The Journal of Systems & Software

A Systematic Tertiary Review on Source-code Metrics

--Manuscript Draft--

Manuscript Number:	JSSOFTWARE-D-22-00090
Article Type:	Research Paper
Keywords:	Source-code metrics; software quality; source-code readability; cognitive load
Abstract:	The difficulty of software development tasks depends on several factors including the characteristics of the underlying source-code. These characteristics can be captured and measured using source-code metrics, which, in turn, can provide indications about the difficulty of the source-code. From a cognitive perspective, this difficulty is due to an increase of developers' cognitive load, which can be estimated using psycho-physiological measures (e.g., electroencephalography). Based on these measures, a handful of studies investigated the relationship between source-code metrics and developers' cognitive load. For most of the metrics, such a relationship could not be established. While these studies have used a small subset of metrics, the literature comprises hundreds of other metrics, which regardless of the existing literature reviews surveying them, a consolidated overview is still needed to understand their properties and leverage their potential to align with developers' cognitive load. This need is addressed in this paper through a Systematic Tertiary Review (STR) covering the full spectrum of source-code metrics and studying their properties across several dimensions. The outcome of this STR is intended to provide a consolidated overview of source-code metrics, guide practitioners in choosing the appropriate ones and raise new research challenges to be addressed in the future.

- A systematic tertiary review covering a wide spectrum of source-code metrics
- A conceptual framework consolidating the characteristics of source-code metrics
- A research agenda delineating the path for future research on source-code metrics

Title page with author details

A Systematic Tertiary Review on Source-code Metrics

Amine Abbad-Andaloussi[a,*]

[a]*Institute of Computer Science, University of St. Gallen, 9000 St. Gallen, Switzerland*

Abstract

The difficulty of software development tasks depends on several factors including the characteristics of the underlying source-code. These characteristics can be captured and measured using source-code metrics, which, in turn, can provide indications about the difficulty of the source-code. From a cognitive perspective, this difficulty is due to an increase of developers' cognitive load, which can be estimated using psycho-physiological measures (e.g., electroencephalography). Based on these measures, a handful of studies investigated the relationship between source-code metrics and developers' cognitive load. For most of the metrics, such a relationship could not be established. While these studies have used a small subset of metrics, the literature comprises hundreds of other metrics, which regardless of the existing literature reviews surveying them, a consolidated overview is still needed to understand their properties and leverage their potential to align with developers' cognitive load. This need is addressed in this paper through a Systematic Tertiary Review (STR) covering the full spectrum of source-code metrics and studying their properties across several dimensions. The outcome of this STR is intended to provide a consolidated overview of source-code metrics, guide practitioners in choosing the appropriate ones and raise new research challenges to be addressed in the future.

Keywords: Source-code metrics, software quality, source-code readability, cognitive load

*Work supported by the International Postdoctoral Fellowship (IPF) Grant (Number: 1031574) from the University of St. Gallen, Switzerland.

*Corresponding author

Email address: amine.abbad-andaloussi@unisg.ch (Amine Abbad-Andaloussi)

A Systematic Tertiary Review on Source-code Metrics

Amine Abbad-Andaloussi[a,*]

[a]*Institute of Computer Science, University of St. Gallen, 9000 St. Gallen, Switzerland*

Abstract

The difficulty of software development tasks depends on several factors including the characteristics of the underlying source-code. These characteristics can be captured and measured using source-code metrics, which, in turn, can provide indications about the difficulty of the source-code. From a cognitive perspective, this difficulty is due to an increase of developers' cognitive load, which can be estimated using psycho-physiological measures (e.g., electroencephalography). Based on these measures, a handful of studies investigated the relationship between source-code metrics and developers' cognitive load. For most of the metrics, such a relationship could not be established. While these studies have used a small subset of metrics, the literature comprises hundreds of other metrics, which regardless of the existing literature reviews surveying them, a consolidated overview is still needed to understand their properties and leverage their potential to align with developers' cognitive load. This need is addressed in this paper through a Systematic Tertiary Review (STR) covering the full spectrum of source-code metrics and studying their properties across several dimensions. The outcome of this STR is intended to provide a consolidated overview of source-code metrics, guide practitioners in choosing the appropriate ones and raise new research challenges to be addressed in the future.

Keywords: Source-code metrics, software quality, source-code readability, cognitive load

1. Introduction

In today's connected world, software systems are everywhere. Powered by millions of lines of code, these systems support almost every aspect of our life [1]. The development of software systems underlies a series of implementation, comprehension, extension and maintenance tasks [2]. The difficulty of these tasks depends on several factors, notably the characteristics of the source-code on which they are applied [3]. These characteristics are typically captured and measured using source-code metrics, providing a mapping between different quality attributes (e.g., *understandability, flexibility and maintainability*) and numerical

*Work supported by the International Postdoctoral Fellowship (IPF) Grant (Number: 1031574) from the University of St. Gallen, Switzerland.

*Corresponding author

Email address: amine.abbad-andaloussi@unisg.ch (Amine Abbad-Andaloussi)

values indicating the extent to which the source-code possess these attributes and supports the tasks whose difficulty relies on them [1, 4–7].

From a cognitive perspective, the difficulty of software development tasks can be associated with the cognitive load theory [8], which posits that humans' working memory has a limited capacity allowing it to accommodate a limited amount of cognitive load (i.e., the load imposed on the memory while performing a mentally demanding task [8]). When dealing with complex artifacts (e.., source-code), this limit is rapidly approached, which in turn, manifests in increased difficulty requiring humans to invest more effort in order to keep a constant performance [8–10]. Cognitive load can be captured using a wide array of measures [9], notably, those derived from modalities such as electroencephalogram (EEG), functional magnetic resonance imaging (fMRI), heart rate variability (HRV) and eye-tracking [9, 11–13]. These psycho-physiological measures have shown their robustness in estimating users' cognitive load in many studies within the software engineering field (overview in [14–17]).

Problem description. Grounded in the cognitive load theory, previous research has attempted to relate source-code metrics to developers' cognitive load captured using different psycho-physiological measures [18–22] in order to investigate the extent to which these metrics can estimate the difficulty of software development tasks. However, the majority of the source-code metrics (e.g, Lines of Code (LOC), Mccabe's Cyclomatic Complexity (CC) [23], Halstead's metric suite [24]) investigated in this context, could not be associated with cognitive load. Delving into these empirical studies, one could notice that only a small subset of (popular) metrics has been covered. Moreover, these metrics may not be representative of the body of source-code metrics that have emerged in the literature.

Indeed, the literature comprises a large spectrum of source-code metrics capturing a wide array of attributes [7, 25, 26] at different levels of source-code granularity (e.g., method, class, package) [27, 28]. These metrics have been organized in different categories based on a plenitude of characteristics [29–32] and have been used individually or combined in different settings [25]. Although dozens of literature reviews surveying source-code metrics exist, each of them focuses on specific quality attributes [33–36], type of metrics [31, 37] and language paradigm [25, 38]. In the absence of a global and consolidated literature review on source-code metrics, it is still difficult to develop a holistic overview allowing to understand their properties and leverage their potential to align with developers' estimates of cognitive load. Such an alignment would demonstrate the ability of source-code metrics to measure the difficulty of software development tasks from a cognitive perspective grounded in the theory of cognitive load.

Contributions. To address the need for a holistic overview on source-code metrics, this paper proposes a Systematic Tertiary Review (STR) exploring the existing literature reviews and engaging with the wide spectrum of existing source-code metrics to provide a detailed analysis of their properties. Throughout

this work, source-code metrics are organized and investigated across several dimensions, allowing for their similarities and disparities to emerge and thus shedding the light on their characteristics. Overall, the contributions of this work (C1-C3) can be formulated as follows:

- C1: Provide a global and consolidated overview of the literature on source-code metrics by conducting an STR study covering the literature reviews that have emerged within this topic.
- C2: Synthesize the findings of the STR study into a conceptual framework emphasizing the key notions identified through the study and capturing their relationship with each other.
- C3: Based on the synthesis of the literature findings, identify and discuss the research gaps in the literature to investigate the misalignment observed in the past empirical studies between source-code metrics and developers' cognitive load. Then, provide a research agenda delineating the key directions for future work.

Structure and overview of the article. This paper is structured as follows. Section 2 provides a background of the notions and theories discussed in this tertiary review. In particular, the notions of software quality, attributes and measurements are defined. Moreover, the cognitive load theory is introduced. Section 3 presents the research method followed to conduct this STR study. In line with the existing guidelines, this section explains the procedure allowing to reduce potential biases during the selection of relevant studies and ensure the reproducibility of the reported results. Section 4 reports the findings of the STR study. In particular, the existing literature reviews on source-code metrics are identified and the common research questions addressed by these studies are highlighted. Following that, different classes of metrics are investigated across several dimensions. Moreover, insights are provided on the machine learning approaches combining several metrics to predict particular quality attributes. Furthermore, the tools and projects covering source-code metrics are surveyed. Section 5 discusses the STR findings. Along this discussion, a conceptual framework bringing all the pieces of the findings together is proposed. In addition, the gaps in the literature are highlighted and a research agenda is suggested to address them. Section 6 lists the aspects threatening the validity of this research. Finally Section 7 concludes the paper.

2. Background

This section introduces the key concepts discussed throughout this study. Section 2.1 provides a background on software quality and measurement, while Section 2.2 explains the theory of cognitive load.

2.1. Software quality and measurement

According to the IEEE Software Engineering Glossary [4], *Quality* is defined as the *"degree to which a system, component, or process meets specified requirements"*. The terms "system", "component" and

"process" refer to instances of software artifacts. Other examples of software artifacts could be "source-code" and "conceptual models". A quality attribute, in turn, is defined in the IEEE Glossary [4] as a *"feature or characteristic that affects an item's* [software artifact] *quality"*. Examples of quality attributes are maintainability, understandability and testability. Quality attributes are typically seen as abstract concepts [39]. Hence they cannot be directly used to describe the extent to which they apply to software artifacts.

A *measurement* refers to the process of assigning a number to an attribute in order to describe the degree to which this attribute applies to an artifact [39–41]. In Software Engineering, a measurement is conducted through a *metric* that is a pre-defined *"function whose inputs are software data and whose output is a single numerical value that can be interpreted as the degree to which the software possesses a given quality attribute"* [4]. Notable examples of metrics are Lines of Code (LOC), Cyclomatic Complexity (CC) [23], Halstead's suite [24] and Chidamber and Kemerer suite (*C&K*) [42].

Software measurements provide (at least) three key benefits i.e., *understanding*, *control* and *improvement* of the software artifact [39, 43, 44]. Starting with the first benefit, the output of metrics delivers meaningful information allowing to *understand* the extent to which an attribute (e.g., maintainability) applies to a software artifact (e.g., source-code). Metrics for the *C&K* suite [42] can, for instance, show that a set of classes have low cohesion and high coupling with each other, which can make the source-code hard to maintain in the future. This information enables also *control* as developers can refactor the source-code to raise the cohesion of classes and lower their coupling, which eventually would contribute to *improve* its maintainability.

2.2. Cognitive load theory

The *Cognitive load theory [45]* provides a conceptual base allowing to link the difficulty of software development tasks to developers' cognitive load. To explain the cognitive load theory, it is necessary to differentiate the concepts of (human's) *long term memory* and *working memory* [46–48]. The former refers to a store holding information for a long time (or indefinitely). The latter refers to a buffer storing information temporally to be processed or integrated with other information. The cognitive load theory investigates the properties of the working memory. It defines the concept of *cognitive load* as a *"a multi-dimensional construct representing the load imposed on the working memory during [the] performance of a cognitive task"* [8, 9]. A *cognitive task* (or shortly a task), in turn, refers to a set of *operations* and an *artifact* (e.g., source-code) on which a person should apply these operations. The cognitive load theory postulates that the human working memory has a *limited capacity* [8, 9]. Hence, the amount of information that can be held and processed at the same time is also limited (i.e., typically varying between 7 ± 2 items at a time [49]). Following this theory, humans' working memory may become a bottleneck when performing tasks that require holding and processing more information than the memory can support [8, 9, 45]. In this situation, humans are likely to

experience cognitive overload, which in turn, manifests in increased difficulty requiring them to invest more effort to keep a constant performance and avoid error-prone situations [8–10]. Similarly, when conducting complex software development tasks, developers might experience cognitive overload, which, in turn, would challenge their ability to complete the given tasks and require them to invest more effort in them.

The literature discerns two types of cognitive load, i.e., *intrinsic* and *extraneous*. The former rises from the complexity inherent to the operations and the artifact within the task [9, 45]. During a software development task, requiring, for instance, a set of changes on source-code, intrinsic load would emerge from the complexity inherent to the required changes and the complexity inherent to the software functionalities. *Extraneous load*, in turn, arises from the way the operations and the artifact within the task are represented [9, 45]. In the context of the aforementioned change task, extraneous load would emerge from the complicated formulation of the required changes in the task and the poor representation of the source-code.

3. Methodology for the systematic tertiary review

A Systematic Tertiary Review (STR) (also called meta-review [50]) is a review of existing secondary studies (e.g., literature reviews) summarizing the state-of-the-art literature in a particular field [51]. Kitchenham recommends the same rigorous methodology proposed for Systematic Literature Reviews (SLRs) when conducting STRs [51]. Accordingly, in the literature, many STRs (e.g., [50, 52, 53]) follow Kitchenham SLR guidelines [51] or use similar guidelines (e.g., Webster and Watson [54], Synder [55]). Besides a few disparities at the operational level of the search protocol, both old (e.g, [51]) and recent (e.g, [55]) guidelines highlight the importance of defining research questions, documenting the literature search, filtering and screening the identified articles and reporting the method used to extract the relevant information from the articles. This STR follows Kitchenham's approach, being the popular reference for conducting literature reviews in the software engineering field. An overview of this approach is illustrated in Figure 1. First, the research questions addressed in this STR are formulated (cf. Section 3.1). Then, a pilot search is conducted prior to the main literature search (cf. Section 3.2). Following that, a set of data sources are identified (cf. Section 3.3), the search strings are composed (cf. Section 3.4) and the inclusion and exclusion criteria used to select the relevant literature are defined (cf. Section 3.5). Subsequently, the main literature search is conducted (cf. Section 3.6) and then enriched with a snowballing search (cf. Section 3.7). Once all the relevant literature reviews are identified, a data extraction scheme is defined and used to collect information from the selected reviews (cf. Section 3.8). This information is, afterwards, analyzed following a qualitative approach (cf. Section 3.9).

3.1. Research questions

This work aims at investigating the state-of-the-art literature on source-code metrics. As mentioned in Section 1, many literature reviews covering source-code metrics have emerged. However, each review

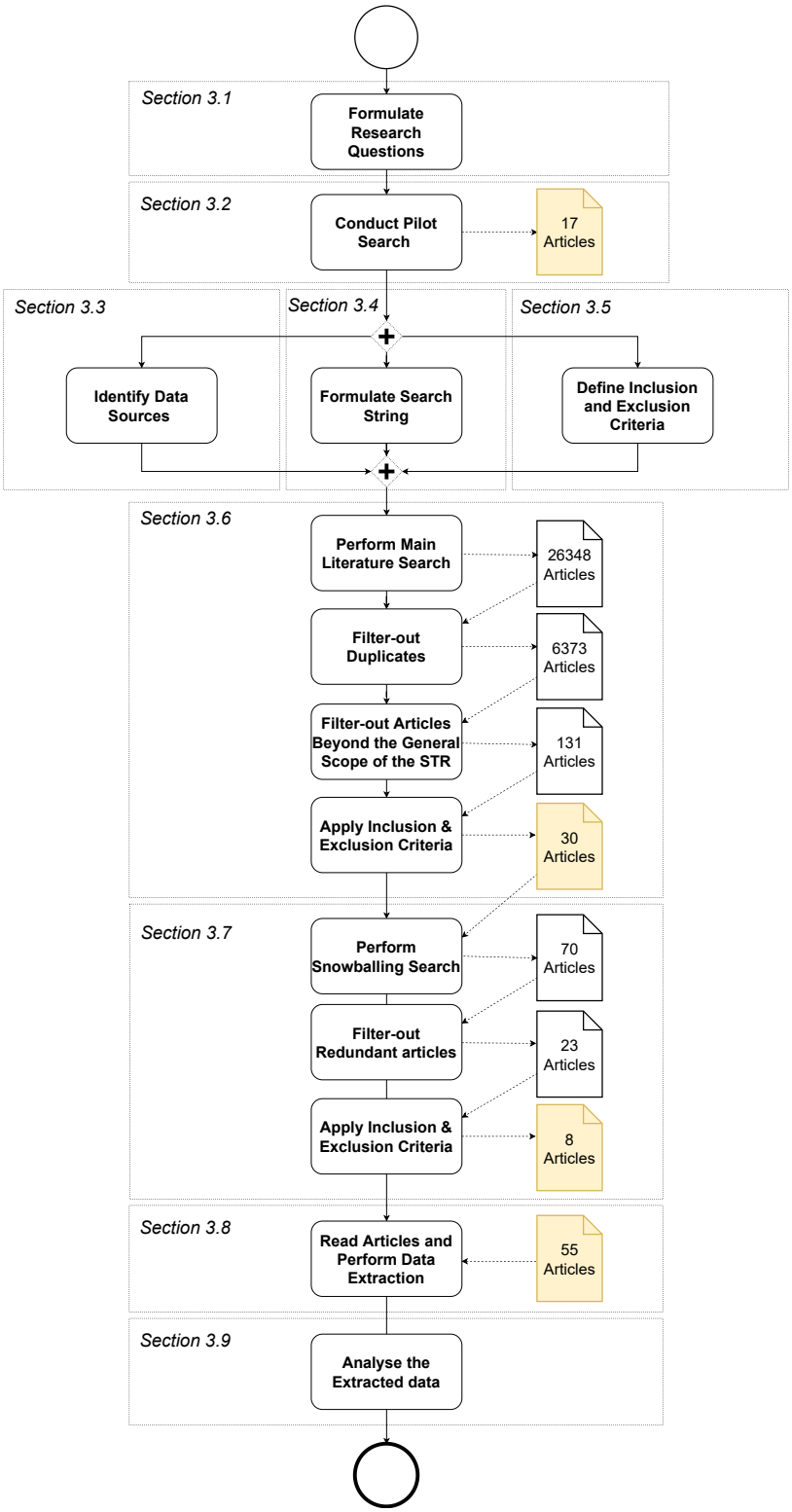


Figure 1: Overview of the STR approach

focuses on a particular quality attribute, type of metrics or language paradigm. To provide a fundamental understanding and holistic overview on source-code metrics, the following research questions are addressed:

- *RQ1: What literature reviews about source-code metrics exist and how are these studies distributed over time, venue, publication type, review method and topic of review?*
  This question allows identifying the literature reviews on source-code metrics and investigating their distribution over the aforementioned dimensions. These insights provide a consolidated overview on the related work and outline the different contexts in which source-code metrics have been investigated.

- *RQ2: What research questions have been addressed in existing literature reviews?*
  This research question highlights the aspects that have received increased attention in secondary studies. These aspects are covered in this STR in a more comprehensive way.

- *RQ3: What is the state of the literature on source-code metrics?*

  - *RQ3.1: What source-code metrics have been proposed in the literature?*
    This research question provides an overview of the different classes of metrics that have been identified by the existing literature reviews. Therein, a wide spectrum of existing source-code metrics is explored and not only the popular ones.

  - *RQ3.2: At what levels of granularity are the different source-code metrics calculated?*
    This research question investigates the metrics identified in RQ3.1 to develop an overview on the different (source-code) granularity levels at which they can be applied.

  - *RQ3.3: What quality attributes are captured by existing source-code metrics?*
    This research question explores the different facets of quality that can be measured using the metrics proposed in the literature. These facets can be organized into internal quality attributes (i.e., focusing on the structural properties of the source-code [56]) and external quality attributes (i.e., emphasizing the quality of the source-code as perceived by its stakeholders [56]).

  - *RQ3.4: How have the existing source-code metrics been categorized in the literature?*
    The answer to this research question provides an overview of the different categorizations of source-code metrics that have been mentioned in the past literature reviews. This overview goes beyond the aforementioned groupings (i.e., by granularity level and quality attribute, cf. RQ3.2 and RQ3.3 respectively) to discover other possible categorizations of metrics.

  - *RQ3.5: What metrics have been validated?*
    This research question gives insights on the validations of existing metrics and their outcome. This, in turn, reveals the metrics that are most likely to capture the different quality attributes of source-code accurately.

- *RQ4: What metric-based approaches have been proposed in the literature to evaluate the quality of source-code?*

  This question goes beyond the use of individual source-code metrics to cover machine learning approaches where several metrics are combined to deliver models that can make accurate estimations of different quality attributes of source-code.

- *RQ5: What tools and projects are popular to extract source-code metrics?*

  This question provides insights on the tools that have been commonly cited in existing literature reviews. These tools can make the extraction of metrics straightforward. In addition, this research question aims at reporting the popular projects covering source-code metrics. These projects, in turn, can be used to benchmark metrics and models meant to assess the quality of source-code.

3.2. Pilot search

A pilot search was conducted before the main literature search. Some of the articles covered in this phase were identified in the bibliography of notable scientists in the field of software engineering (e.g., [57]), while other articles were found through simple search queries (e.g., software metrics). Overall the pilot search resulted in the identification of 17 article covering source-code metrics (cf. Table 4 – Source: P).

3.3. Data sources

The data sources covered by the literature search are the following: ACM Digital Library, IEEE Xplore, Science Direct, Wiley InterScience, Scopus and Springer. These data sources have been also used by the majority of the secondary studies identified in the pilot search.

3.4. Search strings

The keywords used in the literature search are shown in Table 1. Most of them are derived from the studies identified in the pilot search as reported in Table 2. Moreover, an additional set of keywords was considered (i.e., "program", "code lexicon", "semantic", "code formatting") to ensure a wider coverage of the literature. The keyword "program" is usually interchanged in the literature with the keywords "software" and "source-code". Regarding the keywords "code lexicon" and "semantic", they have been used in primary studies investigating the quality of the source-code identifiers and comments [58–61]. As for the keyword "code formatting", it was used in a set of studies investigating the indentation and the ordering of source-code elements [59, 62]. All these factors have been shown to affect the readability of the source-code [60–63]. Overall, the keywords extracted from the literature (cf. "Keyword Set 1" in Table 1) are composed from general terms such as software, source-code, program and quality metrics in addition to a number of properties associated with software complexity, maintainability, understandability and fault-proneness. Lastly, to ensure the coverage of a broad range of literature reviews, keywords (cf. "Keyword Set

2" in Table 1) targeting the secondary studies in the literature were added (i.e., literature review, literature survey, mapping study)

Based on the keywords in Table 1, a number of search strings were composed from the "AND" union of the terms in the first and second sets. These search strings were used systematically to run the literature search on the covered data sources (cf. Section 3.3).

Keywords Set 1	software metrics, program metrics, code metrics, source-code metrics, source code metrics, quality metrics, complexity metrics, fault metrics, error metrics, bug metrics, defect metrics, flaw metrics, change metrics, maintenance metrics, maintainability metrics, understandability metrics, comprehension metrics, readability metrics, code smell metrics, antipattern metrics, technical debt metrics, semantic metrics, code lexicon metrics, code formatting metrics
Keywords Set 2	literature review, literature survey, mapping study

Table 1: Sets of keywords using during the literature search

3.5. Inclusion and exclusion criteria

Sections 3.5.1 and 3.5.2 present the inclusion and exclusion criteria used to guide the selection of the articles collected during the literature search. Note that the term "literature review" refers to all types of secondary studies providing a review of the literature (i.e., SLRs, Systematic mapping Studies – SMSs [51]).

3.5.1. Inclusion criteria

A study is included if one of the following criteria apply:

- **IC1**: Literature review covering primary studies proposing source-code metrics.
- **IC2**: Literature review covering primary studies proposing approaches to evaluate the quality of source-code using metrics.
- **IC3**: Literature review covering primary studies proposing tools to derive source-code metrics.
- **IC4**: Literature review covering empirical studies validating existing source-code metrics.

3.5.2. Exclusion criteria

A study is excluded if one of the following criteria apply:

- **EX1**: Study not published in English.
- **EX2**: Study not addressing the quality of source-code.

Study	Derived keywords
What's up with software metrics? – A preliminary mapping study [57]	software, metrics
A Tool-Based Perspective on Software Code Maintainability Metrics: A Systematic Literature Review [64]	source code, maintainability
Measurement in Software Engineering: From the Roadmap to the Crossroads [29]	quality
Software Source Code Readability [65]	readability
A Systematic Review Measurement in Software Engineering: State-of-the-Art in Measures [66]	maintenance
A Systematic Literature Review on Empirical Analysis of the Relationship Between Code Smells and Software Quality Attributes [26]	code smell, antipattern
Predicting Change Using Software Metrics: A Review [35]	change
Source code metrics: A systematic mapping study [32]	fault, complexity, understandability, comprehension
Software defect prediction using bad code smells: A systematic literature review [67]	bug, flaw, defect, error
A systematic literature review on the detection of smells and their evolution in object-oriented and service-oriented systems [68]	technical debt

Table 2: Origin of the keywords composing the search string

- **EX3**: Study not covering source-code metrics.
- **EX4**: Primary study covering source-code metrics.
- **EX5**: Literature review on approaches assessing the quality of source-code without metrics.
- **EX6**: Literature review on factors or guidelines affecting the quality of source-code.
- **EX7**: Work-in-progress or duplicated articles (i.e., using the same data).
- **EX8**: Literature review focusing on metrics (e.g., product line metrics [69]) requiring software artifacts beyond the source-code are excluded. Metrics operating at the level of the code interfaces (e.g., interface metrics [70]) and metrics taking the evolution of source-code into account (e.g., code change metrics [71]) are still covered.

3.6. Main literature search and selection process

The main literature search yielded 26348 articles. After filtering out the duplicate ones, 6373 potentially relevant articles remained. Such a high number of papers is typically found in the first search phases

of literature reviews in the Software Engineering field (e.g, [32, 72–74]). The main literature search and selection of articles were conducted by the author of this paper following the inclusion and exclusion criteria defined in Section 3.5. These criteria have been carefully defined, discussed with colleagues in the software engineering field and refined to ensure a good coverage of the relevant literature. In addition, borderline articles were discussed with the author's colleagues prior to their inclusion or exclusion.

The selection of relevant articles has been conducted in a systematic manner. Firstly, the meta-data (title, authors, venue, type of venue, year of publication, abstract, keywords) of the articles returned by the search engines of the different data sources were downloaded and organized in a spreadsheet. The selection process began by excluding the articles with titles that are clearly beyond the general scope of this tertiary review. This step allowed to exclude 6243 articles. Thereafter, the abstracts and keywords of the remaining 131 articles were read and the non-relevant articles were excluded following the inclusion and exclusion criteria defined in Section 3.5. While the abstract and keywords of some articles provided a clear overview of their content and allowed to decide on their inclusion or exclusion, other articles had to be fully read before being assessed. Overall, 36 articles were selected during this phase (cf. Table 4 – Source: M). As 6 of these articles were already identified in the pilot search, the final number of articles resulting from this phase was 30. The list of the selected articles mapped to the inclusion and exclusion criteria is available online[1].

3.7. Snowballing search

A snowballing search has been conducted over the articles selected from the main literature research. Overall, 70 potentially relevant articles were found during this phase. After removing the duplicate articles (in this case 24 articles) and those already covered by the pilot or the main literature search (in this case 23 articles), 23 articles have gone through the aforementioned inclusion and exclusion criteria (cf. Section 3.5), of which 8 were selected as relevant for the STR (cf. Table 4 – Source: S).

3.8. Data extraction scheme

The literature reviews identified during the pilot, main and snowballing search phases provide rich insights on source-code metrics. To extract them, a data extraction scheme was developed following the structure presented in Table 3. The attributes and the sub-attributes in this scheme were defined with the aim to provide the necessary information allowing to answer the research questions formulated in Section 3.1.

3.9. Analysis procedure

During the analysis, a qualitative coding approach based on *grounded theory* [75] was followed to group the concepts that are common across several reviews and provide a holistic understanding of the state-of-the-art literature. Overall, three coding techniques were used: (1) *Initial coding* [75] allowed to highlight

[1]See `http://andaloussi.org/JSS2022/search/`

RQ	Attribute	Sub-Attributes
RQ1	Article meta-data	Title
		Authors
		Publication year
		Publication type
		Publication venue
		Keywords
	General review info.	Review method (SLR, SMS)
		Topic of review
RQ2	Research questions	Question
		Short answer
RQ3.1	Metrics	List of metrics
RQ3.2	Granularity levels	Level of granularity
		Examples of metrics
RQ3.3	Quality attributes	Attribute
		Type
		Example of metrics
RQ3.4	Categorizations	Types of categorization
		Example of metrics
RQ3.5	Metrics Evaluations	Evaluated metric
		Quality attribute
		Results
RQ4	Approaches based on metrics	List of approaches
RQ5	Tools	Tool name
		Link
		Academic or commercial
		Delivers metrics or code smells
		Languages supported
		Eg., metrics extracted
	Projects	Project name
		Link
		Type
		Language

Table 3: Data Extraction Scheme

the salient aspects in the review articles. (2) *Focused coding* [75] enabled to group these aspects based on their similarities. (3) *Axial coding* [75] allowed to establish the relationships between the grouped aspects. These coding techniques were used at different stages of the analysis to address several research questions.

In particular, to identify the topics covered by existing literature reviews (part of RQ1.1), initial codes were assigned to each review article based on its meta-information (i.e., title, abstract and keywords). Afterward, focused coding was applied to group the articles covering the same research topic. To develop a clear overview of the research questions investigated in the literature (RQ2), several rounds of initial, focused and axial codings were conducted based on the research questions and answers recorded in the data extraction scheme. As a result, existing research questions were grouped into themes and then organized into categories. Moreover, for each category, a question capturing the key aspects addressed in the literature was phrased. A similar coding was performed to address the remaining questions (RQ3, RQ4 and RQ5) where existing metrics, approaches, tools and projects were analyzed.

Last but not least, the findings of this STR were synthesized into a conceptual framework. The development of this conceptual framework was driven by axial coding. Following this qualitative coding approach, it was possible to discover how the different concepts identified throughout the study relate to each other.

4. Findings

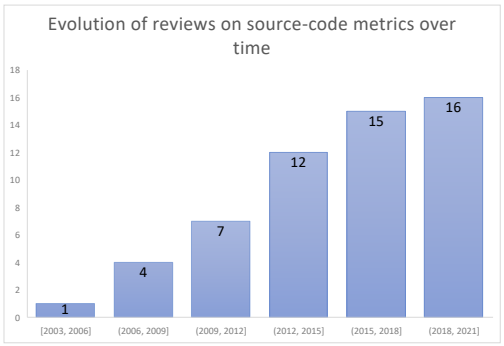
This section reports the findings of this STR. Section 4.1 identifies the existing literature reviews and shows their distribution over time, venue, publication type, review method and topic of review. Section 4.2 presents the research questions that have been addressed by the existing literature reviews. Section 4.3 provides an overview of the state-of-the-art literature on source-code metrics. Section 4.4 discerns the metric-based approaches allowing to evaluate the quality of source-code. Section 4.5 identifies the popular tools and projects to extract source-code metrics.

4.1. What literature reviews about source-code metrics exist and how are these studies distributed over time, venue, publication type, review method and topic of review? (RQ1)

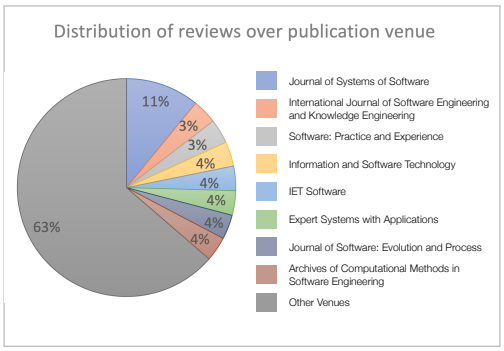
The search protocol allowed the identification of 55 literature reviews covering source-code metrics. These reviews are reported in Table 4. The distribution of review articles over time depicted in Figure 2a shows that source-code metrics have been reviewed since 2003 with an increasing trend over the last two decades. Interestingly, the last three years denote the period with the highest number of reviews on source-code metrics. Figure 2b provides an overview of the venues where the past reviews have been published. Overall, 43 venues are identified among which *"Journal of Systems and Software"* is the most popular (i.e., 6 papers, 11%), followed by *"International Journal of Software Engineering and Knowledge Engineering"*, *"Software: Practice and Experience"* (journal), *"Information and Software Technology"* (journal), ‘‘*IET Software"* (journal), *"Expert Systems with Applications"* (journal), *"Journal of Software: Evolution and Process"* and *"Archives of Computational Methods in Engineering"* (journal) (i.e., 2 papers, 3% each). Figure 2c shows that the majority of reviews (i.e., 32 papers, 58%) have been published in journal venues. Nevertheless, a significant number of reviews (i.e., 22 papers, 40%) have appeared in conference venues.

Title	Ref.	Source
Software Product Quality Metrics: A Systematic Mapping Study	[28]	M
Software smell detection techniques: A systematic literature review	[76]	M
Can we benchmark Code Review studies? A systematic mapping study of methodology, dataset, and metric	[72]	M
Reusability affecting factors and software metrics for reusability: A systematic literature review	[73]	M
A Systematic Literature Review on Empirical Analysis of the Relationship Between Code Smells and Software Quality Attributes	[26]	P
Code smells detection and visualization: A systematic literature review	[77]	P
Software Source Code Readability	[65]	P
Software Defect Prediction Using Bad Code Smells: A Systematic Literature Review	[67]	P
A Tool-Based Perspective on Software Code Maintainability Metrics: A Systematic Literature Review	[64]	P
Quality and Success in Open Source Software: A Systematic Mapping	[78]	M
Impact of Design Patterns on Software Quality: A Systematic Literature Review	[6]	M
A survey on Software Coupling Relations and Tools	[79]	M
Quality Metrics in Software Design: A Systematic Review	[80]	M
Code Smells Enabled by Artificial Intelligence: A Systematic Mapping	[81]	M
Software Quality Assessment Model: A Systematic Mapping Study	[82]	M
Software Design Smell Detection: A Systematic Mapping Study	[74]	M
Construction of a Software Measurement Tool Using Systematic Literature Review	[83]	M
Progress on Approaches to Software Defect Prediction	[84]	M
Coupling and Cohesion Metrics for Object-Oriented Software: A Systematic Mapping Study	[85]	M
Early Software Defect Prediction: A Systematic Map and Review	[86]	M
A Systematic Literature Review on the Detection of Smells and their Evolution in Object-Oriented and Service-Oriented Systems	[68]	P, M
Software Metrics for Fault Prediction Using Machine Learning Approaches: A Literature Review with PROMISE Repository Dataset	[87]	M
A Literature Review on Cross Project Defect Prediction	[88]	M
A Systematic Review of Software Usability Studies	[89]	M
A Mapping Study on Design-time Quality Attributes and Metrics	[7]	M
Source Code Metrics: A Systematic Mapping Study	[32]	P, M
Maintenance Effort Estimation for Open Source Software: A Systematic Literature Review	[90]	M
Metrics and Statistical Techniques Used to Evaluate Internal Quality of Object-oriented Software: A Systematic Mapping	[91]	M
Systematic Mapping Study of Metrics Based Clone Detection Techniques	[92]	M
A Review-Based Comparative Study of Bad Smell Detection Tools	[93]	M
Software Code Maintainability: A Literature Review	[94]	P
A systematic Review of Machine Learning Techniques for Software Fault Prediction	[95]	S
Reusability Metrics of Software Components: Survey	[5]	S
Software Fault Prediction: A Systematic Mapping Study	[96]	S
Predicting Change Using Software Metrics: A Review	[35]	P
A Review of Code Smell Mining Techniques	[97]	M
Empirical Evidence on the Link Between Object-oriented Measures and External Quality Attributes: A Systematic Literature Review	[25]	S
A Report on the Analysis of Metrics and Measures on Software Quality Factors – A Literature Study	[98]	S
Open Source Tools for Measuring the Internal Quality of Java Software Products. A survey	[99]	M
Software Fault Prediction Metrics: A Systematic Literature Review	[34]	P, M
A Systematic Review of the Empirical Validation of Object-Oriented Metrics towards Fault-proneness Prediction	[100]	S
A Mapping Study to Investigate Component-based Software System Metrics	[37]	M
Empirical Evidence of Code Decay: A Systematic Mapping Study	[36]	M
A Systematic Mapping Study on Dynamic Metrics and Software Quality	[31]	M
A Systematic Literature Review on Fault Prediction Performance in Software Engineering	[30]	P
A Systematic Review on the Impact of CK Metrics on the Functional Correctness of Object-Oriented Classes	[101]	S
Aspect-oriented Software Maintenance Metrics: A Systematic Mapping Study	[102]	P, M
Software Fault Prediction: A Literature Review and Current Trends	[103]	M
Coupling Metrics for Aspect-Oriented Programming: A Systematic Review of Maintainability Studies	[38]	M
What's Up with Software Metrics? – A Preliminary Mapping Study	[57]	P, M
A Systematic Review of Software Fault Prediction Studies	[27]	P
A Systematic Review of Software Maintainability Prediction and Metrics	[33]	S
Measurement in Software Engineering: From the Roadmap to the Crossroads	[29]	P
A Systematic Review Measurement in Software Engineering: State-of-the-Art in Measures	[66]	P, M
Product Metrics For Object-oriented Systems	[104]	P

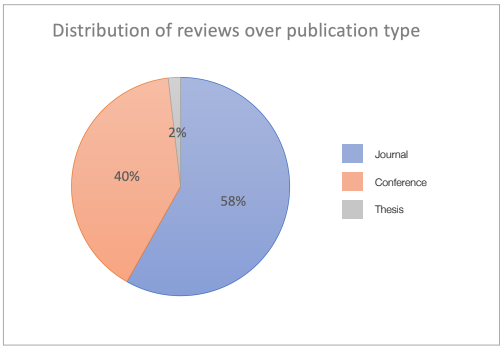
Table 4: Literature reviews selected during the pilot, main literature search and snowballing. Abbreviations: Ref.: Full Reference, J: Journal, C: Conference, T: Thesis, M: Main literature Search, P: Primary Search, S: Snowballing search



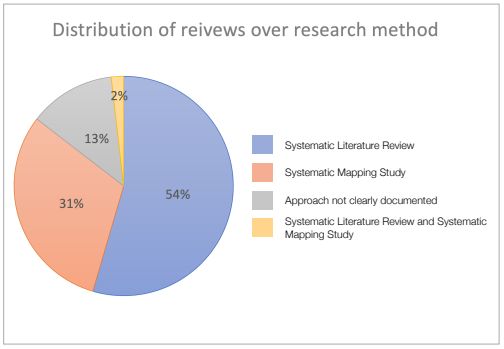
(a) Distribution of existing reviews over time.



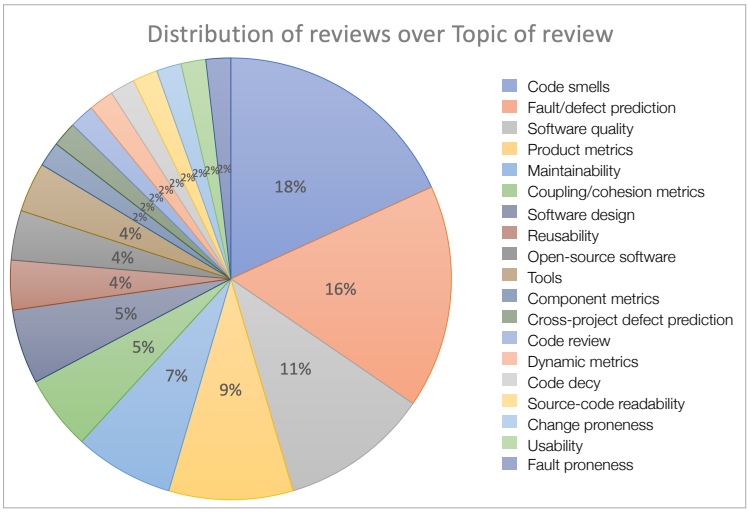
(b) Distribution of reviews over publication venue



(c) Distribution of reviews over publication type



(d) Distribution of reviews over research method



(e) Distribution of reviews over topic of review

Figure 2: Different distributions of the reviews in Table 4

When it comes to the distribution over research method, Figure 2d shows that the majority of reviews used an SLR approach [51] (i.e., 30 papers, 54%), while a smaller portion of studies (i.e., 17 papers, 31%) followed an SMS approach [51].

Last but not least, the distribution of articles over the topic of review depicted in Figure 2e shows that the topic of code smells (i.e., 10 papers, 18%) is the most popular subject reviewed in the literature, followed by fault prediction techniques (i.e., 9 papers, 16%), software quality (i.e., 6 papers, 11%) product metrics (i.e., 5 papers, 9%), and software maintainability (i.e., 4 papers, 7%). The full list of the identified topics is shown in Figure 2e.

4.2. What research questions have been addressed in existing literature reviews? (RQ2)

The literature reviews identified in this STR study address a wide array of research questions. Following the qualitative approach described in Section 3.9, these research questions were grouped into themes and categories. Then, a representative research question was appended to each category. Overall, as shown in Table 5, 4 themes (i.e., *metrics*, *approaches*, *tools* and *projects*) and 25 categories were identified. In the following (cf. Sections 4.3–4.5 corresponding to the answers to RQ3–RQ5), a consolidated overview is provided on the most salient questions within each theme (i.e., having high occurrence numbers within the theme as marked in bold in Table 5). Unlike the existing literature reviews, the new answers to these questions embrace the whole spectrum of research topics, quality attributes, types of metrics and language paradigms, which have been investigated previously in separate studies.

4.3. What is the state of the literature on source-code metrics? (RQ3)

Based on the findings of all the covered literature reviews, this section identifies the existing metrics (cf. Section 4.3.1), reports the levels of granularity at which they are calculated (cf. Section 4.3.2), identifies the captured quality attributes (cf. Section 4.3.3), enumerates the different categorization proposed for existing metrics (cf. Section 4.3.4) and explores the results of the empirical studies testing them (c. Section 4.3.5).

4.3.1. What source-code metrics have been proposed in the literature? (RQ3.1)

The analysis of the literature reviews covering source-code metrics yielded the identification of 855 metrics[2]. These metrics belong to different suites and have been proposed in different contexts. Table 6 presents a set of representative metric suites identified in the literature, while Table 7 groups them into a set of classes based on their context.

Basic and object-oriented metrics. Among the basic (and oldest) metrics proposed for source-code written in procedural and structured languages are LOC and McCabe's Cyclomatic Complexity (CC) [23]. Following these metrics comes the Halstead's suite [24]. With the rise of Object-oriented (OO) programming, new suites such as the Chidamber and Kemerer C&K suite [42], Li and Henry suite [106], Lorenz and Kidd suite [107], Metrics for Object-oriented Design (MOOD) suite [108] and Quality Metrics for

[2]The list of metrics is available online at See `http://andaloussi.org/JSS2022/metrics.xlsx`

Theme	**Category**	**Key questions**	**#**	References
Metrics	Lists	What source-code metrics have been proposed in the literature?	**38**	[5, 6, 25–27, 31–38, 64, 65, 67, 68, 73, 76, 78, 80, 83, 85, 86, 88–92, 94, 95, 97–100, 102, 104, 105]
Metrics	Validations	What metrics have been validated?	**16**	[7, 25, 33–38, 57, 65, 66, 73, 85, 94, 100, 101]
Metrics	Categorizations	How have these metrics been categorized in the literature?	**10**	[29–32, 66, 72, 78, 79, 82, 90]
Metrics	Attributes	What quality attributes are captured by existing source-code metrics?	**9**	[5–7, 25, 26, 73, 78, 85, 98]
Metrics	Granularity	At what granularity levels can the different source-code metrics be calculated?	**5**	[27, 28, 37, 38, 73]
Metrics	Development phases	What source-code metrics are extracted on each of the software development phases?	4	[7, 33, 34, 66]
Metrics	Studies contexts and applications	Which studies and applications are carried by means of source-code metrics?	3	[31, 32, 85]
Metrics	Thresholds	Are there known thresholds for the existing source-code metrics?	2	[28, 35]
Metrics	Range	What range of values do these metrics show and what do they mean?	1	[80]
Metrics	Paradigm	Which programming paradigms are currently measured by source-code metrics?	1	[32]
Metrics	Relation to cognitive complexity	How do some metrics relate to the concept of cognitive complexity?	1	[100]
Metrics	Trends	Are new source-code metrics being proposed?	1	[32]
Metrics	Directions for future work	Which aspects of dynamic metrics could be recommended as topics for future research?	1	[31]
Approaches	Lists	What approaches have been proposed in the literature to evaluate the quality of source-code?	**22**	[25, 27, 28, 30, 35, 36, 68, 74, 81, 82, 84, 86–88, 90–92, 95, 96, 100, 103, 105]
Approaches	Validations	How have these approaches been validated and what are the results of these validations?	13	[26, 30, 33–35, 82, 84, 87, 90, 95, 96, 103, 105]
Approaches	Issues	What are the challenges of the existing approaches?	2	[81, 82]
Tools	Lists	What tools are extracting source-code metrics?	**12**	[7, 28, 32, 64, 73, 74, 79, 82, 83, 92, 93, 97]
Tools	Categorizations	How have these tools been organized in the literature?	2	[64, 82]
Tools	Features	What are the features of these tools?	2	[83, 93]
Tools	Maturity	What is the degree of maturity of these tools?	1	[99]
Projects	Lists	What projects have been used in studies covering source-code metrics?	**6**	[26, 32, 84, 87, 90, 95]
Projects	Categorizations	How have these projects been categorized in the literature?	4	[27, 32, 34, 35]
Projects	Size	What is size of the projects used in studies covering source-code metrics?	2	[34, 90]
Projects	Languages	In what programming languages are these projects?	2	[32, 34]
Projects	Fault distribution	What is the fault distribution in the datasets used for software fault prediction?	1	[95]

Table 5: Research questions investigated in the literature reviews. Abbreviations: #: number of reviews addressing this category of questions

Object-oriented Design (QMOOD) suite [109] have emerged (cf. Table 6). Several variants of these basic and object-oriented metrics have been proposed and covered by almost all the literature reviews identified in this STR, which in turn, hints toward their popularity in the literature. Nonetheless, there exist other classes of relevant metrics that have been identified in fewer reviews. These classes are presented in the following paragraphs.

Component metrics. The need to build larger, robust and more complex software systems in a shorter time and at a reduced cost has led to the emergence of component-based software systems. These systems are based on the assembly and integration of small software units (i.e., components) into larger ones [110]. While such an architecture is scalable and can support better reusability, the evaluation of software components is challenging due to their black-box nature as their internal structure is usually hidden and instead only public interfaces are provided [37]. Many of the existing procedural and OO metrics are therefore inapplicable to components. Alternatively, component metrics are used to assess the quality of interfaces and the cost of their integration. Examples of these metrics are those proposed by Narasimhan and Hendradjaya [111] (cf. Table 6).

Slice-based metrics. Another branch of metrics is based on the program slicing technique proposed by Weiser [112, 113]. The technique itself consists of decomposing the source-code into chunks (i.e., slices) representing independent units of code that preserve their control-flow and data-flow dependencies [113]. The slice-based metrics, in turn, are computed at the level of the code chunks. These metrics can be used to narrow down the focus and provide a quality assessment that is specific to the parts of the code that are relevant to infer a certain feature of interest [113, 114]. In [112], Weiser introduced 5 slice-based metrics. This set was extended by two other metrics proposed by Ott and Thuss [115] (cf. Table 6).

Suite	**Metrics**
Halstead's suite [24]	Program vocabulary (η), length (N), volume (V), difficulty (D), effort (E)
Chidamber and Kemerer (C&K suite) [42]	Weighted Methods Per Class (WMC), Depth of Inheritance Tree (DIT), Number of Children (NOC), Response for a class (RFC), Coupling Between Objects (CBO), Lack of Cohesion in Methods (LCOM)
Li and Henry suite [106]	Number of Methods ($NOM_{L\&H}$), Message Passing Coupling (MPC), Data Abstraction Coupling (DAC), Number of Semi-columns Per Class (Size1), Number of Methods Plus Number of Attributes (Size2)

Lorenz and Kidd [107]	Number of Public Methods (NPM), Number of Methods (NM), Number of Friends of a Class (NF), Number of Methods Inherited by a subclass (NMI), Average Method Size (AMS)
Metrics for Object-oriented Design (MOOD) suite [108]	Coupling Factor (CF) Method Hiding Factor (MHF), Attribute Hiding Factor (AHF), Method Inheritance Factor (MIF), Attribute Inheritance Factor (AIF), Polymorphism Factor (AF), Clustering Factor (CluF), Reuse Factor (RF)
Quality Metrics for Object-oriented Design (QMOOD) suite [109]	Design Size in Classes (DSC), Number of Hierarchies (NOH), Average Number of Ancestors (ANA_{QMOOD}), Data Access Metric (DAM), Direct Class Coupling (DCC), Cohesion Among Methods of class (CAM), Measure of Aggregation (MOA), Measure of Functional Abstraction (MFA), Number of Polymorphic Methods (NOP), Number of Methods (NOM_{QMOOD})
Narasimhan and Hendradjaya metrics [111]	Component Packing Density (CPD), Component Interaction Density (CID), Component Incoming Interaction Density (CIID), Component Outgoing Interaction Density (COID), Component Average Interaction Density (CAID), Criticality metrics (CRIT), Number of Cycle (NC), Average Number of Active Component (ANAC), Active Component Density (ACD), Average Active Component Density (AACD), Peak Number of Active Components (PNAC)
Weiser metrics [112]	Slices' length to the length of the entire program (Coverage), number of statements covered by more than one slice (Overlap), extent to which the statements of a slice are contiguous in space (Clustering), number of slices having few statements in common (Parallelism), number of statements present in every slice (Tightness)
Ott and Thuss metrics [115]	Coverage of the shortest slice (MinCoverage), Coverage of the longest slice (MaxCoverage)
Misra et al. suite [116]	Method Complexity (MC), Coupling Weight for a Class (CWC), Attribute Complexity (AC), Class Complexity (CLC), Code Complexity (CC), Average Method Complexity (CC), Average Method Complexity per Class (AMCC), Average Class Complexity (ACC), Average Coupling Factor (ACF), Average Attribute Per Class (AAC)

Scalabrino et al. metrics [117]	Comments and Identifiers Consistency (CIC), Identifier Terms in Dictionary (ITID), Narrow Meaning Identifiers (NMI), Textual Coherence (TC), Comments Readability (CR), Number of Meanings (NM)
Buse and Weimer metrics [118]	Length of source-code lines and identifiers, count of identifiers, keywords, digits, comments, periods, commas, spaces, parenthesis, arithmetic operators, comparison operators, assignments, branches, loops, blank lines; occurrences of single characters and identifiers, indentation in the source-code
Dorn [119]	Large set of structural, syntactical, visual, perceptional and alignment features. Details in [119].
Taba et al. metrics [120]	Average Number of Antipatterns ($ANA_{\text{Taba et al.}}$), Antipattern Complexity Metric (ACM), Antipattern Recurrence Length (ARL), Antipattern Cumulative Pairwise Differences (ACPD)
Moser et al. suite [71]	Number of revisions of a file (REVISIONS), number of times a file has been refactored (RFACTORINGS), sum over all revisions of the lines of code added to a file (LOC_ADDED), maximum number of lines of code added for all revisions (MAX_LOC_ADDED), average lines of code added per revision (AVE_LOC_ADDED), sum over all revisions of the lines of code deleted from a file (LOC_DELETED), maximum number of lines of code deleted for all revisions (MAX_LOC_DELETED), average lines of code deleted per revision (AVE_LOC_DELETED), sum of (added lines of code – deleted lines of code) over all revisions (CODECHURN), maximum CODECHURN for all revisions (MAX_CODECHURN), average CODECHURN per revision (AVE_CODECHURN)

Table 6: Representative metric suites

Cognitive metrics. Another class of metrics (i.e., cognitive metrics) aims at capturing the mental effort required for a human to make sense of the encoded semantics. A notable example in this vein is the *Cognitive Functional Size (CFS)* metric proposed by Shao and Wang [121]. Therein, the authors assumed three factors affecting the complexity of a program i.e., the number of inputs, the internal complexity of the source-code and the number of outputs [121]. The internal complexity of the source-code, in turn, is based on a set of cognitive weights assigned to the different control-flow structures of the source-code (e.g., sequence, branching, iterations, embedded component, concurrency) based on their presumed difficulty. For instance, a for-do iteration is more difficult, according to Shao and

Class	Eg. of metrics and suites
Basic metrics	Lines of Code (LOC), McCabe's Cyclomatic Complexity (CC) [23], Halstead's suite [24]
Object-oriented metrics	Chidamber and Kemerer (C&K) suite [42], Li and Henry suite [106], Lorenz and Kidd suite [107], Metrics for Object-oriented Design (MOOD) suite [108], Quality Metrics for Object-oriented Design (QMOOD) suite [109]
Component-based metrics	Narasimhan and Hendradjaya suite [111]
Slice-based metrics	Weiser suite[112], Ott and Thuss suite [115]
Cognitive metrics	Shao and Wang suite [121], Misra et al. suite [116]
Textual and readability metrics	Scalabrino et al. suite [117], Buse and Weimer suite [118], Dorn suite [119]
Anti-pattern metrics	Taba et al. suite [120]
Code change metrics	Moser et al. suite [71]

Table 7: Representative classes of metrics

Wang [121], than a sequence-flow structure and hence, it is assigned a higher cognitive weight. Following the work of Shao and Wang [121], Misra et al. [116] proposed a large suite of cognitive metrics (cf. Table 6).

Textual and readability metrics. A number of metrics have emerged to evaluate the readability of source-code based on its textual properties. For instance, Scalabrino et al. [117] proposed a set of metrics addressing the quality of the source-code comments and identifiers (cf. Table 6). These metrics are based on natural language processing (NLP) techniques. Buse and Weimer [118], in turn, used a simpler set of readability metrics (cf. Table 6) which do not require NLP. Their metrics capture several properties related to the count and length of different code tokens and the indentation in the source-code. In the same vein, Dorn [119] introduced a suite with a set of features covering the source-code structure, syntax, visual perception and alignment (cf. Table 6).

Anti-pattern metrics. The quality of the source-code was also evaluated in the literature in terms of the presence of code smells (also called antipatterns). Originally introduced by Fowler et al. [122], these antipatterns indicate poor design solutions and implementation issues in the source-code. Taba et al. [120] proposed a suite of metrics based on the antipatterns identified in the source-code (cf. Table 6). Taba et al. metrics are not computed on a single version of the source-code, but rather consider its whole development history including all the previous releases. Doing so, these metrics can also provide in-

dications about the potential improvements or degradation along the development process of the source-code.

Code change metrics. Metrics incorporating the development process of source-code have been also proposed by other authors in the literature. Typically, these metrics capture changes across different releases and quantify their impact on the overall software quality. Examples of these metrics are those proposed in the Moser et al. suite [71] (cf. Table 6).

4.3.2. At what levels of granularity are the different source-code metrics calculated? (RQ3.2)

The metrics identified in the existing reviews are calculated at different levels of granularity. Based on the insights reported in [27, 28, 37, 38, 73], the following classification emerged: (a) *file/method level metrics*, (b) *class level metrics*, (d) *package level metrics*, (e) *component level metrics* and (f) *slice level metrics*.

File/method level metrics. These metrics are the most general as they can be applied to any source-code file or method regardless of the language paradigm in which it is written. Examples of metrics that are collected at this granularity level are *LOC, CC* [23], *Halstead's suite* [24], *Buse and Weimer suite[118]*, *Sclabrino et al. suite* [117], *Taba et al. suite* [120] and metrics from the *Moser et al. suite* [71].

Class level metrics. These metrics are based on the concept of a "class" and thus they are only applicable to source-code written in object-oriented programming languages. *C&K metrics* [42] are the most common class metrics identified in the existing literature reviews. These metrics require an overview about the class methods and attributes as well as the interactions between the different classes in the code base. Other examples of class level metrics are within the *Li and Henry [106], Lorenz and Kidd [107], MOOD [108] and QMOOD [109]* suites.

Package level metrics. These metrics require a package structure in the code base. Herein, metrics such as *Package Level Cohesion (Pcoh)* [123] and *Package Level Coupling (PLC)* [124] were proposed to measure the extent of coupling and cohesion at the level of source-code packages.

Component level metrics. These metrics investigate the quality of software components. This class can be further divided into interface metrics (e.g., *number of interfaces in an application* [125]), interface method metrics (e.g., *number of methods and parameters* [70]) and property signature metrics (e.g., *number of arguments passed by reference and arguments passed by values* [126]).

Slice level metrics. These metrics take a different perspective to divide the source-code. Such a perspective relies on the control and data-flow dependencies to divide the source-code into units that can be assessed in a singular manner [112]. As mentioned in Section 4.3.1, examples of slice-based metrics include those in the *Weiser* [112] and *Ott and Thuss* [115] *suites*.

4.3.3. What quality attributes are captured by existing source-code metrics? (RQ3.3)

Source-code metrics were suggested to capture a wide array of quality attributes. These attributes can be organized into *internal* and *external* attributes. Internal attributes are meant to capture the structural properties of the source-code [56]. The attributes that have been classified in the existing reviews as being internal [25, 26] are *abstraction, cohesion, coupling, complexity, encapsulation, inheritance, messaging, modularity, polymorphism* and *size*. External attributes, in turn, denote the quality of the source-code as perceived by its stakeholders [56]. In this vein, *understandability, cognitive complexity, readability, fault proneness, change proneness, maintainability, effort, code decay, stability, completeness, effectiveness, flexibility, functionality, reliability, reusability* and *testability* have been all classified in the literature as external quality attributes [7, 25, 26, 127, 128]. Tables 8 and 9 summarize the identified internal and external quality attributes respectively and provide examples of metrics capturing each of them.

Among the metrics identified in Section 4.3.1, the *C&K suite* [42] has been proposed to evaluate internal attributes including complexity, coupling, cohesion, inheritance and modularity as well as external attributes such as understandability, change proneness, completeness, effort, error proneness, maintainability, reliability, reusability, stability and testability. Similarly, metrics from the QMOOD suite [109] have been suggested to capture abstraction, cohesion, complexity, coupling, encapsulation, inheritance, messaging, size, in addition to, understandability, effectiveness, flexibility, functionality, reliability and reusability. The *LOC* metric, *MOOD* [108], *Lorenz and kidd* [107], and *Li and Henry suites* [106] were also commonly proposed to investigate several internal and external quality attributes (cf. Tables 8 and 9 respectively). While the aforementioned metrics are suggested to capture many of the identified internal and external attributes, source-code readability requires a different set of metrics emphasizing the textual properties of the source-code (e.g., *Buse and Weimer suite* [118], *Dorn suite* [119], *Sclabrino et al. suite* [117]). Similarly, cognitive complexity is captured using a particular set of metrics including those in *Misra et al.* [116] suite and the *CFS* metric of Shao and Wang [121].

In the literature, external quality attributes were either directly operationalized using source-code metrics (as illustrated in Tables 8 and 9) or associated with internal quality attributes, which in turn are operationalized with different source-code metrics. This is particularly the case for maintainability, understandability, fault proneness, reusability, reliability, which have been related to internal attributes such as complexity, coupling, cohesion and inheritance [73, 85, 98].

Internal Attribute	Examples of Metrics	Ref. in Reviews
Abstraction	ANA from QMOOD	[25]
Cohesion	LCOM from C&K suite; CAM from QMOOD suite; Weiser metrics; Ott and Thuss metrics; Pcoh	[5, 25, 26, 84, 85]
Coupling	CBO from C&K; CF from MOOD suite; DCC from QMOOD suie; NF from Lorenz and Kidd suite; Halstead's metrics; PLC	[5, 7, 25, 26, 73]
Complexity	WMC from C&K; NOM_{QMOOD} from QMOOD; NPM, NM from Lorenz and Kidd suite; CC; Halstead's metrics	[25, 26, 85]
Encapsulation	CluF, MHF, AHF from MOOD suite; DAM from QMOOD	[25]
Inheritance	DIT, NOC from C&K; AIF, MIF, RF from MOOD; MFA from QMOOD, NMI from Lorenz and Kidd suite	[5, 25, 26]
Messaging	CIS from QMOOD suite	[25]
Modularity	MPC from Li and Henry suite; NF from Lorenz and Kidd suite; LCOM from C&K	[5]
Polymorphism	PF from MOOD suite	[25]
Size	LOC; DSC, NOH from QMOOD suite; AMS from Lorenz and Kidd suite; NOM_{QMOOD}, NOP from QMOOD	[25, 26]

Table 8: List of identified internal attributes and examples of metrics capturing them. The abbreviations of the metrics are provided in Table 6

4.3.4. How have the existing source-code metrics been categorized in the literature? (RQ3.4)

Source-code metrics have been categorized differently in the literature. Notably as mentioned in Sections 4.3.2 and 4.3.3, existing metrics can be, respectively, categorized by level of granularity and attribute. Moreover, some authors have distinguished between **product** and **process** metrics [29, 30]. Product metrics capture properties associated directly with the source-code (e.g., *C&K* [42], *MOOD* [108], *QMOOD* [109], *Lorenz and Kidd* [107], *Li and Henry* [106] suites), while process metrics capture properties associated with the process of editing the source-code (e.g., code change metrics in *Moser et al. suite* [71]).

Another categorization in the literature discerns **static** and **dynamic** metrics [31]. Static metrics evaluate properties derived following the static analysis of the source-code, while dynamic metrics capture properties that are observed at run-time. The suite proposed by Narasimhan and Hendradjaya [111], for instance, comprises both static and dynamic metrics. The static metrics (i.e., *CPD, CID, CIID, COID, CAID, CRIT*, cf. Table 6) characterize the internal complexity of a component [111]. Being static, these metrics can be collected early during the software development process. The dynamic metrics (i.e., *NC,*

External Attribute	Examples of Metrics	Ref. in Reviews
Understandability	CC; Halstead's volume; WMC, DIT, CBO, RFC from C&K suite; QMOOD suite; LOC	[5–7]
Cognitive complexity	CFS, Misra et al. metrics	[84]
Readability	Buse and Weimer suite; Dorn suite; Sclabrino et al. suite	[26, 65, 85]
Fault proneness	DIT, NOC, CBO, RFC, LCOM, WMC from C&K suite; Data DAC from Li and Henry suite; LOC; Taba et al. metrics; Moser et al. metrics	[7, 25, 26, 67, 68, 84]
Change proneness	DIT, NOC, CBO, RFC, LCOM from C&K suite	[7]
Maintainability	NOC, RFC, LCOM from C&K suite, MPC, DAC, NOM from Li and Henry suite; LOC; Halstead's Volume; Weiser metrics; number of interfaces in an application; Code change metrics	[6, 7, 26, 37, 84]
Effort	CBO, WMC from C&K; LOC; Size1, Size2, MPC, DAC, $NOM_{L\&H}$ from Li and Henry suite	[25]
Code decay	Code change metrics; LOC	[36]
Stability	WMC from C&K suite; LOC	[7]
Completeness	CC, Halstead's volume, WMC, DIT, CBO, RFC from C&K suite	[5]
Effectiveness	MIF, AIF from MOOD suite; QMOOD suite	[6, 98]
Flexibility	QMOOD suite	[6]
Functionality	QMOOD suite	[6]
Reliability	CBO, RFC from C&K suite; CC; DCC from QMOOD suite	[26]
Reusability	QMOOD suite; LOC; LCOM, CBO, RFC, DIT, WMC, NOC from C&K; MPC, DAC from Li and Henry suite, Package Level Cohesion (Pcoh); PLC; number of methods and parameters (in interface); number of arguments passed by reference and arguments passed by values (in interface)	[5–7, 37, 73]
Testability	RFC, CBO, LCOM from C&K; LOC	[7, 25]

Table 9: List of identified external attributes and examples of metrics capturing them. The abbreviations of the metrics are provided in Table 6

ANAC, ACD, AACD, PNAC, cf. Table 6), in turn, assess the dynamic behavior and interplay between the

components at run-time [111]. These metrics require a complete iteration in the development of the software application.

Source-code metrics have been also organized by **language paradigm**. For instance, Nuñez-Varela et al. [32] discerned *object-oriented metrics* (e.g., *C&K metrics* [42]), *aspect-oriented metrics* (e.g., *Base-Aspect Coupling (BAC)* [129]), *feature-oriented metrics* (e.g., *Scattering of a Feature (FSCA)* [130]) and *procedural metrics* (e.g., *LOC*).

Last but not least, by looking at the large body of metrics identified by the existing literature reviews, it is possible to distinguish **control-flow** and **data-flow** metrics. While the former captures the interplay of instructions within the program and aims at quantifying the complexity of execution paths within the system (e.g., *CC* [23]), the latter emphasizes the data perspective of the system and measures the flow of information between the program variables (e.g., *MPC* [106] i.e., part of Li and Henry suite [106], cf. Table 6). Another possible categorization of existing metrics differentiates between the **syntax** and **semantic** ones. Syntax metrics evaluate the source-code at the level of its grammar (e.g., *C&K metrics*), while semantic metrics evaluate the source-code at the level of its meaning. Semantic metrics can be further divided into metrics capturing the semantics expressed by the programming language in which the code is written (e.g., *CFS* and *Misra et al. suite* [116, 121]) and metrics capturing the semantics derived from the natural language vocabulary used in the source-code identifiers and comments (e.g., *Sclabrino et al. suite* [117]).

4.3.5. What metrics have been validated? (RQ3.5)

The existing reviews cover empirical studies investigating the relationship between source-code metrics and different quality attributes, among which *fault-proneness*, *maintainability* and *code decay* are the most popular. For instance, Jabangwe et al. [25] identified the studies linking *C&K metrics* to fault-proneness. Overall, *WMC, CBO, RFC* and partially *LCOM* (cf. Table 6) were found to have a significant relationship with fault-proneness, while *DIT* and *NOC* (cf. Table 6) could not be related to fault-proneness in the majority of the studies . Similar insights were provided by Isong and Obetent [100]. Besides, Radjenovic et al. [34] found that process metrics (e.g., [71]) have a significant relationship with fault-proneness in several studies. The issue of inconclusiveness of empirical results, in turn, could be seen when comparing the empirical studies on the relationship of *LOC*, *Halstead's metric*s and fault-proneness. [34]. Therein, the findings were different among the studies, making it difficult to draw clear conclusions. As for the *MOOD suite*, many studies could not associate its metrics with fault-proneness [34, 85].

The studies investigating the relationship between source-code metrics and maintainability have tested mostly the metrics from *C&K* [42] and *Li and Henry* [106] suites. The insights reported by Jabangwe et al. [25] on the *C&K suite* are similar to those reported previously with regards to fault-proneness. Indeed, *WMC, CBO, RFC and LCOM* were found to have a significant relationship with maintainability, whereas *DIT and NOC* could not be related to maintainability in the majority of the studies. As for the metrics

from the Li and Henry suite, it has been shown that *DAC and MPC* (cf. Table 6) were also associated with maintainability.

The existing empirical studies have also investigated the relationship between source-code metrics and code decay. Process metrics have been tested in this context. However, the findings in the review of Bandi et al. [36] were inconclusive regarding this class of metrics. Alternatively, the *WMC metric* (from the *C&K suite* [42]) was found to be a good indicator of code decay in some studies.

While a large set of metrics have been identified, only a few of them (i.e., mainly basic, object-oriented and process metrics) have been reported in the existing literature reviews as being empirically validated. When exploring the primary studies (cited in the reviews) where the metrics were proposed, more empirical studies can be found. For instance, the metrics capturing the textual features of source-code proposed by Scalabrino et al. [117], Dorn [119], Buse and Weimer [118] have been associated with source-code readability in their respective studies. Moreover, the anti-pattern-based metrics proposed by Taba et al. [120] have been shown to provide better predictions of faults in the source-code.

4.4. What metric-based approaches have been proposed in the literature to evaluate the quality of source-code? (RQ4)

Several metric-based approaches have been used to evaluate the quality of source-code. These approaches rely on machine learning models based on classification, regression, clustering and dimensional reduction techniques. The models are trained with the support of traditional machine learning algorithms (e.g., decision tree) or deep learning algorithms (neural networks) [131].

Based on the findings of the existing literature reviews, fault-proneness is the most popular attribute predicted by the proposed machine learning models [25, 27, 30, 51, 84, 86–88, 95, 96, 100, 103]. Singh et al. [132], for instance, proposed a decision tree model based on *LOC* and the *C&K metrics* to predict faults in the source-code. Similarly, Quah and Thwin proposed a neural network model based on a set of metrics including those from the *C&K suite*. Catal et al. in [133] proposed a clustering model based on *LOC, CC* and part of the *Halstead's metrics suite* [24]. Besides product metrics, process metrics based on code change have been also used to build models that predict fault-proneness. This is the case in [134] where Arisholm et al. suggested that Adaptive boosting (Adaboost) and C4.5 Decision tree models comprising process metrics can provide accurate predictions of fault-proneness.

Readability is among the attributes that have been also predicted by the existing machine learning models. However, the set of literature reviews covering this attribute is rather limited [65]. Scalabrino et al. [117] derived a number of features (i.e., average, max) from the metrics they proposed (cf. Section 4.3.1) and used logistic regression to construct their readability model. Similarly, Dorn [119] extracted a number of source-code features and used the wrapper approach [135] to identify the most discriminative ones, which, in turn, can be used to build an accurate classifier. Another readability model was proposed by Buse and

Weimer [118]. The authors extracted a set of textual features based on their metrics (cf. Section 4.3.1) and used Principal Component Analysis (PCA) [131] to identify the most discriminative features. Furthermore, the authors trained a Bayesian classifier [131] to predict whether a source-code snippet is more readable or less readable. Dorn [119], Buse and Weimer[118] relied on dimensional reduction techniques to identify the important features characterizing their source-code.

4.5. What tools and projects are popular to extract source-code metrics? (RQ5)

The analysis of the covered literature reviews led to the identification of a set of recurrent tools (i.e., cited in more than one review). After filtering out the non-available and discontinued ones, the list shown in Table 10 has emerged. This list comprises both academic and commercial tools. The majority of them operate on Java Source-code and deliver product metrics (e.g., CKJM). In addition, some of them go a step further and use the metrics to suggest code smells and violations of coding conventions (e.g., Xradar).

Tool name	**Tool link**	**Ac./Com.**	**Mtr/CS**	**Lang. supported**	**Refs**
CCCC	http://cccc.sourceforge.net	Academic	P. Metrics	C, C++	[32, 83]
CKJM	https://www.spinellis.gr/sw/ckjm/	Academic	P. Metrics	Java	[32, 64, 73, 79, 99]
CMT++/CMTJava	https://www.verifysoft.com/en_cmtx.html	Commercial	P. Metrics	C, C++, C#, Java	[64, 83]
DÉCOR	http://www.ptidej.net/research/designsmells/	Academic	Code smells	Java	[74, 97]
Eclipse Metrics Plugin	http://easyeclipse.org/site-1.0.2/plugins/metrics.html	Academic	P. Metrics	Java	[7, 83]
JHawk	http://www.virtualmachinery.com/jhawkmetrics.htm	Commercial	P. Metrics	Java	[32, 64, 83]
Stench Blossom	https://multiview.cs.pdx.edu/refactoring/smells/	Academic	Code smells	Java	[93, 97]
Understand	https://www.scitools.com	Commercial	P. Metrics	C, C++, C#, Java, Python, and 15 other languages	[32, 64, 83, 92, 93]
Xradar	http://xradar.sourceforge.net	Academic	P. Metrics and Code smells	Java	[82, 99]

Table 10: List of tools to extract metrics. Abbreviations: Ac.: Academic, Com.: Commercial, Mtr: Metrics, CS: Code smell, Lang.: language, Refs: References in the identified reviews. P.: Product

Source code metrics have been extracted in a number of projects. Table 11 lists the projects that have been identified by more than one literature review. Most of them comprise open-source code, while only a single one (i.e., the PROMISE Repository) provides data-sets of metrics pre-computed for several code bases at different levels of granularity. All the open-source projects are written (partially or fully) in Java. In addition, many of them are based on Apache products (e.g., ANT, Camel, Ivy, Lucene, Synapse and Tomcat).

5. Discussion

In this discussion section, the findings of this STR are synthesized into a conceptual framework (cf. Section 5.1), Following that, a set of gaps are identified in the literature (cf. Section 5.2). Finally, a research agenda is proposed to address these gaps and thus pave the path for the future research (cf. Section 5.3).

Project name	Link	Type	Language	Refs
Apache ANT	https://github.com/apache/ant	source-code	Java	[26, 32, 95]
Apache Camel	https://github.com/apache/camel	source-code	Java	[26, 95]
Apache Ivy	https://github.com/apache/ant-ivy	source-code	Java	[26, 95]
Apache Lucene	https://github.com/apache/lucene	source-code	Java	[32, 95]
Apache Synapse	https://github.com/apache/synapse	source-code	Java	[26, 95]
Apache Tomcat	https://github.com/apache/tomcat	source-code	Java	[26, 95]
Argo uml	https://github.com/argouml-tigris-org	source-code	Java	[32, 35, 95]
Eclipse	https://git.eclipse.org/c/	source-code	Java	[26, 32, 35, 84, 90, 95]
FreeCol	https://github.com/FreeCol/freecol	source-code	Java	[26, 35]
GanttProject	https://www.ganttproject.biz	source-code	Java	[26, 32]
Hibernate	https://github.com/hibernate	source-code	Java	[32, 35, 90]
Jabref	https://github.com/JabRef/jabref	source-code	Java	[26, 32]
JBoss	https://github.com/wildfly/wildfly	source-code	Java	[26, 90]
jEdit	https://github.com/albfan/jEdit	source-code	Java	[26, 32]
JFreeChart	https://github.com/jfree/jfreechart	source-code	Java	[26, 35]
JHotDraw	https://github.com/wumpz/jhotdraw	source-code	Java	[26, 32]
Mozilla Porject	https://github.com/mozilla/	source-code	Several languages inc. Java	[90, 95]
PROMISE repository	http://promise.site.uottawa.ca/SERepository/	pre-computed metrics		[32, 84, 87]

Table 11: List of projects. Abbreviation: Refs: References in the identified reviews

5.1. Conceptual framework

The analysis of the existing literature reviews provides rich insights on source-code metrics. These insights can be synthesized into the conceptual framework depicted in Figure 3. Therein, a metric has a granularity defining the level of source-code at which it can be collected. As presented in Section 4.3.2 metrics can be collected at the file/method, class, package, component and slice levels.

A metric can be of different types. Among the categorizations introduced in Section 4.3.4, one can notably discern product versus process metrics (i.e., capture properties related directly to the source-code or capture properties related to the process of editing it), static versus dynamic metrics (i.e., capture properties derived from the static analysis of source-code or from the behavior observed at run-time), control-flow versus data-flow metrics (i.e., capture the interplay of instructions or flow of data within the program) and syntax versus semantic metrics (i.e., evaluate the source-code grammar or meaning). Semantic metrics, in turn, can be divided into those evaluating the semantics of the used programming language and those evaluating the semantics of the natural language used in the source-code (i.e., identifiers and comments).

A metric can be applied directly to the source-code, or combined with other metrics as part of a machine learning model, which, in turn, is applied to the source-code (cf. Section 4.4). The source-code is written in a specific programming language (e.g., Java) belonging to a particular paradigm (e.g., Object-oriented).

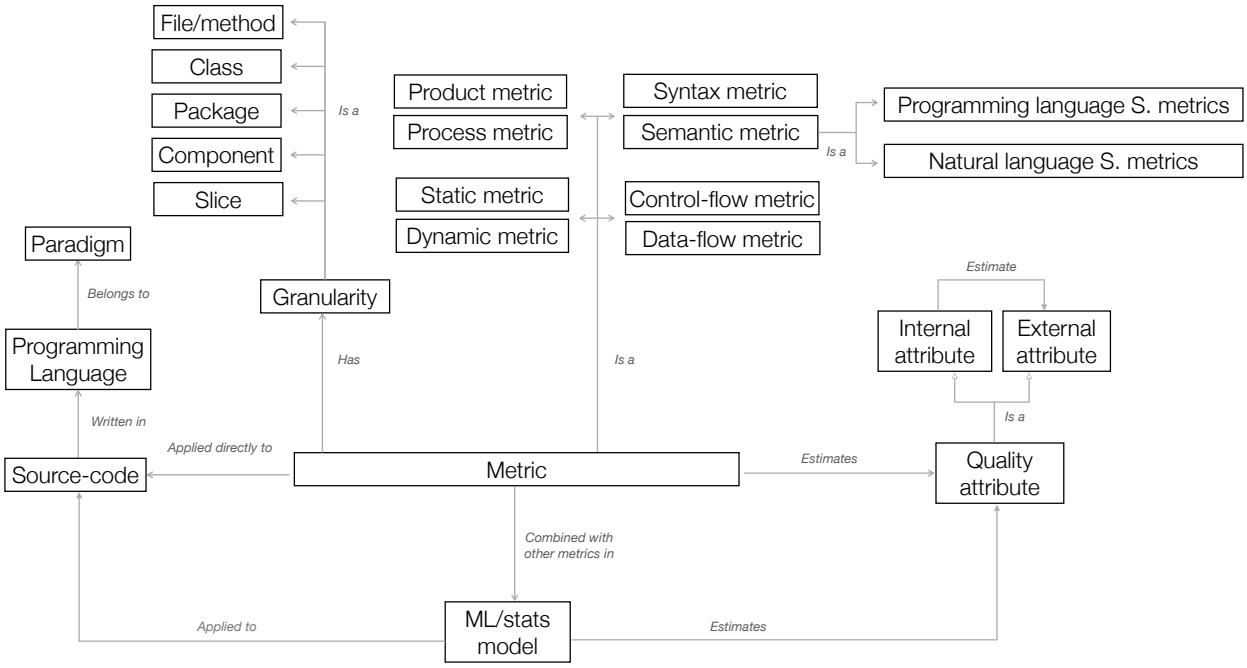


Figure 3: A conceptual framework synthesizing the findings of the tertiary review. Abbreviation: S.: Semantic

An individual metric or a set of metrics (blended into a machine learning model) aim at estimating one or more quality attributes. Following the classification given in Section 4.3.3, attributes can be either internal (addressing the internal properties of the source-code, e.g., coupling) or external (reflecting the stakeholder perception of the source-code, e.g., maintainability). Last but not least, as explained in Section 4.3.3, a quality attribute can be estimated directly from a metric, a machine learning model, or throughout another quality attribute, which is the case for external attributes such as maintainability and understandability that have been captured based on internal attributes like coupling and cohesion.

All in all, through its different attributes, the conceptual framework synthesized in this work provides an overarching characterization of source-code metrics which is meant to guide researchers and practitioners in the choice of the metrics fulfilling their needs for particular purposes. Such a choice can be based on the targeted quality attribute, programming language and paradigm as well as properties associated with the source-code, its evolution over time and its static, dynamic, syntax, semantics, control-flow and data-flow features.

5.2. Literature gaps

The framework presented in Section 5.1 allows positioning the empirical studies investigating the relationship between source-code metrics and cognitive load [18–22]. Looking at these studies, one could notice that the investigated source-code metrics capture mainly properties associated with the program control-flow and data-flow (e.g., LOC, CC [23], Halstead's metrics [24], DepDegree i.e., a metric with similar properties

than MPC), which in turn, cover a narrow part within the conceptual framework synthesized in this STR (cf. Figure 3). The metrics capturing the readability of source-code (i.e., natural language semantic metrics in Figure 3), were, for instance, not covered. Following the cognitive load theory (cf. Section 2.2), poor source-code readability can affect developers' extraneous load by inducing an extra layer of (unnecessary) complexity emerging from its representation. Overlooking this class of metrics would, therefore, result in measurements that do not reflect developers' cognitive load. These insights can provide a plausible explanation for the inability to associate the majority of the used source-code metrics with developers' cognitive load (except for the DepDegree which had moderate correlations with the used psycho-physiological measures of cognitive load [19]).

While, the framework presented in Section 5.1 shows that source-code metrics can be calculated at different levels of granularity, the metrics used in the previous empirical studies are extracted at the file, method and class levels only. Measuring the properties of source-code at these levels of granularity may not always provide estimates that align with developers' cognitive load. As pointed out by Petrusel and Mendling [136] (in the context of conceptual models), when given a task, users do not inspect the entire artifact but only the task-relevant parts of it. Similarly, when conducting a task on source-code, developers are likely to focus only on the source-code fragments that are relevant for that task. Although the used metrics can characterize the source-code at the level of its files, methods and classes, the obtained measurements may not necessarily reflect developers' cognitive load if their task addresses fragments that do not span over the entire (measured) files, methods and classes of the code base but rather cover small fragments within each of them. However, this factor might not have influenced the findings of the empirical studies in [18–22] as they have used small source-code snippets and formulated tasks where users were required to engage with the snippets as a whole. This, in turn, made their entire content relevant for the task. Nevertheless, knowing that in realistic settings, tasks typically cover specific fragments within large code bases, the used metrics might not align with users' cognitive load if not tailored to measure the properties of the task-relevant source-code only.

5.3. Research agenda

This section suggests a research agenda based on the literature gaps identified in Section 5.2. To ensure a better alignment between source-code metrics and developers' cognitive load, it is necessary for the future research in this topic to consider the metrics capturing the readability of source-code, together with other types of metrics (e.g., control-flow, data-flow, syntax, product, process, static and dynamic metrics). These metrics should be used in a combined manner to develop machine learning models capturing a wide spectrum of the features characterizing source-code.

In addition, it is crucial to develop source-code metrics that capture the task-relevant code in order to better align with developers' cognitive load. Applying slicing techniques [113] can help to attain this

objective assuming that the task-relevant fragments address features of interest that share some common control and data flow dependencies. The few slice-based metrics proposed in the literature [112, 115] make use of these techniques and thus can be adapted to provide measurements fitting the task-relevant code. In the same vein, new slice-based metrics can be proposed. Unlike the existing ones, the new slice-based metrics would measure the same properties as those captured by the metrics operating at the method and class levels (e.g., *Halstead's* and *C&K* suites), with the only constraint to cover just the task-relevant source-code.

Last but not least, while source-code metrics can be leveraged to estimate developers' cognitive load, this load should be reduced to ease software development tasks. Based on the background introduced in Section 2.2, engaging with source-code can raise two levels of cognitive load i.e., intrinsic and extraneous. Intrinsic load would emerge from the complexity inherent to the software functionalities encoded in the source-code. Such a complexity is referred in the software engineering literature as the essential source-code complexity [3, 137]. Extraneous load would emerge from the poor representation of the source-code, which is known in the literature as the accidental source-code complexity [3, 137]. Intrinsic load might not be reducible without affecting the source-code functionalities. Extraneous load, in turn, can be reduced if the source-code representation is improved. To this end, it is necessary to know which metrics capture the source-code essential complexity and which metrics capture the source-code accidental complexity. The measurements provided by the latter class of metrics should be, subsequently, reduced (e.g., through refactoring) to ease software development tasks.

6. Threats to validity

This STR study is subject to several threats to validity. These threats can be divided into *descriptive*, *theoretical* and *repeatability*.

Descriptive validity. Descriptive validity denotes the extent to which the findings reported in the study are described objectively and accurately. In this STR, this threat was mitigated by using a well-defined data extraction scheme (cf. Section 3) that has been applied to all the literature reviews identified during the literature search.

Theoretical validity. Theoretical validity denotes the extent to which the study covers the body of literature it intends to investigate. To ensure a wide coverage of the literature, the search strings were carefully designed (cf. Section 1) and several search iterations were conducted i.e., pilot search, main search and snowballing search (cf. Sections 3.2, 3.6 and 3.7). Also, to reduce any bias in the selection of the relevant literature, a set of inclusion and exclusion criteria were defined and borderline papers were discussed with colleagues within the software engineering community. Nevertheless, it is worthwhile to mention that the

search process was completed in August 2021 and therefore all work published after this date was not covered.

Repeatability validity. To be able to reproduce the finding of any research, it is crucial to report the approach followed to collect and analyze data. To fulfill this requirement, the steps of this STR and were documented in detail following well-recognized guidelines [51] (cf. Section 3).

7. Conclusion

Besides the high number of literature reviews covering source-code metrics, none of them provided a holistic overview of their properties and included the different facets of this topic. To address this issue, this study provided a global and consolidated tertiary review on source-code metrics. Following the STR approach presented in Section 3, the finding of this study yielded 55 literature reviews (cf. Section 4.1) covering 25 different categories of research questions (cf. Section 4.2). The analysis of these reviews enabled the identification of several classes of metrics (i.e., basic, object-oriented, component-based, slice-based, cognitive, textual and readability, anti-pattern, code change; cf. Section 4.3.1), which have been collected at different levels of granularity (i.e., file/method, class, package, component, slice; cf. Section 4.3.2), captured different internal and external quality attributes (cf. Section 4.3.3) and were categorized differently in the literature (e.g., product vs. process, static vs. dynamic, control vs. data-flow, syntax vs. semantics; cf. Section 4.3.4). The analysis also covered the empirical evaluations of existing metrics and found that fault-proneness, maintainability and code decay were the most popular attributes investigated for their relationship with different metrics (cf. Section 4.3.5). Moreover, the analysis allowed to scrutinize the machine learning models combining several metrics to estimate different quality attributes of the source-code (Section 4.4). The identified models were based on various techniques (i.e., classification, regression, clustering, dimensional reduction) and trained with the support of different algorithms (i.e., traditional or deep learning algorithms). Fault-proneness was the most popular attribute predicted by these models. Last but not least, a set of popular tools and projects covering source-code metrics were identified (Section 4.5). The findings of this study have been used to synthesize a conceptual framework linking the different concepts identified throughout this tertiary review (Section 5.1). After positioning the empirical studies investigating the relationship between source-code metrics and cognitive load within this framework, it has emerged that these studies cover a narrow part of the synthesized conceptual framework as they focused only on control-flow and data-flow metrics that are collected at the file, method and class granularity levels. The underlying limitations were discussed and a research agenda paving the road for the future work was proposed (Section 5.3). In brief, the research agenda recommends embedding the metrics capturing the readability of source-code together with other types of metrics (e.g., control-flow, data-flow, syntax, product, process, static and dynamic metrics) in machine learning models in order to provide an overarching characterization

of the source-code. Moreover, it sheds the light on code-slicing techniques which can be adopted to propose a new generation of slice-based metrics that capture the task-relevant code in their calculations. Last but the least, the research agenda raises the need to differentiate the metrics capturing the essential complexity of source-code and those capturing its accidental complexity and then focus on reducing the values of the latter class of metrics.

The insights of this study are expected to have an impact on both the software industry and our research community. Indeed, the findings of the STR and the proposed conceptual framework deliver a consolidated overview that can be used by industrials to identify the most adequate metrics to use in a particular setting. As for research, the raised literature gaps and the subsequent research agenda highlight the aspects requiring more attention from researchers in the software engineering community. Addressing these research challenges, would, in turn, result in source-code metrics providing a better alignment with developers' cognitive load and thus enabling better estimates of the difficulty associated with different software development tasks.

Acknowledgment

Work supported by the International Postdoctoral Fellowship (IPF) Grant (Number: 1031574) from the University of St. Gallen, Switzerland.
Acknowledgment and special thanks to Prof. Dr. Barbara Weber and Prof. Dr. Ekkart Kindler for their feedback and suggestions during the development of this STR.

References

[1] R. Dumke, C. Ebert, Software measurement: Establish-extract-evaluate-execute, Chapter 8.3 Measurements for Project Control.

[2] J. F. Dooley, Dooley, Software Development, Design and Coding, Springer, 2017.

[3] V. Antinyan, Evaluating essential and accidental code complexity triggers by practitioners' perception, IEEE Software 37 (6) (2020) 86–93.

[4] Ieee standard glossary of software engineering terminology, IEEE Std 610.12-1990 (1990) 1–84doi:10.1109/IEEESTD.1990.101064.

[5] M. Mijač, Z. Stapić, Reusability metrics of software components: survey, in: Proceedings of the 26th Central European Conference on Information and Intelligent Systems, 2015, pp. 221–231.

[6] F. Wedyan, S. Abufakher, Impact of design patterns on software quality: a systematic literature review, IET Software 14 (1) (2020) 1–17.

[7] E. M. Arvanitou, A. Ampatzoglou, A. Chatzigeorgiou, M. Galster, P. Avgeriou, A mapping study on design-time quality attributes and metrics, Journal of Systems and Software 127 (2017) 52–77.

[8] F. Paas, J. E. Tuovinen, H. Tabbers, P. W. Van Gerven, Cognitive load measurement as a means to advance cognitive load theory, Educational psychologist 38 (1) (2003) 63–71.

[9] F. Chen, J. Zhou, Y. Wang, K. Yu, S. Z. Arshad, A. Khawaji, D. Conway, Robust multimodal cognitive load measurement, Springer, 2016.

[10] J. Veltman, C. Jansen, The role of operator state assessment in adaptive automation, TNO DEFENCE SECURITY AND SAFETY SOESTERBERG (NETHERLANDS), 2005.

[11] K. A. McKiernan, J. N. Kaufman, J. Kucera-Thompson, J. R. Binder, A parametric manipulation of factors affecting task-induced deactivation in functional neuroimaging, Journal of cognitive neuroscience 15 (3) (2003) 394–408.

[12] R. Riedl, P.-M. Léger, Fundamentals of neurois, Studies in neuroscience, psychology and behavioral economics (2016) 127.

[13] K. Holmqvist, M. Nyström, R. Andersson, R. Dewhurst, H. Jarodzka, J. Van de Weijer, Eye tracking: A comprehensive guide to methods and measures, OUP Oxford, 2011.

[14] B. Weber, T. Fischer, R. Riedl, Brain and autonomic nervous system activity measurement in software engineering: A systematic literature review, Journal of Systems and Software (2021) 110946.

[15] L. Gonçales, K. Farias, B. C. da Silva, Measuring the cognitive load of software developers: An extended systematic mapping study, Information and Software Technology (2021) 106563.

[16] L. Gonçales, K. Farias, B. da Silva, J. Fessler, Measuring the cognitive load of software developers: A systematic mapping study, in: IEEE/ACM 27th International Conference on Program Comprehension, 2019, pp. 42–52.

[17] R. Riedl, T. Fischer, P.-M. Léger, F. D. Davis, A decade of neurois research: progress, challenges, and future directions, ACM SIGMIS Database: the DATABASE for Advances in Information Systems 51 (3) (2020) 13–54.

[18] N. Peitek, S. Apel, C. Parnin, A. Brechmann, J. Siegmund, Program comprehension and code complexity metrics: An fmri study, in: 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), IEEE, 2021, pp. 524–536.

[19] N. Peitek, J. Siegmund, S. Apel, C. Kästner, C. Parnin, A. Bethmann, T. Leich, G. Saake, A. Brechmann, A look into programmers' heads, IEEE Transactions on Software Engineering 46 (4) (2018) 442–462.

[20] J. Medeiros, R. Couceiro, J. Castelhano, M. C. Branco, G. Duarte, C. Duarte, J. Durães, H. Madeira, P. Carvalho, C. Teixeira, Software code complexity assessment using eeg features, in: 2019 41st Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC), IEEE, 2019, pp. 1413–1416.

[21] J. Medeiros, R. Couceiro, G. Duarte, J. Durães, J. Castelhano, C. Duarte, M. Castelo-Branco, H. Madeira, P. de Carvalho, C. Teixeira, Can eeg be adopted as a neuroscience reference for assessing software programmers' cognitive load?, Sensors 21 (7) (2021) 2338.

[22] R. Couceiro, R. Barbosa, J. Duráes, G. Duarte, J. Castelhano, C. Duarte, C. Teixeira, N. Laranjeiro, J. Medeiros, P. Carvalho, et al., Spotting problematic code lines using nonintrusive programmers' biofeedback, in: 2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE), IEEE, 2019, pp. 93–103.

[23] T. J. McCabe, A complexity measure, IEEE Transactions on software Engineering (4) (1976) 308–320.

[24] M. H. Halstead, Elements of Software Science (Operating and programming systems series), Elsevier Science Inc., 1977.

[25] R. Jabangwe, J. Börstler, D. Šmite, C. Wohlin, Empirical evidence on the link between object-oriented measures and external quality attributes: a systematic literature review, Empirical Software Engineering 20 (3) (2015) 640–693.

[26] A. Kaur, A systematic literature review on empirical analysis of the relationship between code smells and software quality attributes, Archives of Computational Methods in Engineering 27 (4) (2020) 1267–1296.

[27] C. Catal, B. Diri, A systematic review of software fault prediction studies, Expert systems with applications 36 (4) (2009) 7346–7354.

[28] F. N. Colakoglu, A. Yazici, A. Mishra, Software product quality metrics: A systematic mapping study, IEEE Access.

[29] C. G. P. Bellini, R. D. C. D. F. Pereira, J. L. Becker, Measurement in software engineering: From the roadmap to the crossroads, International Journal of Software Engineering and Knowledge Engineering 18 (01) (2008) 37–64.

[30] T. Hall, S. Beecham, D. Bowes, D. Gray, S. Counsell, A systematic literature review on fault prediction performance in software engineering, IEEE Transactions on Software Engineering 38 (6) (2011) 1276–1304.

[31] A. Tahir, S. G. MacDonell, A systematic mapping study on dynamic metrics and software quality, in: 2012 28th IEEE International Conference on Software Maintenance (ICSM), IEEE, 2012, pp. 326–335.

[32] A. S. Nuñez-Varela, H. G. Pérez-Gonzalez, F. E. Martínez-Perez, C. Soubervielle-Montalvo, Source code metrics: A systematic mapping study, Journal of Systems and Software 128 (2017) 164–197.

[33] M. Riaz, E. Mendes, E. Tempero, A systematic review of software maintainability prediction and metrics, in: 2009 3rd International Symposium on Empirical Software Engineering and Measurement, IEEE, 2009, pp. 367–377.

[34] D. Radjenović, M. Heričko, R. Torkar, A. Živkovič, Software fault prediction metrics: A systematic literature review, Information and software technology 55 (8) (2013) 1397–1418.

[35] R. Malhotra, A. Bansal, Predicting change using software metrics: A review, in: 2015 4th International Conference on Reliability, Infocom Technologies and Optimization (ICRITO)(Trends and Future Directions), IEEE, 2015, pp. 1–6.

[36] A. Bandi, B. J. Williams, E. B. Allen, Empirical evidence of code decay: A systematic mapping study, in: 2013 20th Working Conference on Reverse Engineering (WCRE), IEEE, 2013, pp. 341–350.

[37] M. Abdellatief, A. B. M. Sultan, A. A. A. Ghani, M. A. Jabar, A mapping study to investigate component-based software system metrics, Journal of systems and software 86 (3) (2013) 587–603.

[38] R. Burrows, A. Garcia, F. Taïani, Coupling metrics for aspect-oriented programming: A systematic review of maintainability studies, Evaluation of Novel Approaches to Software Engineering (2009) 277–290.

[39] J. Mendling, Detection and prediction of errors in epc business process models, Ph.D. thesis, Wirtschaftsuniversität Wien Vienna (2007).

[40] W. S. Torgerson, Theory and methods of scaling.

[41] N. Fenton, S. Pfleeger, Software metrics, 2nd edn, a rigorous and practical approach (1997).

[42] S. R. Chidamber, C. F. Kemerer, A metrics suite for object oriented design, IEEE Transactions on software engineering 20 (6) (1994) 476–493.

[43] R. E. Park, W. B. Goethert, W. A. Florac, Goal-driven software measurement. a guidebook., Tech. rep., Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Inst (1996).

[44] T. Borchert, Code profiling: Static code analysis (2008).

[45] J. Sweller, Cognitive load theory, in: Psychology of learning and motivation, Vol. 55, Elsevier, 2011, pp. 37–76.

[46] R. J. Sternberg, K. Sternberg, Cognitive psychology, Nelson Education, 2016.

[47] R. Z. Zheng, Cognitive load measurement and application: a theoretical framework for meaningful research and practice, Routledge, 2017.

[48] A. D. Baddeley, G. Hitch, Working memory, in: Psychology of learning and motivation, Vol. 8, Elsevier, 1974, pp. 47–89.

[49] G. A. Miller, The magical number seven, plus or minus two: Some limits on our capacity for processing information., Psychological review 63 (2) (1956) 81.

[50] G. Schryen, Preserving knowledge on is business value, Business & Information Systems Engineering 2 (4) (2010) 233–244.

[51] B. Kitchenham, Guidelines for performing systematic literature reviews in software engineering, Tech. rep. (2007).

[52] I. Nurdiani, J. Börstler, S. A. Fricker, The impacts of agile and lean practices on project constraints: A tertiary study, Journal of Systems and Software 119 (2016) 162–183.

[53] V. Sima, I. G. Gheorghe, J. Subić, D. Nancu, Influences of the industry 4.0 revolution on the human capital development and consumer behavior: A systematic review, Sustainability 12 (10) (2020) 4035.

[54] J. Webster, R. T. Watson, Analyzing the past to prepare for the future: Writing a literature review, MIS quarterly (2002) xiii–xxiii.

[55] H. Snyder, Literature review as a research methodology: An overview and guidelines, Journal of Business Research 104 (2019) 333–339.

[56] M. Glinz, Software quality: Software product quality, teaching material.

[57] B. Kitchenham, What's up with software metrics?–a preliminary mapping study, Journal of systems and software 83 (1) (2010) 37–51.

[58] C. Gall, S. Lukins, L. Etzkorn, S. Gholston, P. Farrington, D. Utley, J. Fortune, S. Virani, Semantic software metrics computed from natural language design specifications, IET software 2 (1) (2008) 17–26.

[59] D. Hutton, Clean code: A handbook of agile software craftsmanship, Kybernetes.

[60] S. Butler, M. Wermelinger, Y. Yu, H. Sharp, Exploring the influence of identifier names on code quality: An empirical study, in: 2010 14th European Conference on Software Maintenance and Reengineering, IEEE, 2010, pp. 156–165.

[61] S. Fakhoury, Y. Ma, V. Arnaoudova, O. Adesope, The effect of poor source code lexicon and readability on developers' cognitive load, in: 2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC), IEEE, 2018, pp. 286–28610.

[62] R. J. Miara, J. A. Musselman, J. A. Navarro, B. Shneiderman, Program indentation and comprehensibility, Communications of the ACM 26 (11) (1983) 861–867.

[63] M. Hansen, R. L. Goldstone, A. Lumsdaine, What makes code hard to understand?, arXiv preprint arXiv:1304.5257.

[64] L. Ardito, R. Coppola, L. Barbato, D. Verga, A tool-based perspective on software code maintainability metrics: A systematic literature review, Scientific Programming 2020.

[65] A. Bexell, Software source code readability: A mapping study (2020).

[66] O. Gómez, H. Oktaba, M. Piattini, F. García, A systematic review measurement in software engineering: State-of-the-art in measures, in: International Conference on Software and Data Technologies, Springer, 2006, pp. 165–176.

[67] P. Piotrowski, L. Madeyski, Software defect prediction using bad code smells: A systematic literature review, Data-Centric Business and Applications (2020) 77–99.

[68] F. Sabir, F. Palma, G. Rasool, Y.-G. Guéhéneuc, N. Moha, A systematic literature review on the detection of smells and their evolution in object-oriented and service-oriented systems, Software: Practice and Experience 49 (1) (2019) 3–39.

[69] S. Montagud, S. Abrahão, E. Insfran, A systematic review of quality attributes and measures for software product lines, Software Quality Journal 20 (3) (2012) 425–486.

[70] O. P. Rotaru, M. Dobre, Reusability metrics for software components, in: The 3rd ACS/IEEE International Conference onComputer Systems and Applications, 2005., IEEE, 2005, p. 24.

[71] R. Moser, W. Pedrycz, G. Succi, A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction, in: Proceedings of the 30th international conference on Software engineering, 2008, pp. 181–190.

[72] D. Wang, Y. Ueda, R. G. Kula, T. Ishio, K. Matsumoto, Can we benchmark code review studies? a systematic mapping study of methodology, dataset, and metric, Journal of Systems and Software (2021) 111009.

[73] B. Mehboob, C. Y. Chong, S. P. Lee, J. M. Y. Lim, Reusability affecting factors and software metrics for reusability: A systematic literature review, Software: Practice and Experience 51 (6) (2021) 1416–1458.

[74] K. Alkharabsheh, Y. Crespo, E. Manso, J. A. Taboada, Software design smell detection: a systematic mapping study, Software Quality Journal 27 (3) (2019) 1069–1148.

[75] J. Corbin, A. Strauss, Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory, SAGE Publications, 2014.
URL https://books.google.dk/books?id=hZ6kBQAAQBAJ

[76] A. AbuHassan, M. Alshayeb, L. Ghouti, Software smell detection techniques: A systematic literature review, Journal of Software: Evolution and Process 33 (3) (2021) e2320.

[77] J. P. dos Reis, F. B. e Abreu, G. de Figueiredo Carneiro, C. Anslow, Code smells detection and visualization: A systematic literature review, Archives of Computational Methods in Engineering (2021) 1–48.

[78] B. Gezici, N. Özdemir, N. Yılmaz, E. Coşkun, A. Tarhan, O. Chouseinoglou, Quality and success in open source software: A systematic mapping, in: 2019 45th Euromicro Conference on Software Engineering and Advanced Applications (SEAA),

IEEE, 2019, pp. 363–370.

[79] E. Fregnan, T. Baum, F. Palomba, A. Bacchelli, A survey on software coupling relations and tools, Information and Software Technology 107 (2019) 159–178.

[80] E. Y. Hernandez-Gonzalez, A. J. Sanchez-Garcia, M. K. Cortes-Verdin, J. C. Perez-Arriaga, Quality metrics in software design: A systematic review, in: 2019 7th International Conference in Software Engineering Research and Innovation (CONISOFT), IEEE, 2019, pp. 80–86.

[81] M. A. Zaidi, R. Colomo-Palacios, Code smells enabled by artificial intelligence: a systematic mapping, in: International conference on computational science and its applications, Springer, 2019, pp. 418–427.

[82] M. Yan, X. Xia, X. Zhang, L. Xu, D. Yang, S. Li, Software quality assessment model: A systematic mapping study, Science China Information Sciences 62 (9) (2019) 1–18.

[83] K. Valença, E. D. Canedo, R. M. D. C. Figueiredo, Construction of a software measurement tool using systematic literature review, in: 2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData), IEEE, 2018, pp. 1852–1859.

[84] Z. Li, X.-Y. Jing, X. Zhu, Progress on approaches to software defect prediction, Iet Software 12 (3) (2018) 161–175.

[85] S. Tiwari, S. S. Rathore, Coupling and cohesion metrics for object-oriented software: a systematic mapping study, in: Proceedings of the 11th Innovations in Software Engineering Conference, 2018, pp. 1–11.

[86] R. Özakıncı, A. Tarhan, Early software defect prediction: A systematic map and review, Journal of Systems and Software 144 (2018) 216–239.

[87] S. Karim, H. L. H. S. Warnars, F. L. Gaol, E. Abdurachman, B. Soewito, et al., Software metrics for fault prediction using machine learning approaches: A literature review with promise repository dataset, in: 2017 IEEE international conference on cybernetics and computational intelligence (CyberneticsCom), IEEE, 2017, pp. 19–23.

[88] L. Goel, D. Damodaran, S. K. Khatri, M. Sharma, A literature review on cross project defect prediction, in: 2017 4th IEEE Uttar Pradesh Section International Conference on Electrical, Computer and Electronics (UPCON), IEEE, 2017, pp. 680–685.

[89] K. Sagar, A. Saha, A systematic review of software usability studies, International Journal of Information Technology (2017) 1–24.

[90] H. Wu, L. Shi, C. Chen, Q. Wang, B. Boehm, Maintenance effort estimation for open source software: A systematic literature review, in: 2016 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE, 2016, pp. 32–43.

[91] M. Santos, P. Afonso, P. H. Bermejo, H. Costa, Metrics and statistical techniques used to evaluate internal quality of object-oriented software: A systematic mapping, in: 2016 35th International Conference of the Chilean Computer Science Society (SCCC), IEEE, 2016, pp. 1–11.

[92] D. Rattan, J. Kaur, Systematic mapping study of metrics based clone detection techniques, in: Proceedings of the International Conference on Advances in Information Communication Technology & Computing, 2016, pp. 1–7.

[93] E. Fernandes, J. Oliveira, G. Vale, T. Paiva, E. Figueiredo, A review-based comparative study of bad smell detection tools, in: Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering, 2016, pp. 1–12.

[94] B. Seref, O. Tanriover, Software code maintainability: a literature review, Int J Softw Eng Appl.

[95] R. Malhotra, A systematic review of machine learning techniques for software fault prediction, Applied Soft Computing 27 (2015) 504–518.

[96] J. Murillo-Morera, C. Quesada-López, M. Jenkins, Software fault prediction: A systematic mapping study., in: CIbSE, 2015, p. 446.

[97] G. Rasool, Z. Arshad, A review of code smell mining techniques, Journal of Software: Evolution and Process 27 (11) (2015) 867–895.

[98] N. Vanitha, R. ThirumalaiSelvi, A report on the analysis of metrics and measures on software quality factors–a literature study.

[99] P. Tomas, M. J. Escalona, M. Mejias, Open source tools for measuring the internal quality of java software products. a survey, Computer Standards & Interfaces 36 (1) (2013) 244–255.

[100] B. Isong, E. Obeten, A systematic review of the empirical validation of object-oriented metrics towards fault-proneness prediction, International Journal of Software Engineering and Knowledge Engineering 23 (10) (2013) 1513–1540.

[101] Y. A. Khan, M. O. Elish, M. El-Attar, A systematic review on the impact of ck metrics on the functional correctness of object-oriented classes, in: International Conference on Computational Science and Its Applications, Springer, 2012, pp. 258–273.

[102] J. Saraiva, E. Barreiros, A. Almeida, F. Lima, A. Alencar, G. Lima, S. Soares, F. Castor, Aspect-oriented software maintenance metrics: A systematic mapping study, in: 16th International Conference on Evaluation & Assessment in Software Engineering (EASE 2012), IET, 2012, pp. 253–262.

[103] C. Catal, Software fault prediction: A literature review and current trends, Expert systems with applications 38 (4) (2011) 4626–4636.

[104] S. Purao, V. Vaishnavi, Product metrics for object-oriented systems, ACM Computing Surveys (CSUR) 35 (2) (2003) 191–221.

[105] J. P. dos Reis, F. B. e Abreu, G. de Figueiredo Carneiro, C. Anslow, Code smells detection and visualization: A systematic literature review, Archives of Computational Methods in Engineering (2021) 1–48.

[106] W. Li, S. Henry, Object-oriented metrics that predict maintainability, Journal of systems and software 23 (2) (1993) 111–122.

[107] M. Lorenz, J. Kidd, Object-oriented software metrics: a practical guide, Prentice-Hall, Inc., 1994.

[108] F. B. Abreu, R. Carapuça, Object-oriented software engineering: Measuring and controlling the development process, in: Proceedings of the 4th international conference on software quality, Vol. 186, 1994.

[109] J. Bansiya, C. G. Davis, A hierarchical model for object-oriented design quality assessment, IEEE Transactions on software engineering 28 (1) (2002) 4–17.

[110] I. Crnkovic, M. P. H. Larsson, Building reliable component-based software systems, Artech House, 2002.

[111] V. L. Narasimhan, B. Hendradjaya, Some theoretical considerations for a suite of metrics for the integration of software components, Information Sciences 177 (3) (2007) 844–864.

[112] M. D.Weiser, Program slicing, in: Proceedings of the 5th International Conference on Software Engineering, 1981, pp. 439–449.

[113] M. Weiser, Program slicing, IEEE Transactions on software engineering (4) (1984) 352–357.

[114] B. Alqadi, Slice-based cognitive complexity metrics for defect prediction, Ph.D. thesis, Kent State University, USA (2020).

[115] L. M. Ott, J. J. Thuss, Slice based metrics for estimating cohesion, in: [1993] Proceedings First International Software Metrics Symposium, IEEE, 1993, pp. 71–81.

[116] S. Misra, A. Adewumi, L. Fernandez-Sanz, R. Damasevicius, A suite of object oriented cognitive complexity metrics, IEEE Access 6 (2018) 8782–8796.

[117] S. Scalabrino, M. Linares-Vasquez, D. Poshyvanyk, R. Oliveto, Improving code readability models with textual features, in: 2016 IEEE 24th International Conference on Program Comprehension (ICPC), IEEE, 2016, pp. 1–10.

[118] R. P. Buse, W. R. Weimer, Learning a metric for code readability, IEEE Transactions on Software Engineering 36 (4) (2009) 546–558.

[119] J. Dorn, A general software readability model, MCS Thesis available from (http://www. cs. virginia. edu/weimer/students/dorn-mcs-paper. pdf) 5 (2012) 11–14.

[120] S. E. S. Taba, F. Khomh, Y. Zou, A. E. Hassan, M. Nagappan, Predicting bugs using antipatterns, in: 2013 IEEE International Conference on Software Maintenance, IEEE, 2013, pp. 270–279.

[121] J. Shao, Y. Wang, A new measure of software complexity based on cognitive weights, Canadian Journal of Electrical and Computer Engineering 28 (2) (2003) 69–74.

[122] M. Fowler, Refactoring: improving the design of existing code, Addison-Wesley Professional, 2018.

[123] V. Gupta, J. K. Chhabra, Package level cohesion measurement in object-oriented software, Journal of the Brazilian Computer Society 18 (3) (2012) 251–266.

[124] A. Tripathi, D. Kushwaha, A metric for package level coupling, CSI transactions on ICT 2 (4) (2015) 217–233.

[125] N. Salman, Complexity metrics as predictors of maintainability and integrability of software components, Cankaya University Journal of Arts and Sciences 1 (5) (2006) 39–50.

[126] M. A. Boxall, S. Araban, Interface metrics for reusability analysis of components, in: 2004 Australian Software Engineering Conference. Proceedings., IEEE, 2004, pp. 40–51.

[127] J. A. McCall, M. T. Matsumoto, Software quality measurement manual. volume 2, Tech. rep., GENERAL ELECTRIC CO SUNNYVALE CA (1980).

[128] B. W. Boehm, J. R. Brown, M. Lipow, Quantitative evaluation of software quality, in: Proceedings of the 2nd international conference on Software engineering, 1976, pp. 592–605.

[129] R. Burrows, F. C. Ferrari, A. Garcia, F. Taïani, An empirical evaluation of coupling metrics on aspect-oriented programs, in: Proceedings of the 2010 ICSE Workshop on Emerging Trends in Software Metrics, 2010, pp. 53–58.

[130] A. Olszak, B. N. Jørgensen, Modularization compass navigating the white waters of feature-oriented modularity, in: 2014 9th International Conference on Software Engineering and Applications (ICSOFT-EA), IEEE, 2014, pp. 48–59.

[131] C. C. Aggarwal, Data mining: the textbook, Springer, 2015.

[132] Y. Singh, A. Kaur, R. Malhotra, Comparative analysis of regression and machine learning methods for predicting fault proneness models, International journal of computer applications in technology 35 (2-4) (2009) 183–193.

[133] C. Catal, U. Sevim, B. Diri, Clustering and metrics thresholds based software fault prediction of unlabeled program modules, in: 2009 Sixth International Conference on Information Technology: New Generations, IEEE, 2009, pp. 199–204.

[134] E. Arisholm, L. C. Briand, E. B. Johannessen, A systematic and comprehensive investigation of methods to build and evaluate fault prediction models, Journal of Systems and Software 83 (1) (2010) 2–17.

[135] R. Kohavi, G. H. John, Wrappers for feature subset selection, Artificial intelligence 97 (1-2) (1997) 273–324.

[136] R. Petrusel, J. Mendling, Eye-tracking the factors of process model comprehension tasks, in: International Conference on Advanced Information Systems Engineering, Springer, 2013, pp. 224–239.

[137] F. Brooks, H. Kugler, No silver bullet, 1987.