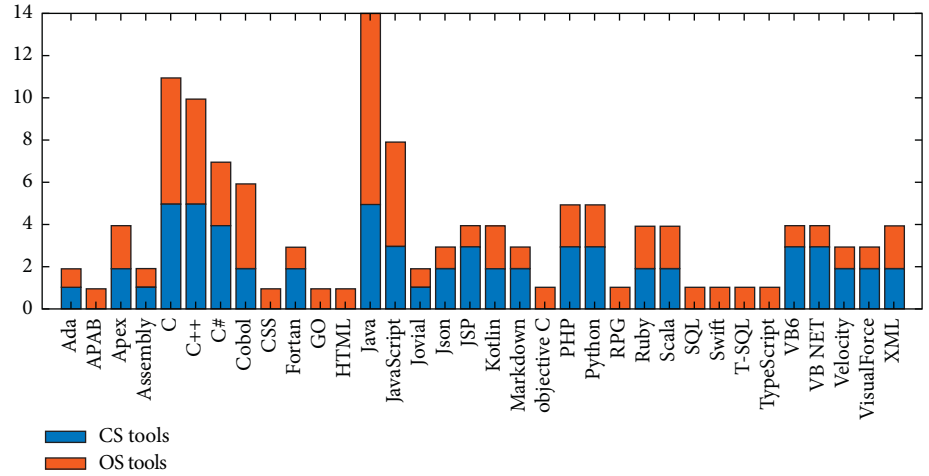

Figure 5: Number of tools (closed or open source) per language.

	CAST's AIP	Codacy	CMT++/CMTJava	Jhawk	Understand	Visual studio	CKJM	MetricsReloaded	CodeMetrics	Squale	Quamoco benchmark	CBR insight	Halstead metrics tool	SonarQube-codeAnalyzer	Jsinspect	Escomplex	Eslint	CCFinderX	Ref-finder tool
CC	X	X	X	X	X	X		X	X	X				X		X	X	X	
CE	X			X	X	X		X		X									
CHANGE																		X	X
C&K					X	X	X	X											
CLOC		X		X	X							X		X			X		
Halstead's			X	X									X			X			
JLOC		X						X		X									
LOC	X	X	X	X	X	X		X		X	X	X		X	X		X	X	
LCOM2	X				X														
MI			X	X		X								X		X			
MPC	X			X															
NOM					X					X				X					
NPM					X		X			X				X					
STAT				X	X									X			X		
WMC					X		X					X							
AvgCC					X														
CA							X	X		X									
CBO							X					X							
Code Smells														X	X		X	X	X
Essential Complexity					X			X											
MND			X	X	X												X		
Mood's								X											
NIV				X	X														
NOC							X												
N. Classes								X		X				X			X		
N. Commands				X															
N. Directories														X					
N. Files					X									X					
N. Queries	X																		
RFC				X			X					X							
Test Results								X		X				X					
Others	X			X	X			X		X	X	X		X	X	X	X	X	X

Figure 6: Tool support to the metrics found in primary studies.

available version is 6.1.3, from 2017. It is used and cited in more than twenty of the selected primary studies. JHawk aids the empirical evaluation of software metrics with the possibility of reporting the computed measures in various formats, including XML and CSV, and it supports a CLI interface.

(v) *Understand.* Developed by SciTools, it can calculate several metrics, and the results can be extracted automatically via command line, graphical interface, or through their AIP. Most of the metrics supported by this program are complexity metrics (e.g., McCabe's CC), volume metrics (e.g., LOC),

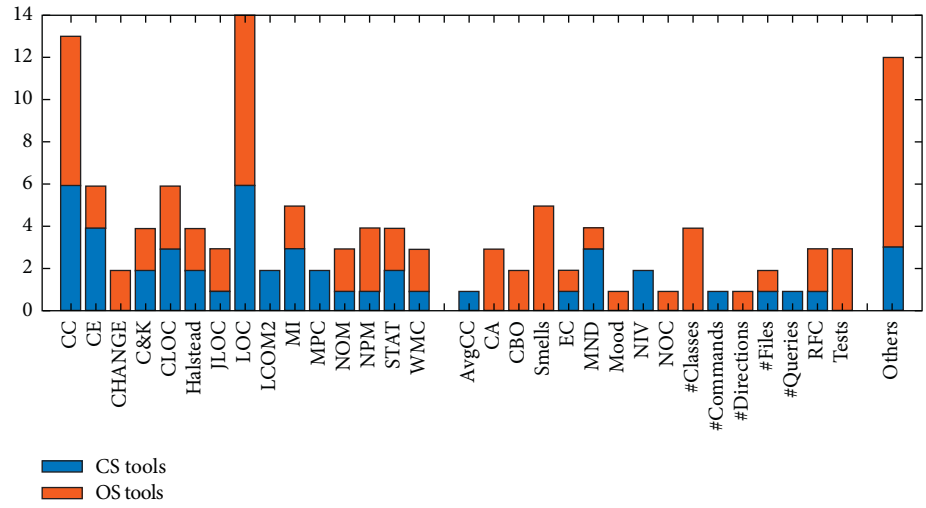

Figure 7: Number of tools (closed or open source) per metric.

Table 9: Tools available for computing the most popular metrics for the most supported programming languages (open-source tools in bold).

Metric	C	C++	C#	Java	JavaScript
CC	CAST'S AIP Codacy CMT++ Understand Visual Studio **MetricsReloaded** **Squale** **CodeAnalyzers** **CCFinderX**	CAST'S AIP Codacy CMT++ Understand Visual Studio **Squale** **CodeAnalyzers** **CCFinderX**	CAST'S AIP Codacy CMT++ Understand **CodeAnalyzers** **CCFinderX**	CAST'S AIP Codacy CMTJava JHawk Understand **MetricsReloaded** **CodeMetrics** **CodeAnalyzers** **CCFinderX**	CAST'S AIP Codacy Understand **CodeAnalyzers** **Escomplex** **Eslint**
CE	CAST'S AIP Understand Visual Studio **MetricsReloaded** **Squale**	CAST'S AIP Understand Visual Studio **Squale**	CAST'S AIP Understand	CAST'S AIP JHawk Understand **MetricsReloaded**	CAST'S AIP Understand
CHANGE	**CCFinderX**	**CCFinderX**	**CCFinderX**	**CCFinderX** **Ref-Finder**	
C&K	Understand Visual Studio **MetricsReloaded**	Understand Visual Studio	Understand	Understand **CKJM** **MetricsReloaded**	Understand
CLOC	Codacy Understand **CBR Insight** **CodeAnalyzers**	Codacy Understand **CBR Insight** **CodeAnalyzers**	Codacy Understand **CBR Insight** **CodeAnalyzers**	Codacy JHawk Understand **CBR Insight** **CodeAnalyzers**	Codacy Understand **CBR Insight** **CodeAnalyzers** **Eslint**
Halstead's	CMT++ **Halstead Metrics Tool**	CMT++ **Halstead Metrics Tool**	CMT++	CMTJava JHawk **Halstead Metrics Tool**	Escomplex
JLOC	—	—	—	Codacy **MetricsReloaded** **Squale**	—

Table 9: Continued.

Metric	C	C++	C#	Java	JavaScript
LOC	CAST'S AIP Codacy CMT++ Understand Visual Studio **MetricsReloaded** **Squale** **CBR Insight** **CodeAnalyzers** **CCFinderX**	CAST'S AIP Codacy CMT++ Understand Visual Studio **Squale** **CBR Insight** **CodeAnalyzers** **CCFinderX**	CAST'S AIP Codacy CMT++ Understand **CBR Insight** **CodeAnalyzers** **CCFinderX**	CAST'S AIP Codacy CMTJava JHawk Understand **MetricsReloaded** **Quamoco Benchmark** **CBR Insight** **CodeAnalyzers** **CCFinderX**	CAST'S AIP Codacy Understand **CBR Insight** **CodeAnalyzers** **JSInspect** **Eslint**
LCOM2	CAST'S AIP Understand	CAST'S AIP Understand	CAST'S AIP Understand	CAST'S AIP Understand	CAST'S AIP Understand
MI	CMT++ Visual Studio **CodeAnalyzers**	CMT++ Visual Studio **CodeAnalyzers**	CMT++ **CodeAnalyzers**	CMTJava JHawk **CodeAnalyzers**	**CodeAnalyzers** **Eslint**
MPC	CAST'S AIP	CAST'S AIP	CAST'S AIP	CAST'S AIP JHawk	CAST'S AIP
NOM	Understand **Squale** **CodeAnalyzers**	Understand **Squale** **CodeAnalyzers**	Understand **CodeAnalyzers**	Understand **CodeAnalyzers**	Understand **CodeAnalyzers**
NPM	Understand **Squale** **CodeAnalyzers**	Understand **Squale** **CodeAnalyzers**	Understand **CodeAnalyzers**	Understand **CKJM** **CodeAnalyzers**	Understand **CodeAnalyzers**
STAT	Understand **CodeAnalyzers**	Understand **CodeAnalyzers**	Understand **CodeAnalyzers**	JHawk Understand **CodeAnalyzers**	Understand **CodeAnalyzers** **Eslint**
WMC	Understand **CBR Insight**	Understand **CBR Insight**	Understand **CBR Insight**	Understand **CKJM** **CBR Insight**	Understand **CBR Insight**

and object-oriented metrics. The correlation between the supported metrics and the inferred maintainability of software projects is not explicitly mentioned in the tool's documentation.

(v) *Visual Studio.* It is a very well-known IDE developed by Microsoft. It comes embedded with modules for the computation of code quality metrics, in addition to all its other functions. Among the maintainability metrics listed in the previous section, it supports MI, CC, DIT, class coupling, and LOC. The main limitation for the Visual Studio tool is that these metrics can be computed only for projects written in the C and C++ languages, and not for projects in any other of the many languages supported by the IDE. Also, from the Visual Studio documentation, it can be seen that the IDE makes some assumptions about the metrics that are different from the standard ones. As an example, the MI metric used in Visual Studio is an integer between 0 and 100, with different thresholds from the standard ones defined for MI (MI 20 indicates a code easy to maintain, a rating from 10 to 19 indicates that the code is relatively maintainable, and a value below 10 indicates low maintainability).

3.3.2. Open-Source Tools. Fourteen open-source tools could be found in the analyzed primary studies. Most of them, however, require a license to be used in not open-source projects or to be used without limitations. The tools described hereafter are listed in alphabetical order and not in any order of importance:

(i) *CBR Insight.* It is a tool built on top of Understand (see the previous section about closed-source tools), and it uses it to calculate the metrics. The tool calculates metrics that are highly related to software reliability, maintainability, and preventable technical debt. It provides a dashboard to present the data to developers/maintainers. It is worth noting that the tool, although open source, needs a license for the Understand tool to be used.

(ii) *CCFinderX (Code Clones Finder).* Previously known as CCFinder, it is a tool able to detect duplicate code fragments in source codes written in Java, C, C++, C#, COBOL, and VB. At the time

Table 10: Optimal set of tools for most supported programming languages.

Programming Language	Optimal set of tools	Metrics covered
C	CAST's AIP, Understand, CCFinderX, CMT++	14/14
C++	CAST's AIP, Understand, CCFinderX, CMT++	14/14
C#	CAST's AIP, Understand, CCFinderX, CMT++	14/14
Java	(CAST's or JHawk), (CCFinderX or Ref-Finder), Understand, CMTJava, (MetricsReloaded, Squale, or Codacy)	15/15
JavaScript	CAST'S AIP, Understand, escomplex, (CodeAnalyzers or eslint)	14/14

Table 11: Optimal set of open-source tools for most supported programming languages.

Programming Language	Optimal set of tools	Metrics covered
C	CBR Insight, CCFinderX, CodeAnalyzers, Halstead Metrics Tool, MetricsReloaded	12/14
C++	CBR Insight, CCFinderX, CodeAnalyzers, Halstead Metrics Tool, Squale	11/14
C#	CBR Insight, CCFinderX, CodeAnalyzers	9/14
Java	(CCFinderX or Ref-Finder), CKJM, CodeAnalyzers, Halstead Metrics Tool, MetricsReloaded	13/15
JavaScript	CBR Insight, CodeAnalyzers	8/14

of writing this SLR, the project appears to be not maintained, and the last version dates back to May 2010.

(iii) *CKJM*. The tool [64], cited in five of our selected studies, supports only the Java programming language. It can calculate the six metrics of the C&K suite, plus the afferent coupling (CA), and the number of public methods (NPM). The results can be exported in XML format, and the program can be integrated with Ant. The tool appears to have been discontinued, since its last release at the time of the writing of this manuscript, i.e., the 1.9, was released in 2008.

(iv) *CodeMetrics (IntelliJ IDEA Plugin)*. The tool is released under the MIT license. It can compute the complexity of each method and the total for each class of the source code. It does not calculate the standard cyclomatic complexity, but an approximation of that. At the time of writing this article, the project is still maintained.

(v) *Escomplex*. It is a tool that performs a software complexity analysis of JavaScript abstract syntax trees. It can compute several metrics among those previously identified, e.g., the maintainability index, the Halstead suite, McCabe's CC, and LOC. The results are returned in JSON format so that they can be used by front-end programs. At the time of writing this SLR, the last version of the tool dates back to the end of 2015.

(vi) *Eslint*. The tool is a linting (i.e., running a program to analyze code to automatically verify the presence of potential errors) utility for JavaScript. The tool allows using a set of built-in linting rules and also allows adding custom ones as plugins that are dynamically loaded. The tool also allows fixing automatically some of the issues that it finds. At the time of writing the SLR, the project's last available release is v6.5.1, released in September 2019.

(vii) *Halstead Metrics Tool*. A software metrics analyzer for C, C++, and Java programs. It provides a computation of the Halstead metric suite only. It is written in Java and can export the results in HTML and PDF. At the time of writing this SLR, no development of the tools has been performed after 2016.

(viii) *JSInspect*. It is a program to analyze JavaScript code in search of code smells, such as duplicate code and repeated logic. The basic aim of the tool is to identify separate portions of code with a similar structure in a software project, based on the AST node types, e.g., BlockStatement, VariableDeclaration, and ObjectExpression. At the moment of writing this SLR, the tool seems to have been discontinued, since the last commit on the repository dates back to August 2017.

(ix) *MetricsReloaded (IntelliJ IDEA Plugin)*. The tool, in addition to being available as a plugin for the popular IDE IntelliJ IDEA, can also be used standalone from the command line. The project seems to be discontinued since September 2017.

(x) *Quamoco Benchmark for Software Quality*. It is a Java-based tool aimed to analyze code written in Java. It is based on the Quamoco model, aimed at integrating abstract code quality attributes and concrete software quality assessments [65]. The tool is mentioned in several academic studies selected in this SLR, and its code repository is available on GitHub. From the repository, it can be seen that the development has been discontinued, and the last commit dates back to July 2013.

(xi) *Ref-Finder (Eclipse Plugin)*. A tool whose principal aim is to detect refactorings occurred between two

program versions and helping the developers to better understand code changes. The plugin can recognize even complex refactoring with high precision, and it supports 65 of the 72 refactoring types in Fowler's catalogue [66].

(xii) *SonarQube.* Along with CodeAnalyzers, it is a product by SonarSource. The two products are provided in two different editions: the community one, which is open source, and a commercial one. The community edition features fewer metrics and less programming languages and does not provide the security reports that are a main feature of the commercial versions. They support more than 25 programming languages (15 in the OS editions) and hundreds of rules, among which code smells and maintainability metrics.

(xiii) *Squale (Software QUALity Enhancement).* It is based on third party technologies (commercial or open source) that produce raw quality information (such as metrics for instance) and uses quality models (such as ISO-9126) to aggregate the raw information into high-level quality factors. Released under the LGPLv3 license, it is a program to help to assess the software quality, giving as output information to be used from both the development and the management team, dealing with both technical and economic aspects of software quality. It targets different programming languages (including Java, C/C++,.NET, PHP, and Cobol) and utilizes code metrics and quality models to assess the grade of the code. The tool appears to be discontinued, and the last version of the program, v7.1, released in May 2011.

3.3.3. Correspondence between Tools and Languages. Figure 4 shows which languages are supported by each tool. Some of the considered tools support a wide variety of languages, such as Understand, Codacy, and the tools by SonarSource (SonarQube and CodeAnalyzers). CBR Insight, as stated before, is based on Understand; hence, it supports the same set of programming languages. The majority of tools, however, support a limited number of programming languages or also just one. For instance, JHawk, CKJM, CodeMetrics, and Ref-Finder all support only Java; JSInspect, escomplex, and eslint are tailored to work only with JavaScript.

From the table, it is evident that the closed-source tools support more programming languages (an average of 10.5) compared to open-source tools (an average of 4.85). By analyzing the primary studies selected for this SLR, it is also reported that closed-source tools tend to support some metrics better than open-source counterparts: for instance, a comparative study between different tools capable of MI reports a higher dependability of such metric when computed using closed-source tools rather than open-source alternatives [6].

Figure 5 shows how many closed-source and open-source tools have been found for each language. From that chart, it is evident that some languages are better supported than others. Java, C, and C++, followed closely by JavaScript and C#, are supported by at least half of the tools we considered in our study. More specifically, Java, C, C++, and C# are supported by almost all the closed-source programs we found. Some less widespread languages (e.g., APAB, GO, RPG, and T-SQL) are supported only by open-source tools, among the set of tools that we gathered from analyzing the primary studies used for the SLR.

3.3.4. Correspondence between Tools and Metrics. Figure 6 (CS tools and metrics) shows what metrics are calculated by each of the considered tools. For conciseness, only the metrics that are computed by at least one tool are reported in the table. In the upper section of the table, the most popular metrics identified in the answer to RQ1 are reported. Instead, the lower section of the table includes other metrics belonging to the complete set of metrics found in the set of primary studies mined from the literature. The table features a mark for a tool and a metric only in cases when an explicit reference to such metric has been found in the documentation of the tool.

Also, a suite was considered as supported if at least one of its metrics was supported by a given tool.

In the case of the closed-source tools, the metrics have been most of the times inferred from limited documentation. Most of the times, in fact, closed-source tools provide dashboards with custom-defined evaluations of the code, for which the linkage with widespread software metrics is unclear. For instance, the Codacy tool provides a single, overall grade for a software project, between A and F. This grade depends on a set of tool-specific parameters: error-proneness, code complexity, code style, unused code, security, compatibility, documentation, and performance. In addition to some metrics whose usage was explicitly mentioned by the tool's creators (e.g., number of comments and JavaDoc lines for the documentation property and McCabe's CC for the code complexity property), it was not possible to find the complete set of metrics used internally by the tool.

In many cases, the tools compute also compound metrics (i.e., metrics built on top of other ones reported in the literature) or metrics that were not previously found in the analysis of the literature performed to answer to RQ1. In these cases, the tools were labelled as featuring *other* metrics: this information is reported in the last row of the table.

As it is evident from the table, no tool supported all the most popular metrics previously identified. The number of supported metrics among the most popular ones ranged from 1 to 10. Two tools featured just one suite/metric from the set of the most popular ones. The Halstead Metrics Tool, as evident from its name, is an open-source tool with the only purpose of computing the entire set of metrics of the Halstead suite; as well, the CodeMetrics plugin is a basic tool capable of computing only the McCabe cyclomatic complexity (for each method and the total for each class of the project). Quamoco is indeed not only a tool but instead a quality metamodel, based on a set of metrics that are defined, in the scope of the paper presenting the approach, as base

measures; the metamodel is theoretically applicable to any kind of base measure that can be computed through static analysis of source code; however, the literature presenting the tool mentions only the LOC metric explicitly. Some other tools, such as JSInspect, CCFinderX, and Ref-Finder tools, featured a limited set of the maintainability metrics previously identified, since they were mainly focused on other aspects of code quality, e.g., detecting code duplicates and code smells.

Tools such as MetricsReloaded, Squale, and SonarQube featured large sets of derived metrics, which were obtained as specializations, sums, or averages of basic metrics such as the McCabe cyclomatic complexity or the coupling between classes.

The bar graph in Figure 7 reports the number of tools that featured each of the considered metrics. Also, in this case, the metrics were divided into three sections on the x-axis: the 15 metrics/suites deemed as most popular in the answer to RQ1, other metrics from the full set, and other metrics not in the set of metrics mined from the literature. Two metrics stood out in terms of the number of tools that supported them. The LOC metric, despite many papers in the literature question its usefulness as a maintainability metric, was supported by 14 out of 19 tools. The metric is closely followed by the cyclomatic complexity (CC), which was supported by 13 tools. Those numbers were expectable since both the metrics are simple to compute and are needed by many other derived metrics. On the other hand, three of the most popular metrics were used by only two of the selected tools. The CHANGE metric refers to the changed lines of code between different releases of the same application and was not computed by most of the tools that performed static analysis on single versions of the application; it was instead computed by two tools that were particularly aiming at measuring code refactorings and smells. The LCOM2 metric is an extension of the LCOM metric, which is part of the C&K suite; several tools just mentioned the adoption of the suite without explicitly mentioning possible adoptions of enhanced versions of the metrics; finally, the message passing coupling was adopted by two tools and in both cases defined with the synonym fan-out.

In general, closed-source tools featured a higher number of metrics than open-source counterparts. Open-source tools, several times, were, in fact, plugins of limited dimension, tailored to compute just a single metric or suite. If only the measures mined from the primary studies are considered, the closed-source tools were able to compute an average of slightly less than 8 metrics, while open-source tools were able to compute an average of 5 metrics. Of the set of 15 most popular metrics, on average 6 could be computed by the closed-source tools and 3 by the open-source tools.

3.3.5. Correspondence between Tools and Languages. Table 9 reports the tools able to compute each of the set of most popular metrics for the five languages that were supported the most (see bar plot in Figure 5). We took into account C, C++, C#, Java, and JavaScript, since at least 7 tools (more than the average for all programming languages) supported them. The table reports all tools that can compute a metric for a given language. For the case of the JLOC metric, the relevant information is only related to the tools compatible with Java, since the metric cannot be computed for other programming languages. Open-source tools are highlighted by using bold lettering. As it is evident from the table, the most featured metrics (e.g., CC and LOC) can be computed with many alternative tools (either closed source or open source) for the same languages. On the other hand, several metrics can be computed by just a single tool: for instance, CCFinderX is the only tool that explicitly supports the CHANGE metric for all the languages of the C family, or the MPC (message passing coupling) metric is explicitly supported only by the CAST's Application Intelligence Platform for the languages of the C family and JavaScript.

3.4. RQ2.2: Ideal Selection of Tools. Tables 10 and 11 show the optimal set of tools to cover all the most popular metrics shown in Table 5. The former takes into account both closed-source and open-source tools; the latter only considers open-source tools. We define an optimal set of tools as the minimal set of tools which can cover the highest possible amount of metrics (or suites) out of the set of 14 most mentioned ones (15 for Java, for which also the JLOC metric can be computed). Inside round brackets, we identified alternative tools that could be selected without influencing the number of tools in the optimal set or the number of metrics covered.

By using both closed-source and open-source tools, it is possible to compute all the most mentioned metrics with an optimal set of 4 tools for all languages except for Java, for which 5 tools were necessary. Specifically, for all the languages of the C family, all the metrics are covered by CAST's Application Intelligence Platform, Understand, CCFinderX, and CMT++. Java needed also the adoption of a tool among MetricsReloaded, Squale, or Codacy to compute the JLOC metric; JHawk and Ref-Finder could be used, respectively, as alternatives to CAST's AIP and CCFinderX; CMTJava had to be selected instead of CMT++. For JavaScript, escomplex and one between CodeAnalyzers or eslint have to be included in the set, replacing CCFinderX and CMT.

By using open-source tools only, it is not possible to obtain full coverage of the most mentioned metrics. The LCOM2 and MPC metrics were not explicitly supported by any of the considered open-source tools. The maximum amount of metrics that could be supported with an optimal set of tools ranged between 8 (for the JavaScript programming language, with two tools) and 13 (for Java, with 5 tools, also including the JLOC metric).

4. Threats to Validity

Threats to construct validity, for an SLR, are related to failures in the claim of covering all the possible studies related to the topic of the review. In this study, the paper was mitigated with a thorough and reproducible definition of the search strategy and with the use of synonyms in the search

strings. Also, all the principal sources for the scientific literature were taken into consideration for the extraction of the primary studies.

Threats to internal validity are related to the data extraction phase of the SLR. The authors of this paper evaluated the papers manually, according to the defined inclusion and exclusion criteria. The authors limited biases in the inclusion and exclusion of the paper by discussing disagreements. The metric selection phase was performed based on the opinions extracted from the examined primary studies (considered as adverse, neutral, or positive). Again, the reading of the papers and the subsequential opinion assignments are based on the judgment of the authors and may suffer from misinterpretation of the original opinions. It is, however, worth mentioning that none of the authors of this paper were biased towards the demonstration of a specific preference for one of the available metrics.

Threats to external validity are related to the incapability of obtaining generalized conclusions from the conducted study. This threat is limited in this study since its main results, i.e., the sets of most popular metrics, were formulated w.r.t. to a set of programming languages. The results are not generalized to programming languages that were not discussed in the primary studies examined in the SLR.

5. Related Works

The literature offers several secondary studies regarding code metrics and tools. However, usually, those studies analyze or present a set of tools, and they describe the metrics based on the features of the tool. Our review instead started from an analysis of the literature that was tailored at finding all metrics available in relevant studies in the literature, and then the focus was moved to tools to understand whether the found metrics were supported or not by those tools.

For example, in the literature review published in 2008, Lincke et al. [67] compared different software metric tools showing that, in some cases, different tools provided uncompatible results; the authors also defined a simple universal software quality model, based on a set of metrics that were extracted from the examined tools. Dias Canedo et al. [68] performed a systematic literature review for finding tools that can perform software measures. Starting from the tools, the authors analyzed the tool features and described the metrics the software could analyze. For their secondary studies, the authors analyzed papers from 2007 to 2018.

On the other hand, there are also other secondary studies explicitly focused on metrics as the comparative case study published in 2012 by Sjoberg et al. [69], which has a focus on code maintainability metrics but only considers a subset of 11 metrics for the Java language. The work had a primary aim at questioning the consistency between different metrics in the evaluation of maintainability of software projects.

The systematic mapping study published in 2017 by Nuñez-Varela et al. [70] is one of the most complete works on this topic. The authors discovered 300 source code metrics by analyzing papers published from 2010 to 2015. They also mapped those metrics with the tools that can use them. This work, however, covers a limited time window and does not focus on a specific family of software metrics, gathering dynamic and change metrics along with static ones.

In a recent systematic mapping and review, Elmidaoui et al. identified 82 empirical studies about software product maintainability prediction [71]. The paper focuses on analyzing the different methods available for maintainability estimation, including fuzzy, neurofuzzy, artificial neural network (ANN), support vector machines (SVMs), and group method of data building (GMDH). The paper concludes that the prediction of software maintainability, albeit many techniques are available to perform it, is still limited in industrial practice.

Our work differs from the secondary studies presented above. Our point of view is finding the most common maintainability metrics and tools to be applied to new programming languages. For doing so, we analyzed papers in a 20-year time window (2000–2019). We also distinguished open-source tools from closed-source tools, and for each of them, we mapped the maintainability metrics they use. The output of this work is actionable by practitioners wanting to create new tools for applying maintainability metrics to new programming languages.

Other primary studies in the literature presented (or used) popular software metric tools, which were, however, not extracted during our study selection phase, since their primary purpose was not analyzing code from a maintenance point of view, and hence, the manuscripts could not be found by searching for the *maintainability* keyword. A relevant example of those tools is *CCCC*, a widespread tool to evaluate code written with object-oriented languages [72, 73].

6. Conclusion

Maintainability is a fundamental feature for software projects, and the scientific literature has proposed several approaches, metrics, and tools to evaluate it in real-world scenarios. With this systematic literature review, we wanted to have an overview of the most used maintainability metrics in the literature in the last twenty years, to find the most commonly used ones, which can be used to evaluate existing software, and that can be adapted to measure the maintainability of new programming languages. In doing so, we wanted to provide the readers actionable results by identifying sets of (closed- and open-source) tools that can be adopted to be able to compute all the most popular metrics for a specific programming language.

With the application of a formalized SLR procedure, we identified a total of 174 metrics, some of which were distributed in 10 metric suites. Among them, we extracted a set of 15 most frequently mentioned ones, of which we reported the definitions and formulae. We also identified a set of 38 tools mentioned in primary studies about software maintainability metrics: by filtering those that were not made available by the authors, could not be retrieved on the web, or were no longer available, we came up with a set of 6

closed-source and 13 open-source tools that can be used to evaluate software projects, covering 34 different programming languages. By analyzing the tools, we found that Java, JavaScript, C, C++, and C# are the most common programming language compatibles with the analyzed tools. By pairing the information about supported programming languages and supported metrics, we found that it is possible to find an optimal selection of at most five tools to cover all the most mentioned metrics for the languages of the Java and C family. However, not all the most popular metrics could be computed by taking into consideration only open-source tools.

This manuscript can provide actionable guidelines for practitioners who want to measure the maintainability of their software by providing a mapping between popular metrics and tools able to compute them. Also, this manuscript provides actionable guidelines for practitioners and researchers who may want to implement tools to measure software metrics for newer programming languages. Our work identifies which tools can provide the computation of the most popular maintenance metrics and the support they provide to the most common programming languages. Our work also provides pointers to existing open-source tools already available for computing the metrics, which can be leveraged by tool developers as guidelines for their counterparts for source code written in different languages.

As future work, we aim at implementing a tool that uses the set of metrics we found in RQ1.2 to analyze code written in the Rust programming language. For the Rust programming language, we identified no tool capable of computing the most popular maintainability metrics mentioned in the literature. We plan to extend a tool named Tokei[1], which offers compatibility with many modern programming languages. The results of these works are considered capable of easing other researchers to create tools for measuring the maintainability of modern programming languages and for encouraging new comparisons between programming languages.

Data Availability

The data used to support the findings of this study are included within the article in the form of references linking to resources available on the FigShare public open repository.

Conflicts of Interest

The authors declare that there are no conflicts of interest regarding the publication of this paper.

Acknowledgments

Mozilla Research funded this project with the research grant 2018 H2. The project title is "Algorithms clarity in Rust: advanced rate control and multithread support in rav1e." This project aims to understand how the Rust programming language improves the maintainability of code while implementing complex algorithms.

References

[1] F. Zampetti, S. Scalabrino, R. Oliveto, G. Canfora, and M. Di Penta, "How open source projects use static code analysis tools in continuous integration pipelines," in *Proceedings of the 2017 IEEE/ACM 14th International Confer- Ence on Mining Software Repositories (MSR)*, IEEE, Buenos Aires, Argentina, May 2017.

[2] IEEE Standards Association, *IEEE Standard Glossary of Software Engineering Terminology*, IEEE Standards Association, Piscataway, NJ, USA, 1990.

[3] C. Van Koten and A. Gray, "An application of bayesian network for predicting object-oriented software maintainability," *Information and Software Technology*, vol. 48, no. 1, pp. 59–67, 2006.

[4] Y. Zhou and H. Leung, "Predicting object-oriented software maintainability using multivariate adaptive regression splines," *Journal of Systems and Software*, vol. 80, no. 8, pp. 1349–1361, 2007.

[5] A. Kaur, K. Kaur, and K. Pathak, "Software maintainability prediction by data mining of software code metrics," in *Proceedings of the 2014 International Conference on Data Mining and Intelligent Computing (ICDMIC)*, Delhi, India, September 2014.

[6] M. I. Sarwar, W. Tanveer, I. Sarwar, and W. Mahmood, "A comparative study of mi tools: defining the roadmap to mi tools standardization," in *Proceedings of the 2008 IEEE International Multitopic Conference*, Karachi, Pakistan, December 2008.

[7] N. D. Matsakis and F. S. Klock II, "The rust language," *ACM SIGAda Ada Letters*, vol. 34, no. 3, pp. 103-104, 2014.

[8] S. Klabnik and C. Nichols, *The Rust Programming Language*, No Starch Press, San Francisco, CA, USA, 2018.

[9] A. Tahir and R. Ahmad, "An aop-based approach for collecting soft- ware maintainability dynamic metrics," in *Proceedings of the 2010 Second International Conference on Computer Research and Development*, Beijing, China, May 2010.

[10] K. Barbara A and S. Charters, "Guidelines for performing systematic literature reviews in software engineering," Tech. Rep. 2007, Durham University, Durham, England, 2007.

[11] B. A. Kitchenham, T. Dyba, and M. Jorgensen, "Evidence-based soft- ware engineering," in *Proceedings of the 26th International Conference on Software Engineering*, IEEE Computer Society, New York, NY, USA, pp. 273–281, May 2004.

[12] B. Kitchenham, O. Pearl Brereton, D. Budgen, M. Turner, J. Bailey, and S. Linkman, "Systematic literature reviews in software engineering - a systematic literature review," *Information and Software Technology*, vol. 51, no. 1, pp. 7–15, 2009.

[13] R. van Solingen, V. Basili, G. Caldiera, and H. D. Rombach, *Goal Question Metric (GQM) Approach*, 2002.

[14] J. Ostberg and S. Wagner, "On automatically collectable metrics for software maintainability evaluation," in *Proceedings of the 2014 Joint Conference of the International Workshop on Software Measurement and the International Conference on Software Process And Product Measurement*, Rotterdam, The Netherlands, October 2014.

[15] J. Ludwig, S. Xu, and F. Webber, "Compiling static software metrics for reliability and maintainability from github repositories," in *Proceedings of the 2017 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, Banff, AB, Canada, October 2017.

[16] H. Liu, X. Gong, L. Liao, and B. Li, "Evaluate how cyclomatic complexity changes in the context of software evolution," in *Proceedings of the 2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, Tokyo, Japan, July 2018.

[17] P. Jacsó, "Calculating the index and other bibliometric and scientometric indicators from Google Scholar with the Publish or Perish software," *Online information Review*, vol. 33, no. 6, pp. 1189–1200, 2009.

[18] C. Wohlin, "Guidelines for snowballing in systematic literature studies and a replication in software engineering," in *Proceedings of the 18th International conference on evaluation and assessment in software engineering*, Ciudad Real, Spain, May 2014.

[19] I. K´ad´ar, P. Hegedus, R. Ferenc, and T. Gyim´othy, "A code refactoring dataset and its assessment regarding software maintainability," in *Proceedings of the 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Osaka, Japan, March 2016.

[20] J. Gil, M. Goldstein, and D. Moshkovich, "An empirical investigation of changes in some software properties over time," in *Proceedings of the 2012 9th IEEE Working Conference on Mining Software Repositories (MSR)*, Zurich, Switzerland, June 2012.

[21] A. Jain, S. Tarwani, and A. Chug, "An empirical investigation of evolu- tionary algorithm for software maintainability prediction," in *Proceedings of the 2016 IEEE Students' Conference on Electrical, Electronics and Computer Science (SCEECS)*, Bhopal, India, March 2016.

[22] B. Curtis, J. Sappidi, and J. Subramanyam, "An evaluation of the inter- nal quality of business applications: does size matter?" in *Proceedings of the 33rd International Conference on Software Engineering, ICSE '11*), New York, NY, USA, May 2011.

[23] R. S. Chhillar and S. Gahlot, "An evolution of software metrics: a review," in *Proceedings of the International Conference on Advances in Image Processing, ICAIP 2017*), New York, NY, USA, 2017.

[24] Y. Tian, C. Chen, and C. Zhang, "Aode for source code metrics for improved software maintainability," in *Proceedings of the 2008 Fourth International Con- Ference on Semantics, Knowledge And Grid*, Beijing, China, December 2008.

[25] A. Kaur, K. Kaur, and K. Pathak, "A proposed new model for main- tainability index of open source software," in *Proceedings of 3rd International Conference on Reliability, Infocom Technologies And Optimization*, Noida, India, October 2014.

[26] N. Barbosa and K. Hirama, "Assessment of software maintainability evolution using C&K metrics," *IEEE Latin America Transactions*, vol. 11, no. 5, pp. 1232–1237, 2013.

[27] S. Misra, A. Adewumi, L. Fernandez-Sanz, and R. Damasevicius, "A suite of object oriented cognitive complexity metrics," *IEEE Access*, vol. 6, pp. 8782–8796, 2018.

[28] S. Rongviriyapanish, T. Wisuttikul, B. Charoendouysil, P. Pitakket, P. Anancharoenpakorn, and P. Meananeatra, "Changeability prediction model for java class based on multiple layer perceptron neural network," in *Proceedings of the 2016 13th International Conference on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology (ECTI-CON)*, Chiang Mai, Thailand, June 2016.

[29] S. Arshad and C. Tjortjis, "Clustering software metric values extracted from c# code for maintainability assessment," in *Proceedings of the 9th Hellenic Conference on Artificial Intelligence, SETN '16*), New York, NY, USA, May 2016.

[30] M. Pizka, "Code normal forms," in *Proceedings of the 29th Annual IEEE/NASA Software Engineering Workshop*, Greenbelt, MD, USA, April 2005.

[31] M. A. A. Mamun, C. Berger, and J. Hansson, "Correlations of software code metrics: an empirical study," in *Proceedings of the 27th International Workshop on Software Measurement And 12th International Conference on Software Process And Product Measurement, IWSM Mensura '17*, New York, NY, USA, May 2017.

[32] T. L. Alves, C. Ypma, and J. Visser, "Deriving metric thresholds from benchmark data," in *Proceedings of the 2010 IEEE International Conference on Software Maintenance*, Timisoara,Romania, September 2010.

[33] T. Matsushita and I. Sasano, "Detecting code clones with gaps by function applications," in *Proceedings of the 2017 ACM SIGPLAN Work- Shop on Partial Evaluation and Program Manipulation, PEPM 2017*), New York, NY, USA, may 2017.

[34] L. M. d. Silva, F. Dantas, G. Honorato, A. Garcia, and C. Lucena, "Detecting modularity flaws of evolving code: what the history can reveal?" in *Proceedings of the 2010 Fourth Brazilian Symposium on Software Components, Architectures And Reuse*, Bahia, Brazil, September 2010.

[35] A. Ch´avez, I. Ferreira, E. Fernandes, D. Cedrim, and A. Garcia, "How does refactoring affect internal quality attributes?: a multi-project study," in *Proceedings of the 31st Brazilian Symposium on Software Engineering, SBES'17*), New York, NY, USA, May 2017.

[36] Y. Ma, K. He, B. Li, and X. Zhou, "How multiple-dependency structure of classes affects their functions a statistical perspective," in *Proceedings of the 2010 2nd International Conference on Software Technology and Engineering*, San Juan, PR, USA, October 2010.

[37] M. Wahler, U. Drofenik, and W. Snipes, "Improving code maintainability: a case study on the impact of refactoring," in *Proceedings of the 2016 IEEE Inter- national Conference on Software Maintenance and Evolution (ICSME)*, North CA, USA, October 2016.

[38] G. Kaur and B. Singh, "Improving the quality of software by refactoring," in *Proceedings of the 2017 International Conference on Intelligent Computing and Control Systems (ICICCS)*, Madurai, India, June 2017.

[39] M. Yan, X. Zhang, C. Liu, J. Zou, L. Xu, and X. Xia, "Learning to aggre- gate: an automated aggregation method for software quality model," in *Proceedings of the 2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, Buenos Aires, Argentina, May 2017.

[40] K. Chatzidimitriou, M. Papamichail, T. Diamantopoulos, M. Tsapanos, and A. Symeonidis, "npm-miner: an infrastructure for measuring the quality of the npm registry," in *Proceddings of the 2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*, Gothenburg, Sweden, May 2018.

[41] J. Bohnet and J. D¨ollner, "Monitoring code quality and development activity by software maps," in *Proceedings of the 2Nd Workshop on Managing Technical Debt, MTD '11*), New York, NY, USA, May 2011.

[42] N. Narayanan Prasanth, S. Ganesh, and G. Arul Dalton, "Prediction of maintainability using software complexity analysis: An extended frt," in *Proceedings of the 2008 International Conference on Computing, Communication and Networking*, Karur, Tamil Nadu, India, December 2008.

[43] L. Wang, X. Hu, Z. Ning, and W. Ke, "Predicting object-oriented software maintainability using projection pursuit regression," in *Proceedings of the 2009 First International Conference on Information Science and Engineering*, Nanjing, China, December 2009.

[44] D. I. Sjøberg, B. Anda, and A. Mockus, "Questioning software mainte- nance metrics: a comparative case study," in *Proceedings of the ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM '12*ACM, New York, NY, USA, ACM, September 2012.

[45] A. Hindle, M. W. Godfrey, and R. C. Holt, "Reading beside the lines: indentation as a proxy for complexity metric," in *Proceedings of the 2008 16th IEEE In- ternational Conference on Program Comprehension*, Amsterdam, The Netherland, June 2008.

[46] Y. Lee and K. H. Chang, "Reusability and maintainability metrics for object-oriented software," in *Proceedings of the 38th Annual on South-east Regional Conference, ACM-SE 38*), New York, NY, USA, May 2000.

[47] B. R. Sinha, P. P. Dey, M. Amin, and H. Badkoobehi, "Software com- plexity measurement using multiple criteria," *Journal of Computing Sciences in Colleges*, vol. 28, pp. 155–162, April 2013.

[48] P. Vytovtov and E. Markov, "Source code quality classification based on software metrics," in *Proceedings of the 2017 20th Conference of Open Innovations association (FRUCT)*, Saint Petersburg, Russia, April 2017.

[49] N. E. Gold, A. M. Mohan, and P. J. Layzell, "Spatial complexity metrics: an investigation of utility," *IEEE Transactions on Software Engineering*, vol. 31, no. 3, pp. 203–212, 2005.

[50] J. Ludwig, S. Xu, and F. Webber, "Static software metrics for reliability and maintainability," in *Proceedings of the 2018 International Confer- Ence on Technical Debt, TechDebt '18*, pp. 53-54, New York, NY, USA, May 2018.

[51] M. Saboe, "The use of software quality metrics in the materiel release process experience report," in *Proceedings of the Second Asia-Pacific Conference on Quality Software*, Brisbane, Queensland, Australia, December 2001.

[52] A. F. Yamashita, H. C. Benestad, B. Anda, P. E. Arnstad, D. I. K. Sjoberg, and L. Moonen, "Using concept mapping for maintainability assessments," in *Proceedings of the 2009 3rd International Symposium on Empirical Soft-Ware Engineering And Measurement*, Lake Buena Vista, FL, USA, October 2009.

[53] D. Threm, L. Yu, S. Ramaswamy, and S. D. Sudarsan, "Using normalized compression distance to measure the evolutionary stability of software systems," in *Proceedings of the 2015 IEEE 26th International Symposium on Soft- Ware Reliability Engineering (ISSRE)*, Gaithersbury, MD, USA, November 2015.

[54] R. Gon¸calves, I. Lima, and H. Costa, "Using Tdd for Developing Object- Oriented Software — a Case Study," in *Proceedings of the 2015 Latin American Computing Conference (CLEI)*, Arequipa, Peru, October 2015.

[55] A. Jermakovics, R. Moser, A. Sillitti, and G. Succi, "Visualizing software evolution with lagrein," in *Proceedings of the Companion to the 23rd ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications, OOPSLA Companion '08*), New York, NY, USA, May 2008.

[56] T. J. McCabe, "A complexity measure," *IEEE Transactions on Software Engineering*, vol. SE-2, no. 4, pp. 308–320, 1976.

[57] R. S. D. H. N. K. C. W. G. Jay and J. Hale, "Cyclomatic complexity and lines of code: empirical evidence of a stable linear relationship," *Journal of Software Engineering and Application*, vol. 2, pp. 137–143, 2009.

[58] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 476–493, 1994.

[59] M. H. Halstead, *Elements of Software Science (Operating and Program- Ming Systems Series)*, Elsevier Science Inc., New York, NY, USA, 1977.

[60] I. Herraiz, J. Gonzalez-Barahona, and G. Robles, "Towards a theoretical model for software growth," in *Proceedings of the Fourth International Workshop on Mining Software Repositories (MSR'07:ICSE Workshops 2007)*, Minneapolis, MN, USA, May 2007.

[61] P. Oman and J. Hagemeister, "Metrics for assessing a software system's maintainability," in *Proceedings Conference on Software Maintenance*, Victoria, British Columbia, Canada, November 1992.

[62] W. Li and S. Henry, "Object-oriented metrics that predict maintainability," *Journal of Systems and Software*, vol. 23, no. 2, pp. 111–122, 1993.

[63] B. Curtis, J. Sappidi, and A. Szynkarski, "Estimating the principal of an application's technical debt," *IEEE Software*, vol. 29, no. 6, pp. 34–42, 2012.

[64] D. Spinellis, "Working with unix tools," *IEEE Software*, vol. 22, no. 6, pp. 9–11, 2005.

[65] S. Wagner, K. Lochmann, L. Heinemann et al., "The quamoco product quality modelling and assessment approach," in *Proceedings of the 34th International Conference on Software Engineering*, IEEE Press, Zurich, Switzerland, June 2012.

[66] M. Fowler, *Refactoring: Improving the Design of Existing Code*, Addison- Wesley Professional, Boston, MA, USA, 2018.

[67] R. Lincke, J. Lundberg, and W. L¨owe, "Comparing software metrics tools," in *Proceedings of the 2008 International Symposium on Software testing and Analysis*, Seattle, WA, USA, July 2008.

[68] E. Dias Canedo, K. Valen¸ca, and G. A. Santos, "An analysis of measure- ment and metrics tools: a systematic literature review," in *Proceedings of the 52nd Hawaii International Conference on System Sciences*, Maui, HI, USA, January 2019.

[69] D. I. Sjøberg, B. Anda, and A. Mockus, "Questioning software main- tenance metrics: a comparative case study," in *Proceedings of the 2012 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, IEEE, Lund, Sweden, September 2012.

[70] A. S. Nuñez-Varela, H. G. P´erez-Gonzalez, F. E. Mart´ınez-Perez, and C. Soubervielle-Montalvo, "Source code metrics: a systematic mapping study," *Journal of Systems and Software*, vol. 128, pp. 164–197, 2017.

[71] S. Elmidaoui, L. Cheikhi, A. Idri, and A. Abran, "Empirical studies on software product maintainability prediction: a systematic mapping and review," *E-Informatica Software Engineering Journal*, vol. 13, no. 1, 2019.

[72] C. Thirumalai, P. A. Reddy, and Y. J. Kishore, "Evaluating software metrics of gaming applications using code counter tool for c and c++ (cccc)," in *Proceedings of the 2017 International Conference of Electronics, Communication and Aerospace Technology (ICECA)*, Coimbatore, India, April 2017.

[73] U. Poornima, "Unified design quality metric tool for object-oriented ap- proach including other principles," *International Journal of Computer Applications in Technology*, vol. 26, pp. 1–4, 2011.