The Journal of Systems & Software 198 (2023) 111619

Contents lists available at ScienceDirect

The Journal of Systems & Software

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

On the relationship between source-code metrics and cognitive load: A systematic tertiary review☆,☆☆

Amine Abbad-Andaloussi

Institute of Computer Science, University of St. Gallen, 9000 St. Gallen, Switzerland

ARTICLE INFO

Article history:
Received 1 February 2022
Received in revised form 11 September 2022
Accepted 15 January 2023
Available online 17 January 2023

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 in developers' cognitive load, which can be estimated using psycho-physiological measures. Based on these measures, a handful of studies investigated the relationship between source-code metrics and cognitive load. For most of the metrics, such a relationship could not be established. While these studies used a small subset of metrics, the literature comprises hundreds of other metrics. Despite the existing reviews surveying these metrics, a consolidated overview is still needed to understand their properties and leverage their potential to align with cognitive load. This need is addressed in this paper through a Systematic Tertiary Review (STR) covering the full spectrum of source-code metrics, studying their properties and investigating their potential relationship to cognitive load. The outcome of this STR is intended to guide practitioners in choosing appropriate metrics, set the grounds for conceptualizing the relationship between source-code metrics and cognitive load and raise new research challenges for the future.

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 (Dumke and Ebert, 2007). The development of software systems underlies a series of implementation, comprehension, extension and maintenance tasks (Dooley and Dooley, 2017). The difficulty of these tasks depends on several factors, notably the characteristics of the source-code on which they are applied (Antinyan, 2020). 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 values indicating the extent to which the source-code possesses these attributes and supports the tasks whose difficulty relies on them (Anon, 1990; Dumke and Ebert, 2007; Mijač and Stapić, 2015; Wedyan and Abufakher, 2020; Arvanitou et al., 2017).

From a cognitive perspective, the difficulty of software development tasks can be associated with the cognitive load theory (Paas et al., 2003), 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 Paas et al., 2003). When dealing with complex artifacts (i.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 (Chen et al., 2016; Paas et al., 2003; Veltman and Jansen, 2005). Cognitive load can be captured using a wide array of measures (Chen et al., 2016), notably, those derived from modalities such as electroencephalogram (EEG), functional magnetic resonance imaging (fMRI), heart rate variability (HRV) and eye-tracking (Chen et al., 2016; McKiernan et al., 2003; Riedl and Léger, 2016; Holmqvist et al., 2011). These psycho-physiological measures have shown their robustness in estimating users' cognitive load in many studies within the software engineering field (Weber et al., 2021; Gonçales et al., 2021, 2019; Riedl et al., 2020).

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 (Peitek et al., 2021, 2018; Medeiros et al., 2019, 2021; Couceiro et al., 2019) in order to investigate the extent to which these metrics can estimate the difficulty of software

☆ Work supported by the International Postdoctoral Fellowship (IPF) Grant (Number: 1031574) from the University of St. Gallen, Switzerland.
☆☆ Editor: Dr. Shane McIntosh.
E-mail address: amine.abbad-andaloussi@unisg.ch.

https://doi.org/10.1016/j.jss.2023.111619

development tasks. However, the majority of the source-code metrics (e.g., Lines of Code (LOC), Mccabe's Cyclomatic Complexity (CC) McCabe, 1976, Halstead's metric suite Halstead, 1977) 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 (Jabangwe et al., 2015; Kaur, 2020; Arvanitou et al., 2017) at different levels of source-code granularity (e.g., method, class, package) (Catal and Diri, 2009; Colakoglu et al., 2021). These metrics have been organized in different categories based on a plenitude of characteristics (Bellini et al., 2008; Hall et al., 2011; Tahir and MacDonell, 2012; Nuñez-Varela et al., 2017) and have been used individually or combined in different settings (Jabangwe et al., 2015). Although dozens of literature reviews surveying source-code metrics exist, each of them focuses on specific quality attributes (Riaz et al., 2009; Radjenović et al., 2013; Malhotra and Bansal, 2015; Bandi et al., 2013), type of metrics (Abdellatief et al., 2013; Tahir and MacDonell, 2012) and language paradigm (Jabangwe et al., 2015; Burrows et al., 2009). 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 showing their potential to estimate developers' cognitive load, this paper proposes a Systematic Tertiary Review (STR) exploring the existing literature reviews and engaging with the wide spectrum of existing source-code metrics. Throughout this work, source-code metrics are organized and investigated across several dimensions, allowing for their similarities and disparities to emerge and thus shedding light on their characteristics. Overall, the contributions of this work (C1-C3) can be formulated as follows:

- C1: Conduct an STR study to develop a global overview of the literature on source-code metrics and their potential relationship to cognitive load.
- C2: Synthesize the findings of the STR study into a conceptual framework emphasizing the key notions identified through the study and describing the interplay between source-code metrics and cognitive load.
- C3: Based on the synthesis of the literature findings, identify and discuss the research gaps hindering the use of source-code metrics to estimate 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 and their potential to relate to cognitive load is studied. Furthermore, the granularity levels at which different metrics can be calculated are investigated and 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 (Anon, 1990), *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 (Anon, 1990) as a *"feature or characteristic that affects an item's* [i.e., software artifact] *quality"*. Examples of quality attributes are maintainability, understandability and testability. Quality attributes are typically seen as abstract concepts (Mendling, 2007). 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 (Mendling, 2007; Torgerson, 1958; Fenton and Pfleeger, 1997). 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"* (Anon, 1990). Notable examples of metrics are Lines of Code (LOC), Cyclomatic Complexity (CC) (McCabe, 1976) and those within the Halstead's suite (Halstead, 1977) and the Chidamber and Kemerer suite (*C&K*) (Chidamber and Kemerer, 1994).

Software measurements provide (at least) three key benefits, i.e., *understanding*, *control* and *improvement* of the software artifact (Park et al., 1996; Mendling, 2007; Borchert, 2008). 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 (Chidamber and Kemerer, 1994) 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 *improving* its maintainability.

2.2. Cognitive load theory

The *Cognitive load theory (Sweller, 2011)* 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* (Sternberg and Sternberg, 2016; Zheng, 2017; Baddeley and Hitch, 1974). The former refers to a store holding information for a long time. The latter refers to a buffer storing information temporarily 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 multi-dimensional construct representing the load imposed on the working memory during [the] performance of a cognitive task"* (Chen et al., 2016; Paas et al., 2003). 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* (Paas et al., 2003; Chen et al., 2016). 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 (Miller, 1956)). 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 (Paas et al., 2003; Chen et al., 2016; Sweller, 2011). 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 (Chen et al., 2016; Paas et al., 2003; Veltman and Jansen, 2005). 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 (Chen et al., 2016; Sweller, 2011). 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. Such a complexity is referred in the software engineering literature as the essential complexity (Brooks and Kugler, 1987; Antinyan, 2020). *Extraneous load*, in turn, arises from the way the operations and the artifact within the task are represented (Chen et al., 2016; Sweller, 2011). 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. This complicatedness is referred to in the literature as accidental complexity (Brooks and Kugler, 1987; Antinyan, 2020).

3. Methodology for the systematic tertiary review

A Systematic Tertiary Review (STR) (also called meta-review Schryen, 2010) is a review of existing secondary studies (e.g., literature reviews) summarizing the state-of-the-art literature in a particular field (Kitchenham, 2007). Kitchenham recommends the same rigorous methodology proposed for Systematic Literature Reviews (SLRs) when conducting STRs (Kitchenham, 2007). Accordingly, in the literature, many STRs (e.g., Nurdiani et al., 2016; Schryen, 2010; Sima et al., 2020) follow Kitchenham SLR guidelines (Kitchenham, 2007) or use similar guidelines (e.g., Webster and Watson, 2002, Snyder, 2019). Besides a few disparities at the operational level of the search protocol, both old (e.g., Kitchenham, 2007) and recent (e.g., Snyder, 2019) 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 Fig. 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), the inclusion and exclusion criteria used to select the relevant literature are defined (cf. Section 3.5) and the quality assessment criteria are set (cf. Section 3.6). Subsequently, the main literature search is conducted (cf. Section 3.7) and then enriched with a snowballing search (cf. Section 3.8). 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 3.9). This information is, afterward, analyzed following a qualitative approach (cf. Section 3.10).

3.1. Research questions

This work aims at investigating the state-of-the-art literature on source-code metrics with a particular focus on their relationship to cognitive load. To attain this objective the following research questions are formulated:

- *RQ1: What literature reviews about source-code metrics exist, how are these studies distributed over time, venue, publication type, review method and topic of review, and was the relationship between source-code metrics and cognitive load among these topics?*
  This research question allows identifying the literature reviews on source-code metrics and investigating their distribution over the aforementioned dimensions. Moreover, it reveals whether the relationship between source-code metrics and cognitive load has been *a core topic of review* in any of the published reviews. Answering this research question is important to discover the general trends in the literature.
- *RQ2: What research questions have been addressed in existing literature reviews and was the relationship between source-code metrics and cognitive load covered in any of these research questions?*
  Delving further into the identified reviews, this research question provides an overview of the questions raised in the literature and investigates whether the relationship between source-code metrics and cognitive load has been addressed (directly or indirectly) by any of these *research questions*. Answering this questions is motivated by the need for a wide overview of the literature and the necessity to pinpoint the relevant work on source-code metrics and cognitive load.
- *RQ3: What source-code metrics were proposed in the literature, how were these metrics categorized and what classes can be related to cognitive load?*
  This research question identifies the body of the existing source-code metrics and investigates the different ways they have been organized and categorized in the literature. Throughout these different categories, the extent to which various classes of source-code metrics can be linked to cognitive load is studied. The motivation behind this question is to explore this large body of metrics and identify the ones relevant for the scope of this STR.
- *RQ4: At what granularity can existing metrics be calculated?*
  In this research question, the identified metrics are examined to develop an overview of the different source-code granularity levels at which they can be applied. Answering this question is important as metric granularity can influence the alignment between source-code metrics

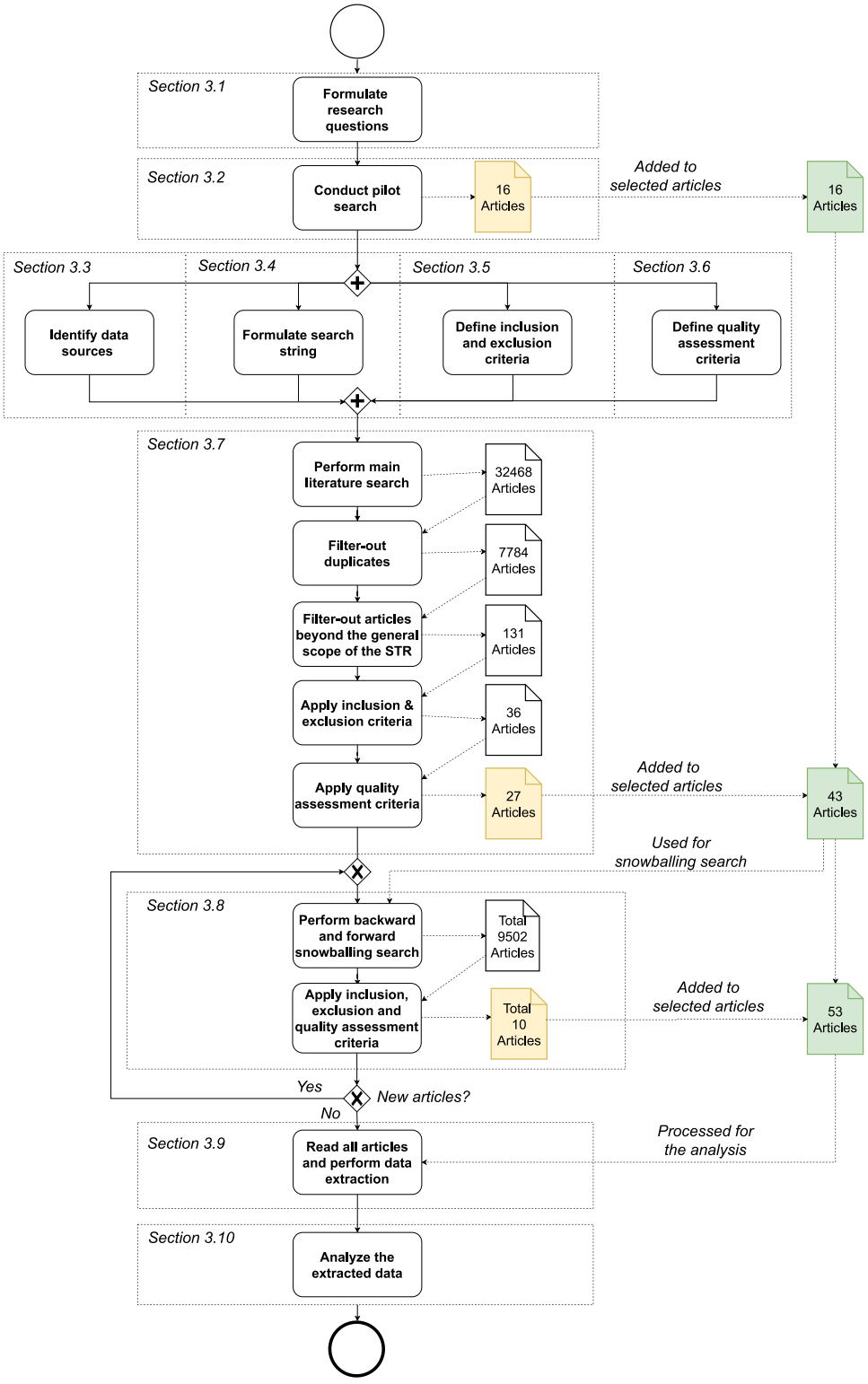


Fig. 1. Overview of the STR approach.

and developers' measurements of cognitive load. This is because, when engaging with software artifacts (e.g., conceptual models, source-code) developers do not browse the entire artifact but only the relevant parts of it (Petrusel and Mendling, 2013). Therefore, metrics that can potentially align with developers' cognitive load would need to characterize the source-code at a fine-grained level.

- *RQ5: What tools and projects can be used to extract source-code metrics and create benchmarking datasets for future experiments?*
  This research question is motivated by the need to provide means to extract source-code metrics and create benchmarking datasets that can be used to support future empirical studies linking source-code metrics to cognitive load.

Table 1
Sets of keywords using during the literature search.

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 2
Origin of the keywords composing the search string.

Study	Derived keywords
What is up with software metrics? – A preliminary mapping study (Kitchenham, 2010)	software, metrics
A Tool-Based Perspective on Software Code Maintainability Metrics: A Systematic Literature Review (Ardito et al., 2020)	source code, maintainability
Measurement in Software Engineering: From the Roadmap to the Crossroads (Bellini et al., 2008)	quality
Software Source Code Readability (Bexell, 2020)	readability
A Systematic Review Measurement in Software Engineering: State-of-the-Art in Measures (Gómez et al., 2006)	maintenance
A Systematic Literature Review on Empirical Analysis of the Relationship Between Code Smells and Software Quality Attributes (Kaur, 2020)	code smell, antipattern
Predicting Change Using Software Metrics: A Review (Malhotra and Bansal, 2015)	change
Source code metrics: A systematic mapping study (Nuñez-Varela et al., 2017)	fault, complexity, understandability, comprehension
Software defect prediction using bad code smells: A systematic literature review (Piotrowski and Madeyski, 2020)	bug, flaw, defect, error
A systematic literature review on the detection of smells and their evolution in object-oriented and service-oriented systems (Sabir et al., 2019)	technical debt

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., Kitchenham, 2010), while other articles were found through simple search queries (e.g., software metrics). Overall the pilot search resulted in the identification of 16 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 also been 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 (Gall et al., 2008; Hutton, 2009; Butler et al., 2010; Fakhoury et al., 2018). As for the keyword "code formatting", it was used in a set of studies investigating the indentation and the ordering of source-code elements (Hutton, 2009; Miara et al., 1983). All these factors have been shown to affect the readability of the source-code (Butler et al., 2010; Fakhoury et al., 2018; Miara et al., 1983; Hansen et al., 2013). 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 targeting the secondary studies in the literature were added (i.e., literature review, literature survey, mapping study, cf. "Keyword Set 2" in Table 1)

Seventy-two search strings were composed. These strings cover all possible concatenations (with the "AND" union, ignoring the order) of the terms in Keywords Set 1 and Keywords Set 2 reported in Table 1. They were used systematically to run the literature search on the data sources mentioned in Section 3.3.

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 systematic review of the literature (i.e., SLRs, Systematic Mapping Studies – SMSs Kitchenham, 2007).

3.5.1. Inclusion criteria

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

- **IC1**: Literature review covering metrics to evaluate the quality of source-code.
- **IC2**: Literature review covering tools to derive source-code metrics.

3.5.2. Exclusion criteria

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

- **EX1**: Literature review not published in English.
- **EX2**: Work-in-progress or duplicated articles (i.e., using the same data).
- **EX3**: Literature review focusing on metrics (e.g., product line metrics Montagud et al., 2012) requiring software artifacts beyond the source-code are excluded if they do not cover or cover marginally metrics based on source-code. Metrics operating at the level of the code interfaces (e.g., interface metrics Rotaru and Dobre, 2005) and metrics taking the evolution of source-code into account (e.g., code change metrics Moser et al., 2008) are still included.

In addition to the aforementioned exclusion criteria, the identified articles went through a quality assessment step which resulted in excluding low-quality literature reviews as explained in Section 3.6.

3.6. Quality assessment

Along with the inclusion and exclusion criteria described in Section 3.5, a set of quality assessment criteria was used to filter out low-quality literature reviews. Following the guidelines in Kitchenham (2007), the following quality criteria were defined:

- **C1**: Are the research questions clearly and explicitly stated?
- **C2**: Are the inclusion and exclusion criteria described appropriately in the review?
- **C3**: Is the literature search likely to have covered all relevant studies?
- **C4**: Did the authors of the review assess the quality/validity of the included studies?
- **C5**: Are the covered studies described adequately?

The aforementioned questions were answered using the following scale, i.e., *Yes*: 1, *No*: 0, *Partially*: 0.5. Thereafter, a quality score in the range of *[0:5]* was assigned to each article. This score was, in turn, used to decide on the article inclusion following a specific threshold. All articles with a quality score above or at *2.5* were included, while the rest were filtered out.

3.7. Main literature search and selection process

The main literature search yielded 32468 articles. After filtering out the duplicate ones, 7784 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., Wang et al., 2021; Nuñez-Varela et al., 2017; Mehboob et al., 2021; Alkharabsheh et al., 2019). 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 the author's colleagues from the software engineering field and refined to ensure 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 excluding 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 deciding on their inclusion or exclusion, other articles had to be fully read before being assessed. Overall, 36 articles were selected during this phase. Subsequently, these articles were assessed following the quality criteria defined in Section 3.6, which resulted in filtering out 3 articles and keeping 33 articles. As 6 of these articles were already identified in the pilot search, the final number of articles resulting from this phase was 27. These articles are reported in Table 4 (Source: M). It is worthwhile to mention that not all the articles from the pilot search were identified during this phase of the search as they may have appeared in venues that are not covered by the used search engines or have been indexed with different keywords. Nevertheless, these articles were still relevant and thus they were included in this STR. The list of the articles examined during the main search mapped to the inclusion, exclusion and quality assessment criteria is available online[1].

3.8. Snowballing search

Backward and forward snowballing were conducted over the articles selected from the main search. Initially, 5285 articles were identified from the backward search and 3954 articles were identified from the forward search. These articles were processed following the inclusion and exclusion criteria defined in Section 3.5. 15 articles were found to be relevant (8 from the backward search and 7 from the forward search), among them 10 (8 from the backward search and 2 from the forward search) met the quality assessment criteria defined in Section 3.6.

To ensure completeness, the snowballing search was conducted until saturation. Therein, the newly identified 10 articles went through a snowballing backward and forward search, which has resulted in identifying 209 and 54 articles respectively. Among these articles, 2 (from the backward search) have met the defined inclusion and exclusion criteria but did not fulfill the quality assessment ones. The list of the articles examined during the snowballing search mapped to the inclusion, exclusion and quality assessment criteria is available online[1].

All in all, throughout both rounds of the snowballing search, 9502 articles were identified and 10 were selected as relevant for this STR study (cf. Table 4 – Source: B(Backward)S and F(Forward)S).

3.9. 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.

3.10. Analysis procedure

A qualitative data analysis inspired by Strauss and Corbin's approach to grounded theory (Corbin and Strauss, 2014) 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* (Corbin and Strauss, 2014) allowed to highlight the salient aspects in the review articles. (2) *Focused coding* (Corbin

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

Table 3
Data extraction scheme.

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	Metrics	List of metrics
	Categorizations	Types of categorization
		Example of metrics
	Quality attributes	Attribute
		Type
		Example of metrics
RQ4	Granularity levels	Level of granularity
		Examples of metrics
RQ5	Tools	Tool name
		URL
		Academic or commercial
		Delivers metrics or code smells
		Languages supported
		Examples of metrics extracted
	Projects	Project name
		Link
		Type
		Language

and Strauss, 2014) enabled to group these aspects based on their similarities. (3) *Axial coding* (Corbin and Strauss, 2014) 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), initial codes were assigned to each review article based on its meta-information (i.e., title, abstract and keywords). Afterward, several rounds of focused and axial coding were conducted to group and relate the articles covering the same research topic. Similarly, to develop a clear overview of the research questions investigated in the literature (RQ2), several rounds of initial, focused and axial coding 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, 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 source-code metrics and their potential link to cognitive load. Section 4.4 discerns the different granularity levels at which metrics can be calculated. Section 4.5 identifies the popular projects and tools to extract source-code metrics.

4.1. What literature reviews about source-code metrics exist, how are these studies distributed over time, venue, publication type, review method and topic of review, and was the relationship between source-code metrics and cognitive load among these topics? (RQ1)

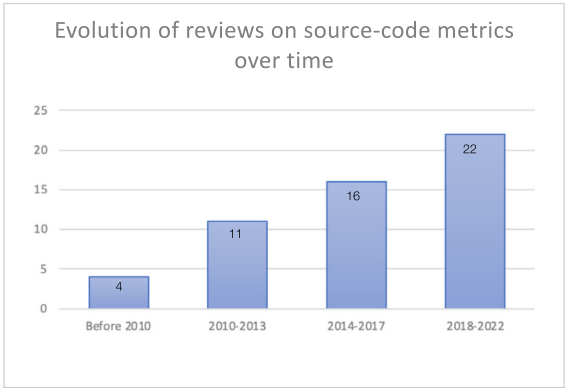
The search protocol allowed the identification of 53 literature reviews covering source-code metrics. These reviews are reported in Table 4. The distribution of review articles over time depicted in Fig. 2(a) shows that source-code metrics have been reviewed with an increasing trend over the last two decades. Interestingly, the last four years denote the period with the highest number of reviews on source-code metrics. Fig. 2(b) 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), *"Journal of Software: Evolution and Process"* and *"Archives of Computational Methods in Engineering"* (journal) (i.e., 2 papers, 4% each). Fig. 2(c) shows that the majority of reviews (i.e., 30 papers, 57%) have been published in journal venues. Nevertheless, a significant number of reviews (i.e., 22 papers, 41%) have appeared in conference venues. When it comes to the distribution over research method, Fig. 2(d) shows that the majority of reviews used an SLR approach (Kitchenham, 2007) (i.e., 32 papers, 54%), while a smaller portion of studies (i.e., 17 papers, 29%) followed an SMS approach (Kitchenham, 2007).

Fig. 3 depicts a scheme organizing the different topics addressed by the identified literature reviews. At a high level, the reviews can be divided into (1) those *conceptualizing software measurements and providing an overview of the literature trends* (Kitchenham, 2010; Bellini et al., 2008; Gómez et al., 2006), (2) those focusing on *specific classes of metrics or metrics capturing specific quality attributes* (AbuHassan et al., 2021; Alkharabsheh et al., 2019; Fernandes et al., 2016; Kaur, 2020; Rasool and Arshad, 2015; Rattan and Kaur, 2016; dos Reis et al., 2021; Sabir et al., 2019; Zaidi and Colomo-Palacios, 2019; Abdellatief et al., 2013; Tahir and MacDonell, 2012; Burrows et al., 2009;

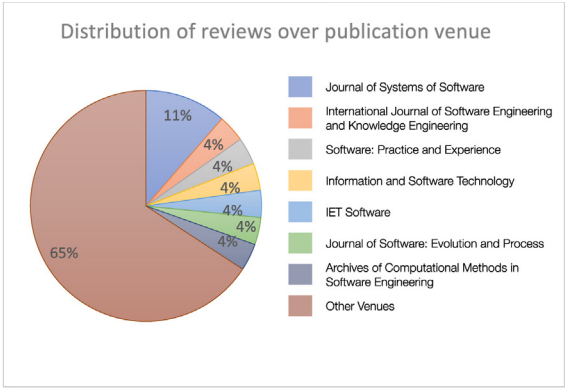
Table 4
Literature reviews selected during the pilot search, main literature search and snowballing search. Abbreviations: Ref.: Full Reference, M: Main literature Search, P: Primary Search, BS: Backward snowballing search, FS: Forward snowballing search.

Title	Ref.	Source
Software Product Quality Metrics: A Systematic Mapping Study	Colakoglu et al. (2021)	M
Software smell detection techniques: A systematic literature review	AbuHassan et al. (2021)	M
Can we benchmark Code Review studies? A systematic mapping study of methodology, dataset, and metric	Wang et al. (2021)	M
Reusability affecting factors and software metrics for reusability: A systematic literature review	Mehboob et al. (2021)	M
A Systematic Literature Review on Empirical Analysis of the Relationship Between Code Smells and Software Quality Attributes	Kaur (2020)	P
Code smells detection and visualization: A systematic literature review	dos Reis et al. (2021)	P
Software Source Code Readability	Bexell (2020)	P
Software Defect Prediction Using Bad Code Smells: A Systematic Literature Review	Piotrowski and Madeyski (2020)	P
A Tool-Based Perspective on Software Code Maintainability Metrics: A Systematic Literature Review	Ardito et al. (2020)	P
Impact of Design Patterns on Software Quality: A Systematic Literature Review	Wedyan and Abufakher (2020)	M
Quality and Success in Open Source Software: A Systematic Mapping	Gezici et al. (2019)	M
A survey on Software Coupling Relations and Tools	Fregnan et al. (2019)	M
Quality Metrics in Software Design: A Systematic Review	Hernandez-Gonzalez et al. (2019)	M
Code Smells Enabled by Artificial Intelligence: A Systematic Mapping	Zaidi and Colomo-Palacios (2019)	M
An analysis of measurement and metrics tools: A systematic literature review	Dias Canedo et al. (2019)	FS
Software Quality Assessment Model: A Systematic Mapping Study	Yan et al. (2019)	M
Software Design Smell Detection: A Systematic Mapping Study	Alkharabsheh et al. (2019)	M
Construction of a Software Measurement Tool Using Systematic Literature Review	Valença et al. (2018)	M
Progress on Approaches to Software Defect Prediction	Li et al. (2018)	M
Coupling and Cohesion Metrics for Object-Oriented Software: A Systematic Mapping Study	Tiwari and Rathore (2018)	M
Early Software Defect Prediction: A Systematic Map and Review	Özakıncı and Tarhan (2018)	M
A Systematic Literature Review on the Detection of Smells and their Evolution in Object-Oriented and Service-Oriented Systems	Sabir et al. (2019)	P, M
A Systematic Review of Software Usability Studies	Sagar and Saha (2017)	M
A Mapping Study on Design-time Quality Attributes and Metrics	Arvanitou et al. (2017)	M
Source Code Metrics: A Systematic Mapping Study	Nuñez-Varela et al. (2017)	P, M
Maintenance Effort Estimation for Open Source Software: A Systematic Literature Review	Wu et al. (2016)	M
Metrics and Statistical Techniques Used to Evaluate Internal Quality of Object-oriented Software: A Systematic Mapping	Santos et al. (2016)	M
Systematic Mapping Study of Metrics Based Clone Detection Techniques	Rattan and Kaur (2016)	M
A Review-Based Comparative Study of Bad Smell Detection Tools	Fernandes et al. (2016)	M
Software maintainability: Systematic literature review and current trends	Malhotra and Chug (2016)	FS
Software Code Maintainability: A Literature Review	Seref and Tanriover (2016)	P
A systematic Review of Machine Learning Techniques for Software Fault Prediction	Malhotra (2015)	BS
Reusability Metrics of Software Components: Survey	Mijač and Stapić (2015)	BS
Software Fault Prediction: A Systematic Mapping Study	Murillo-Morera et al. (2015)	BS
Predicting Change Using Software Metrics: A Review	Malhotra and Bansal (2015)	P
A Review of Code Smell Mining Techniques	Rasool and Arshad (2015)	M
Empirical Evidence on the Link Between Object-oriented Measures and External Quality Attributes: A Systematic Literature Review	Jabangwe et al. (2015)	BS
A Report on the Analysis of Metrics and Measures on Software Quality Factors – A Literature Study	Vanitha and ThirumalaiSelvi (2014)	BS
Open Source Tools for Measuring the Internal Quality of Java Software Products. A survey	Tomas et al. (2013)	M
Software Fault Prediction Metrics: A Systematic Literature Review	Radjenović et al. (2013)	P, M
A Systematic Review of the Empirical Validation of Object-Oriented Metrics toward Fault-proneness Prediction	Isong and Obeten (2013)	BS
A Mapping Study to Investigate Component-based Software System Metrics	Abdellatief et al. (2013)	M
Empirical Evidence of Code Decay: A Systematic Mapping Study	Bandi et al. (2013)	M
A Systematic Mapping Study on Dynamic Metrics and Software Quality	Tahir and MacDonell (2012)	M
A Systematic Literature Review on Fault Prediction Performance in Software Engineering	Hall et al. (2011)	P
A Systematic Review on the Impact of CK Metrics on the Functional Correctness of Object-Oriented Classes	Khan et al. (2012)	BS
Aspect-oriented Software Maintenance Metrics: A Systematic Mapping Study	Saraiva et al. (2012)	P, M
Coupling Metrics for Aspect-Oriented Programming: A Systematic Review of Maintainability Studies	Burrows et al. (2009)	M
What is Up with Software Metrics? – A Preliminary Mapping Study	Kitchenham (2010)	P, M
A Systematic Review of Software Fault Prediction Studies	Catal and Diri (2009)	P
A Systematic Review of Software Maintainability Prediction and Metrics	Riaz et al. (2009)	BS
Measurement in Software Engineering: From the Roadmap to the Crossroads	Bellini et al. (2008)	P
A Systematic Review Measurement in Software Engineering: State-of-the-Art in Measures	Gómez et al. (2006)	P, M

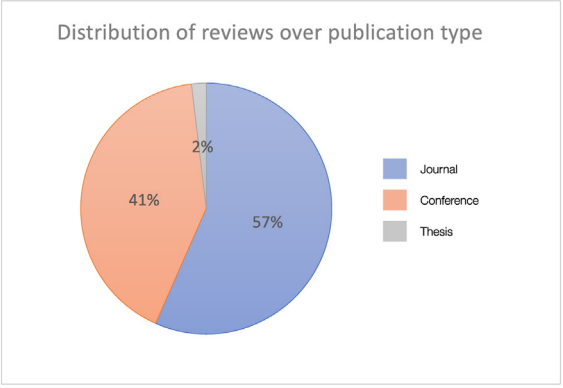
Fregnan et al., 2019; Tiwari and Rathore, 2018; Ardito et al., 2020; Riaz et al., 2009; Saraiva et al., 2012; Seref and Tanriover, 2016; Wu et al., 2016; Bexell, 2020; Malhotra and Bansal, 2015; Bandi et al., 2013; Piotrowski and Madeyski, 2020; Isong and Obeten, 2013; Catal et al., 2009; Hall et al., 2011; Li et al., 2018; Malhotra, 2015; Murillo-Morera et al., 2015; Özakıncı and Tarhan, 2018; Radjenović et al., 2013; Khan et al., 2012; Mehboob et al., 2021; Mijač and Stapić, 2015; Sagar and Saha, 2017), (3), those *targeting a wider set of metrics spanning over different classes and capturing many other quality attributes (i.e., generic metrics)* (Wang et al., 2021; Gezici et al., 2019; Colakoglu et al., 2021; Jabangwe et al., 2015; Santos et al., 2016; Arvanitou et al., 2017; Hernandez-Gonzalez et al., 2019; Wedyan and Abufakher, 2020; Nuñez-Varela et al., 2017; Vanitha and ThirumalaiSelvi, 2014; Yan et al., 2019) and (4) those interested in *tools collecting source-code metrics* Tomas et al. (2013), Valença et al. (2018),



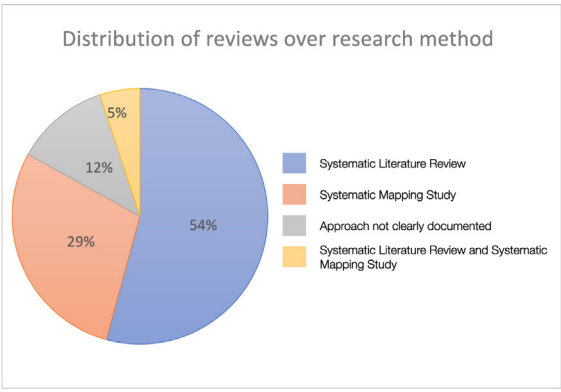
(a) Distribution of existing reviews over time.



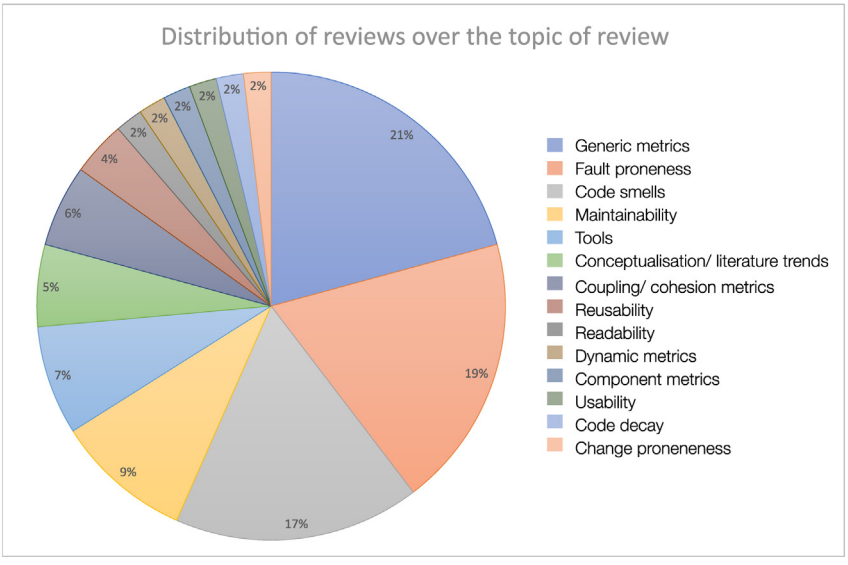
(b) Distribution of reviews over publication venue



(c) Distribution of reviews over publication type



(d) Distribution of reviews over research method



(e) Distribution of reviews over topic of review

Fig. 2. Different distributions of the reviews in Table 4.

Malhotra and Chug (2016), Dias Canedo et al. (2019). At a more fine-grained level, the reviews in (2) can be divided into reviews targeting *specific quality attributes*, which, in turn can be *internal* (i.e., capturing structural properties of the source-code Glinz, 2014) or *external* (i.e., capture the quality of the source-code as perceived by its stakeholders Glinz, 2014). As shown in Fig. 3, each of these categories includes a set of attributes, i.e., *coupling/cohesion* (Burrows et al., 2009; Fregnan et al., 2019; Tiwari and Rathore, 2018), *maintainability* (Ardito et al., 2020; Riaz et al., 2009; Saraiva et al., 2012; Seref and Tanriover, 2016; Wu et al., 2016), *readability* (Bexell, 2020), *change proneness* (Malhotra and Bansal, 2015), *code decay* (Bandi et al., 2013), *fault proneness*

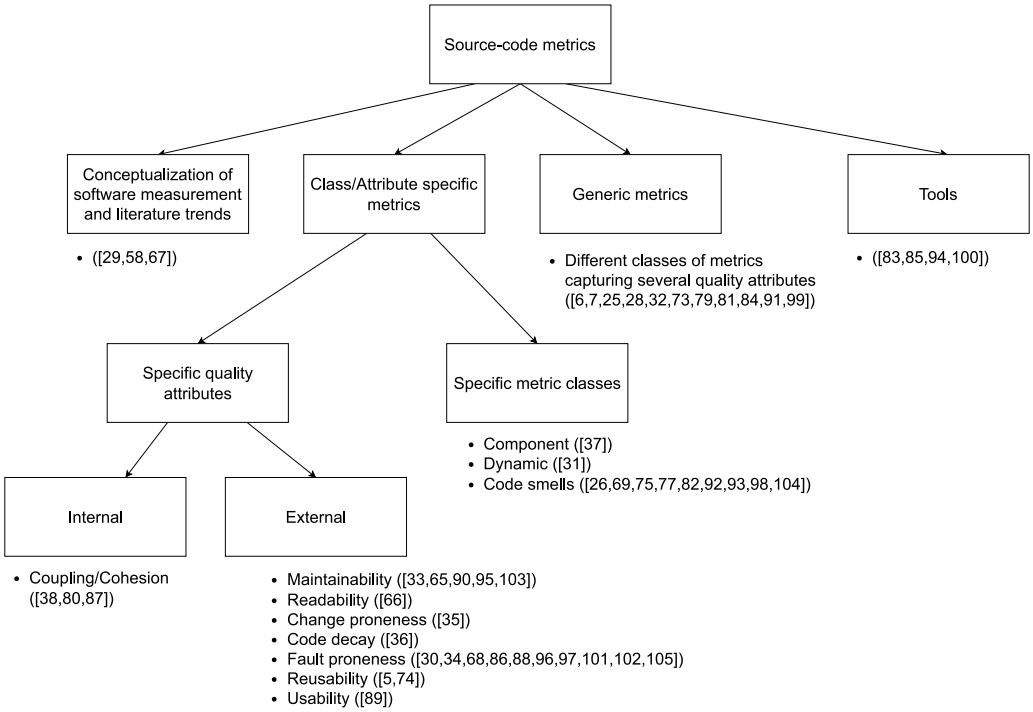


Fig. 3. Topics of review.

(Piotrowski and Madeyski, 2020; Isong and Obeten, 2013; Catal et al., 2009; Hall et al., 2011; Li et al., 2018; Malhotra, 2015; Murillo-Morera et al., 2015; Özakıncı and Tarhan, 2018; Radjenović et al., 2013; Khan et al., 2012), *reusability* (Mehboob et al., 2021; Mijač and Stapić, 2015), *usability* (Sagar and Saha, 2017). As for the reviews emphasizing *specific classes of metrics*, three topics could be identified, i.e., *components metrics* (Abdellatief et al., 2013), *dynamic metrics* (Tahir and MacDonell, 2012) and *code smell metrics* (AbuHassan et al., 2021; Alkharabsheh et al., 2019; Fernandes et al., 2016; Kaur, 2020; Rasool and Arshad, 2015; Rattan and Kaur, 2016; dos Reis et al., 2021; Sabir et al., 2019; Zaidi and Colomo-Palacios, 2019).

The distribution of the identified topics of review is shown in Fig. 2(e). Overall, generic metrics (11 papers, 21%), fault proneness metrics (10 papers, 19%) and code smell metrics (9 papers, 17%) are the most popular topics emphasized in the existing literature reviews.

Based on the findings reported in Fig. 3, while the existing literature reviews covered a wide range of topics, the relationship between source-code metrics and cognitive load has not been a core topic in any of these reviews.

4.2. What research questions have been addressed in existing literature reviews and was the relationship between source-code metrics and cognitive load covered in any of these research questions? (RQ2)

The literature reviews identified in this STR study address a wide array of research questions. Following the qualitative approach described in Section 3.10, 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.

On the relationship between source-code metrics and cognitive load, only a single research question is relevant (i.e., "*How do some metrics relate to the concept of cognitive complexity?*" cf. Table 5). This question was addressed as part of RQ4 in Isong and Obeten (2013). The authors defined the concept of cognitive complexity as *"a measure of the mental burden imposed on the persons (developers, testers, inspectors, maintainers, etc.) who have to deal with the component"*. Such a definition is very similar to the one we adopt for cognitive load (i.e., *"... the load imposed on the working memory during [the] performance of a cognitive task"* Chen et al., 2016; Paas et al., 2003). The authors posit that issues associated with the source-code structure (e.g., coupling, cohesion and inheritance) impact developers' cognitive complexity (i.e., cognitive load) negatively, which, in turn, leads to faulty components with low quality (e.g., in terms of understandability and maintainability). These issues can be detected using several metrics, notably, those from the C&K suite (Chidamber and Kemerer, 1994). Hence, through them, one can identify the critical software components in the system and develop mechanisms to prevent its degradation and improve its quality (Isong and Obeten, 2013).

4.3. What source-code metrics were proposed in the literature, how were these metrics categorized and what classes can be related to cognitive load? (RQ3)

Based on the findings of the covered reviews, this section identifies the existing metrics (cf. Section 4.3.1), enumerates the different categorizations proposed in the literature (cf. Section 4.3.2) and investigates how they could relate to cognitive load (c. Section 4.3.3).

4.3.1. Identification of source-code metrics

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

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

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

Theme	Category	Key questions	#	References
Metrics	Lists	What source-code metrics have been proposed in the literature?	38	Abdellatief et al. (2013), AbuHassan et al. (2021), Ardito et al. (2020), Bandi et al. (2013), Bexell (2020), Burrows et al. (2009), Catal and Diri (2009), Gezici et al. (2019), Hernandez-Gonzalez et al. (2019), Isong and Obeten (2013), Jabangwe et al. (2015), Kaur (2020), Malhotra and Bansal (2015), Malhotra (2015), Mehboob et al. (2021), Mijač and Stapić (2015), Nuñez-Varela et al. (2017), Özakıncı and Tarhan (2018), Piotrowski and Madeyski (2020), Radjenović et al. (2013), Rasool and Arshad (2015), Rattan and Kaur (2016), dos Reis et al. (2021), Riaz et al. (2009), Sabir et al. (2019), Sagar and Saha (2017), Santos et al. (2016), Saraiva et al. (2012), Seref and Tanriover (2016), Tahir and MacDonell (2012), Tiwari and Rathore (2018), Tomas et al. (2013), Valença et al. (2018), Vanitha and ThirumalaiSelvi (2014), Wedyan and Abufakher (2020), Wu et al. (2016), Dias Canedo et al. (2019), Malhotra and Chug (2016)
Metrics	Validations	What empirical studies have evaluated the relationship between source-code metrics and different quality attributes?	16	Abdellatief et al. (2013), Arvanitou et al. (2017), Bandi et al. (2013), Bexell (2020), Burrows et al. (2009), Gómez et al. (2006), Isong and Obeten (2013), Jabangwe et al. (2015), Khan et al. (2012), Kitchenham (2010), Malhotra and Bansal (2015), Mehboob et al. (2021), Radjenović et al. (2013), Riaz et al. (2009), Seref and Tanriover (2016), Tiwari and Rathore (2018)
Metrics	Categorizations	How have these metrics been categorized in the literature?	10	Bellini et al. (2008), Fregnan et al. (2019), Gezici et al. (2019), Gómez et al. (2006), Hall et al. (2011), Nuñez-Varela et al. (2017), Tahir and MacDonell (2012), Wang et al. (2021), Wu et al. (2016), Yan et al. (2019)
Metrics	Attributes	What quality attributes are captured by existing source-code metrics?	9	Arvanitou et al. (2017), Gezici et al. (2019), Jabangwe et al. (2015), Kaur (2020), Mehboob et al. (2021), Mijač and Stapić (2015), Tiwari and Rathore (2018), Vanitha and ThirumalaiSelvi (2014), Wedyan and Abufakher (2020)
Metrics	Granularity	At what granularity levels can the different source-code metrics be calculated?	5	Abdellatief et al. (2013), Burrows et al. (2009), Catal and Diri (2009), Colakoglu et al. (2021), Mehboob et al. (2021)
Metrics	Development phases	What source-code metrics are extracted on each of the software development phases?	4	Arvanitou et al. (2017), Gómez et al. (2006), Radjenović et al. (2013), Riaz et al. (2009)
Metrics	Studies contexts and applications	Which studies and applications are carried by means of source-code metrics?	3	Nuñez-Varela et al. (2017), Tahir and MacDonell (2012), Tiwari and Rathore (2018)
Metrics	Thresholds	Are there known thresholds for the existing source-code metrics?	2	Colakoglu et al. (2021), Malhotra and Bansal (2015)
Metrics	Range	What range of values do these metrics show and what do they mean?	1	Hernandez-Gonzalez et al. (2019)
Metrics	Paradigm	Which programming paradigms are currently measured by source-code metrics?	1	Nuñez-Varela et al. (2017)
Metrics	Relation to cognitive complexity	How do some metrics relate to the concept of cognitive complexity?	1	Isong and Obeten (2013)
Metrics	Trends	Are new source-code metrics being proposed?	1	Nuñez-Varela et al. (2017)
Metrics	Directions for future work	Which aspects of dynamic metrics could be recommended as topics for future research?	1	Tahir and MacDonell (2012)
Approaches	Lists	What approaches have been proposed in the literature to evaluate the quality of source-code?	19	Alkharabsheh et al. (2019), Bandi et al. (2013), Catal and Diri (2009), Colakoglu et al. (2021), Hall et al. (2011), Isong and Obeten (2013), Jabangwe et al. (2015), Li et al. (2018), Malhotra and Bansal (2015), Malhotra (2015), Murillo-Morera et al. (2015), Özakıncı and Tarhan (2018), Rattan and Kaur (2016), dos Reis et al. (2021), Sabir et al. (2019), Santos et al. (2016), Wu et al. (2016), Yan et al. (2019), Zaidi and Colomo-Palacios (2019)
Approaches	Validations	How have these approaches been validated and what are the results of these validations?	12	Hall et al. (2011), Kaur (2020), Li et al. (2018), Malhotra and Bansal (2015), Malhotra (2015), Murillo-Morera et al. (2015), Radjenović et al. (2013), dos Reis et al. (2021), Riaz et al. (2009), Wu et al. (2016), Yan et al. (2019), Malhotra and Chug (2016)
Approaches	Issues	What are the challenges of the existing approaches?	2	Yan et al. (2019), Zaidi and Colomo-Palacios (2019)

(*continued on next page*)

Table 5 (*continued*).

Theme	Category	Key questions	#	References
Tools	Lists	What tools are extracting source-code metrics?	14	Alkharabsheh et al. (2019), Ardito et al. (2020), Arvanitou et al. (2017), Colakoglu et al. (2021), Fernandes et al. (2016), Fregnan et al. (2019), Mehboob et al. (2021), Nuñez-Varela et al. (2017), Rasool and Arshad (2015), Rattan and Kaur (2016), Valença et al. (2018), Yan et al. (2019), Dias Canedo et al. (2019), Malhotra and Chug (2016)
Tools	Categorizations	How have these tools been organized in the literature?	3	Ardito et al. (2020), Yan et al. (2019), Dias Canedo et al. (2019)
Tools	Features	What are the features of these tools?	2	Fernandes et al. (2016), Valença et al. (2018)
Tools	Maturity	What is the degree of maturity of these tools?	1	Tomas et al. (2013)
Projects	Lists	What projects have been used in studies covering source-code metrics?	6	Kaur (2020), Li et al. (2018), Malhotra (2015), Nuñez-Varela et al. (2017), Wu et al. (2016), Malhotra and Chug (2016)
Projects	Categorizations	How have these projects been categorized in the literature?	4	Catal and Diri (2009), Malhotra and Bansal (2015), Nuñez-Varela et al. (2017), Radjenović et al. (2013)
Projects	Size	What is the size of the projects used in studies covering source-code metrics?	2	Radjenović et al. (2013), Wu et al. (2016)
Projects	Languages	In what programming languages are these projects?	2	Nuñez-Varela et al. (2017), Radjenović et al. (2013)
Projects	Fault distribution	What is the fault distribution in the datasets used for software fault prediction?	1	Malhotra (2015)

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. LOC (or SLOC, referring to Source-code Lines of Code Nguyen et al., 2007) and the Halstead's metrics suite (Halstead, 1977) are among the basic and oldest metrics used for a wide variety of source-code languages belonging to different paradigms. McCabe's Cyclomatic Complexity (CC) (McCabe, 1976), in turn, was rather tailored for flow-based languages where the control-flow is explicit meaning that it can be represented as a sequence of actions leading to a specific output (Winograd, 1975), as opposed to constraint-based languages where the control-flow is implicit and rather represented as a set of constraints that must hold during the program execution (Lloyd, 1994). With the rise of Object-oriented (OO) programming, new suites such as the Chidamber and Kemerer C&K suite (Chidamber and Kemerer, 1994), Li and Henry suite (Li and Henry, 1993), Lorenz and Kidd suite (Lorenz and Kidd, 1994), Metrics for Object-oriented Design (MOOD) suite (Abreu and Carapuça, 1994) and Quality Metrics for Object-oriented Design (QMOOD) suite (Bansiya and Davis, 2002) 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 (Crnkovic and Larsson, 2002). 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 (Abdellatief et al., 2013). 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 (Narasimhan and Hendradjaya, 2007) (cf. Table 6).

Slice-based metrics. Another branch of metrics is based on the program slicing technique proposed by Weiser (Weiser, 1981, 1984). 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 (Weiser, 1984). 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 (Weiser, 1984; Alqadi, 2020). In Weiser (1981), Weiser introduced 5 slice-based metrics. This set was extended by two other metrics proposed by Ott and Thuss (1993) (cf. Table 6).

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 (2003). 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 (Shao and Wang, 2003). 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, recursion, concurrency) based on their presumed difficulty. For instance, a for-do iteration is more difficult, according to Shao and Wang (2003), than a sequence-flow structure and hence, it is assigned a higher cognitive weight. Following the work of Shao and Wang (2003), Misra et al. (2018) 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. (2016) 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 (2009), in turn, used a simpler set of readability metrics (cf. Table 6) which do not require NLP. Their metrics capture several

Table 6
Representative metric suites.

Suite	Metrics
Halstead's suite (Halstead, 1977)	Program vocabulary (η), length (N), volume (V), difficulty (D), effort (E)
Chidamber and Kemerer (C&K suite) (Chidamber and Kemerer, 1994)	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 (Li and Henry, 1993)	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 (Lorenz and Kidd, 1994)	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 (Abreu and Carapuça, 1994)	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 (Bansiya and Davis, 2002)	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 (Narasimhan and Hendradjaya, 2007)	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 Cycles (NC), Average Number of Active Component (ANAC), Active Component Density (ACD), Average Active Component Density (AACD), Peak Number of Active Components (PNAC)
Weiser metrics (Weiser, 1981)	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 (Ott and Thuss, 1993)	Coverage of the shortest slice (MinCoverage), Coverage of the longest slice (MaxCoverage)
Misra et al. suite (Misra et al., 2018)	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 (Scalabrino et al., 2016)	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 (Buse and Weimer, 2009)	Length of source-code lines and identifiers, count of identifiers, keywords, digits, comments, periods, commas, spaces, parentheses, arithmetic operators, comparison operators, assignments, branches, loops, blank lines; occurrences of single characters and identifiers, indentation in the source-code
Dorn (Dorn, 2012)	Large set of structural, syntactical, visual, perceptional and alignment features. Details in Dorn (2012).
Taba et al. metrics (Taba et al., 2013)	Average Number of Antipatterns ($ANA_{Taba\ et\ al.}$), Antipattern Complexity Metric (ACM), Antipattern Recurrence Length (ARL), Antipattern Cumulative Pairwise Differences (ACPD)
Moser et al. suite (Moser et al., 2008)	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)

properties related to the count and length of different code tokens and the indentation in the source-code. In the same vein, Dorn (2012) 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 (2018), these antipatterns indicate poor design solutions and implementation issues in the source-code. Taba et al. (2013) proposed a suite of metrics based on the antipatterns identified in the source-code (cf. Table 6). The 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 indications 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 (Moser et al., 2008) (cf. Table 6).

4.3.2. Categorization of metrics

Source-code metrics have been categorized differently in the literature. Notably, in several studies, metrics were organized by the **quality attributes** they capture (Wang et al., 2021; Gezici

Table 7
Representative classes of metrics.

Class	Examples of metrics and suites
Basic metrics	Lines of Code (LOC), McCabe's Cyclomatic Complexity (CC) (McCabe, 1976), Halstead's suite (Halstead, 1977)
Object-oriented metrics	Chidamber and Kemerer (C&K) suite (Chidamber and Kemerer, 1994), Li and Henry suite (Li and Henry, 1993), Lorenz and Kidd suite (Lorenz and Kidd, 1994), Metrics for Object-oriented Design (MOOD) suite (Abreu and Carapuça, 1994), Quality Metrics for Object-oriented Design (QMOOD) suite (Bansiya and Davis, 2002)
Component-based metrics	Narasimhan and Hendradjaya suite (Narasimhan and Hendradjaya, 2007)
Slice-based metrics	Weiser suite (Weiser, 1981), Ott and Thuss suite (Ott and Thuss, 1993)
Cognitive metrics	Shao and Wang suite (Shao and Wang, 2003), Misra et al. suite (Misra et al., 2018)
Textual and readability metrics	Scalabrino et al. suite (Scalabrino et al., 2016), Buse and Weimer suite (Buse and Weimer, 2009), Dorn suite (Dorn, 2012)
Anti-pattern metrics	Taba et al. suite (Taba et al., 2013)
Code change metrics	Moser et al. suite (Moser et al., 2008)

et al., 2019; Colakoglu et al., 2021; Jabangwe et al., 2015; Santos et al., 2016; Arvanitou et al., 2017; Hernandez-Gonzalez et al., 2019; Wedyan and Abufakher, 2020; Nuñez-Varela et al., 2017; Vanitha and ThirumalaiSelvi, 2014; Yan et al., 2019; Burrows et al., 2009; Fregnan et al., 2019; Tiwari and Rathore, 2018; Ardito et al., 2020; Riaz et al., 2009; Saraiva et al., 2012; Seref and Tanriover, 2016; Wu et al., 2016; Bexell, 2020; Malhotra and Bansal, 2015; Bandi et al., 2013; Piotrowski and Madeyski, 2020; Isong and Obeten, 2013; Catal et al., 2009; Hall et al., 2011; Li et al., 2018; Malhotra, 2015; Murillo-Morera et al., 2015; Özakıncı and Tarhan, 2018; Radjenović et al., 2013; Khan et al., 2012; Mehboob et al., 2021; Mijač and Stapić, 2015; Sagar and Saha, 2017). These attributes can be organized into *internal* and *external* attributes. Internal attributes are meant to capture the structural properties of the source-code (Glinz, 2014). The attributes that have been classified in the existing reviews as being internal (Jabangwe et al., 2015; Kaur, 2020) 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 (Glinz, 2014). 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 (Kaur, 2020; Jabangwe et al., 2015; McCall and Matsumoto, 1980; Arvanitou et al., 2017; Boehm et al., 1976). Tables 8 and 9 summarize the identified internal and external quality attributes respectively and provide examples of metrics capturing each of them.

Among the identified metrics, the *C&K suite* (Chidamber and Kemerer, 1994) 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 (Bansiya and Davis, 2002) 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* (Abreu and Carapuça, 1994), Lorenz and Kidd (1994), and *Li and Henry suites* (Li and Henry, 1993) 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* (Buse and Weimer, 2009), *Dorn suite* (Dorn, 2012), *Scalabrino et al. suite* (Scalabrino et al., 2016)). Similarly, cognitive complexity is captured using a particular set of metrics including those in the Misra et al. (2018) suite and the *CFS* metric of Shao and Wang (2003).

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 (Tiwari and Rathore, 2018; Mehboob et al., 2021; Vanitha and ThirumalaiSelvi, 2014).

Beside characterizing metrics based on their quality attributes, some authors have distinguished between **product** and **process** metrics (Bellini et al., 2008; Hall et al., 2011). Product metrics capture properties associated directly with the source-code (e.g., *C&K* Chidamber and Kemerer, 1994, *MOOD* Abreu and Carapuça, 1994, *QMOOD* Bansiya and Davis, 2002, Lorenz and Kidd (1994), Li and Henry (1993) suites), while process metrics capture properties associated with the process of editing the source-code (e.g., code change metrics in the Moser et al., 2008 suite).

Another categorization in the literature discerns **static** and **dynamic** metrics (Tahir and MacDonell, 2012). 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 (2007), 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 (Narasimhan and Hendradjaya, 2007), while the dynamic ones (i.e., *NC, ANAC, ACD, AACD, PNAC*, cf. Table 6) assess the dynamic behavior and interplay between the components at run-time (Narasimhan and Hendradjaya, 2007).

Source-code metrics have been also organized by **language paradigm**. For instance, Nuñez-Varela et al. (2017) discerned *object-oriented metrics* (e.g., *C&K metrics* Chidamber and Kemerer, 1994), *aspect-oriented metrics* (e.g., *Base-Aspect Coupling (BAC)* Burrows et al., 2010), *feature-oriented metrics* (e.g., *Scattering of a Feature (FSCA)* Olszak and Jørgensen, 2014) 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 capture the interplay of instructions within the program and

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

Internal attribute	Examples of metrics	Ref. in reviews
Abstraction	ANA from QMOOD	Jabangwe et al. (2015)
Cohesion	LCOM from C&K suite; CAM from QMOOD suite; Weiser metrics; Ott and Thuss metrics; Pcoh	Jabangwe et al. (2015), Kaur (2020), Mijač and Stapić (2015), Li et al. (2018), Tiwari and Rathore (2018)
Coupling	CBO from C&K; CF from MOOD suite; DCC from QMOOD suite; NF from Lorenz and Kidd suite; Halstead's metrics; PLC	Jabangwe et al. (2015), Kaur (2020), Mijač and Stapić (2015), Arvanitou et al. (2017), Mehboob et al. (2021)
Complexity	WMC from C&K; NOM_{QMOOD} from QMOOD; NPM, NM from Lorenz and Kidd suite; CC; Halstead's metrics	Jabangwe et al. (2015), Kaur (2020), Tiwari and Rathore (2018)
Encapsulation	CluF, MHF, AHF from MOOD suite; DAM from QMOOD	Jabangwe et al. (2015)
Inheritance	DIT, NOC from C&K; AIF, MIF, RF from MOOD; MFA from QMOOD, NMI from Lorenz and Kidd suite	Jabangwe et al. (2015), Kaur (2020), Mijač and Stapić (2015)
Messaging	CIS from QMOOD suite	Jabangwe et al. (2015)
Modularity	MPC from Li and Henry suite; NF from Lorenz and Kidd suite; LCOM from C&K	Mijač and Stapić (2015)
Polymorphism	PF from MOOD suite	Jabangwe et al. (2015)
Size	LOC; DSC, NOH from QMOOD suite; AMS from Lorenz and Kidd suite; NOM_{QMOOD}, NOP from QMOOD	Jabangwe et al. (2015), Kaur (2020)

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

External attribute	Examples of metrics	Ref. in reviews
Understandability	CC; Halstead's volume; WMC, DIT, CBO, RFC from C&K suite; QMOOD suite; LOC	Mijač and Stapić (2015), Wedyan and Abufakher (2020), Arvanitou et al. (2017)
Cognitive complexity	CFS, Misra et al. metrics	Li et al. (2018)
Readability	Buse and Weimer suite; Dorn suite; Scalabrino et al. suite	Kaur (2020), Tiwari and Rathore (2018), Bexell (2020)
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	Jabangwe et al. (2015), Arvanitou et al. (2017), Piotrowski and Madeyski (2020), Li et al. (2018), Kaur (2020), Sabir et al. (2019)
Change proneness	DIT, NOC, CBO, RFC, LCOM from C&K suite	Arvanitou et al. (2017)
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	Arvanitou et al. (2017), Kaur (2020), Wedyan and Abufakher (2020), Li et al. (2018), Abdellatief et al. (2013)
Effort	CBO, WMC from C&K; LOC; Size1, Size2, MPC, DAC, $NOM_{L\&H}$ from Li and Henry suite	Jabangwe et al. (2015)
Code decay	Code change metrics; LOC	Bandi et al. (2013)
Stability	WMC from C&K suite; LOC	Arvanitou et al. (2017)
Completeness	CC, Halstead's volume, WMC, DIT, CBO, RFC from C&K suite	Mijač and Stapić (2015)
Effectiveness	MIF, AIF from MOOD suite; QMOOD suite	Vanitha and ThirumalaiSelvi (2014), Wedyan and Abufakher (2020)
Flexibility	QMOOD suite	Wedyan and Abufakher (2020)
Functionality	QMOOD suite	Wedyan and Abufakher (2020)
Reliability	CBO, RFC from C&K suite; CC; DCC from QMOOD suite	Kaur (2020)
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)	Wedyan and Abufakher (2020), Arvanitou et al. (2017), Mehboob et al. (2021), Mijač and Stapić (2015), Abdellatief et al. (2013)
Testability	RFC, CBO, LCOM from C&K; LOC	Arvanitou et al. (2017), Jabangwe et al. (2015)

aim at quantifying the complexity of execution paths within the system (e.g., *CC* McCabe, 1976), the latter emphasize the data perspective of the system and measure the flow of information between the program variables (e.g., *MPC* i.e., part of Li and Henry suite Li and Henry, 1993, 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* Shao and Wang, 2003; Misra et al., 2018) and metrics capturing the semantics derived from the natural language vocabulary used in the source-code identifiers and comments (e.g., *Scalabrino et al. suite* Scalabrino et al., 2016).

4.3.3. Metrics' relationship with cognitive load

Looking at the findings of the covered literature reviews, a direct relationship between source-code metrics and cognitive load is not very clear. Instead, metrics were usually associated with external quality attributes such as fault proneness (Jabangwe et al., 2015; Arvanitou et al., 2017; Piotrowski and Madeyski,

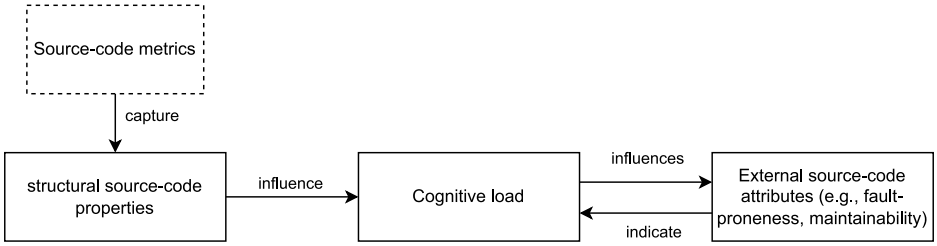


Fig. 4. Theoretical basis linking source-code metrics to external source-code quality attributes through the concept of cognitive load. Note that the terms cognitive complexity (originally adopted in Briand et al. (1998)) and cognitive load are used interchangeably (cf. definitions in Section 4.2).
Source: Adapted from Briand et al. (1998).

2020; Li et al., 2018; Kaur, 2020; Sabir et al., 2019) and maintainability (Arvanitou et al., 2017; Kaur, 2020; Wedyan and Abufakher, 2020; Li et al., 2018; Abdellatief et al., 2013). These associations however lack theoretical foundations as they were typically based on correlations between different metric values and indicators derived from version control code repositories (e.g., change logs, change frequency, reported bugs, code churn Jabangwe et al., 2015). This lack of theoretical foundation was addressed in Isong and Obeten (2013), where (based on an existing framework Briand et al., 1998) the authors used the concept of cognitive load in an attempt to explain how metrics capturing structural source-code properties (cf. Table 8) are linked to different external quality attributes. As shown in Fig. 4, on the one hand, structural source-code properties influence developers' cognitive load which itself can affect several external source-code quality attributes. On the other hand, the quality of the source-code in terms of its different external attributes can be used to estimate developers' cognitive load when engaging with it. These relationships seem plausible considering the cognitive load theory background. As mentioned in Section 2.2, when humans perform difficult tasks, their cognitive load can exceed the capacity of their working memory, which in turn, would hinder their performance and make them more prone to error (Chen et al., 2016; Paas et al., 2003; Veltman and Jansen, 2005). Dealing with poorly structured source-code can raise the difficulty of software development tasks (e.g., code change Müller and Fritz, 2016), which can lead developers to exceed their memory capacity, become less performant and more error-prone. All these characteristics would eventually hinder the quality of the resulting source-code, making it even more mentally demanding to deal with in the future.

A hypothetical relationship between source-code metrics and cognitive load can be constructed when delving into the intrinsic and extraneous components of cognitive load and their link with the essential and accidental complexities of the source-code. As mentioned in Section 2.2, intrinsic load emerges from the essential complexity inherent to the software functionalities encoded in the source-code, while extraneous load emerges from the accidental complexity associated with the representation of the source-code. Many of the metrics identified in Section 4.3.1 can be used to estimate the inherent or accidental complexity of the source-code and thus infer developers' intrinsic and extraneous loads. For instance, the *cognitive metrics* proposed in Shao and Wang (2003) and Misra et al. (2018) suites can be used as indicators for essential complexity, as the use of different control-flow structures (cf. Section 4.3.1) with different cognitive weights (Shao and Wang, 2003) is to a large extent dependent of the inherent complexity of the software functionalities. Conversely, the *textual and readability metrics* in Scalabrino et al. (2016), Buse and Weimer (2009) and Dorn (2012) suites as well as the *anti-pattern metrics* in Taba et al. (2013) suite can be used as indicators for accidental complexity as they capture aspects that are not determined by the software functionalities but rather developers' writing and thus the representation of the source-code (Buse and Weimer, 2009). These accidental complexity aspects can be removed through refactoring (Taba et al., 2013). With regards to the *basic and object-oriented metrics* capturing internal source-code quality attributes (cf. Table 8), they can indicate both the essential and accidental complexities as they can reflect the inherent complexity of the software functionalities and at the same time indicate design and implementation imperfections, hence help identifying anti-patterns in the code (Rasool and Arshad, 2015).

4.4. At what granularity can existing metrics be calculated? (RQ4)

The metrics identified in the existing reviews are calculated at different levels of granularity. Based on the insights reported in Catal and Diri (2009), Abdellatief et al. (2013), Burrows et al. (2009), Mehboob et al. (2021), Colakoglu et al. (2021), 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. Examples of metrics that are collected at this granularity level are *LOC*, and those within the *Halstead's suite* (Halstead, 1977), *Buse and Weimer suite (Buse and Weimer, 2009)*, *Scalabrino et al. suite* (Scalabrino et al., 2016), *Taba et al. suite* (Taba et al., 2013) and *Moser et al. suite* (Moser et al., 2008).

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* (Chidamber and Kemerer, 1994) are the most common class metrics identified in the existing literature reviews. These metrics require an overview of 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 (1993), Lorenz and Kidd (1994), MOOD (Abreu and Carapuça, 1994) and QMOOD (Bansiya and Davis, 2002) suites.

Package level metrics. These metrics require a package structure in the code base. Herein, metrics such as *Package Level Cohesion (Pcoh)* (Gupta and Chhabra, 2012) and *Package Level Coupling (PLC)* (Tripathi and Kushwaha, 2015) 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* Salman, 2006), interface method metrics (e.g., *number of methods and*

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, Ps.: Process.

Tool name	Tool link	Ac./Com.	Mtr/CS	Lang. supported	Refs
CCCC	http://cccc.sourceforge.net	Academic	P. Metrics	C, C++	Valença et al. (2018), Nuñez-Varela et al. (2017), Dias Canedo et al. (2019)
CKJM	https://www.spinellis.gr/sw/ckjm/	Academic	P. Metrics	Java	Ardito et al. (2020), Fregnan et al. (2019), Mehboob et al. (2021), Nuñez-Varela et al. (2017), Tomas et al. (2013), Dias Canedo et al. (2019), Malhotra and Chug (2016)
CMT++/CMTJava	https://www.verifysoft.com/en_cmtx.html	Commercial	P. Metrics	C, C++, C#, Java	Valença et al. (2018), Ardito et al. (2020), Dias Canedo et al. (2019)
DÉCOR	http://www.ptidej.net/research/designsmells/	Academic	Code smells	Java	Alkharabsheh et al. (2019), Rasool and Arshad (2015)
Eclipse Metrics Plugin	http://easyeclipse.org/site- 1.0.2/plugins/metrics.html	Academic	P. Metrics	Java	Valença et al. (2018), Arvanitou et al. (2017), Malhotra and Chug (2016)
JHawk	http://www.virtualmachinery.com/jhawkmetrics.htm	Commercial	P. Metrics	Java	Valença et al. (2018), Ardito et al. (2020), Nuñez-Varela et al. (2017), Dias Canedo et al. (2019)
Stench Blossom	https://multiview.cs.pdx.edu/refactoring/smells/	Academic	Code smells	Java	Fernandes et al. (2016), Rasool and Arshad (2015)
Understand	https://www.scitools.com	Commercial	P. Metrics	C, C++, C#, Java, Python, and 15 other languages	Valença et al. (2018), Fernandes et al. (2016), Ardito et al. (2020), Nuñez-Varela et al. (2017), Rattan and Kaur (2016), Malhotra and Chug (2016)
Xradar	http://xradar.sourceforge.net	Academic	P. Metrics and Code smells	Java	Tomas et al. (2013), Yan et al. (2019)
DePress	https://github.com/ImpressiveCode/ic-depress	Academic	P.&Ps. Metrics	Java	Valença et al. (2018), Dias Canedo et al. (2019)
Essential metrics	https://www.powersoftware.com/em/	Commercial	P. Metrics	C, C++, C#, Java	Valença et al. (2018), Dias Canedo et al. (2019)
UnitMetrics	http://unitmetrics.sourceforge.net	Academic	P. Metrics	Java	Valença et al. (2018), Dias Canedo et al. (2019)
RSM	http://www.msquaredtechnologies.com	Commercial	P. Metrics	C, C++, C#, Java	Valença et al. (2018), Dias Canedo et al. (2019)
JArchitect	http://www.jarchitect.com/	Commercial	P. Metrics	Java	Valença et al. (2018), Dias Canedo et al. (2019)
MASU	https://sourceforge.net/projects/masu/	Academic	P.&Ps. Metrics	Java	Valença et al. (2018), Dias Canedo et al. (2019)
Source Monitor	https://www.derpaul.net/SourceMonitor/	Academic	P. Metrics	C, C++, C#, Java and 4 other languages	Valença et al. (2018), Dias Canedo et al. (2019), Arvanitou et al. (2017)

parameters Rotaru and Dobre, 2005) and property signature metrics (e.g., *number of arguments passed by reference and arguments passed by values* Boxall and Araban, 2004).

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 (Weiser, 1981). As mentioned in Section 4.3.1, examples of slice-based metrics include those in the Weiser (1981) and Ott and Thuss (1993) *suites*.

The implication of these levels of granularity on the alignment between existing source-code metrics and cognitive load is discussed in Section 5.

4.5. What tools and projects can be used to extract source-code metrics and create benchmarking datasets for future experiments? (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 ones (i.e., tools in which neither the source-code nor an executable version is available online), the list reported 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).

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). The tools and datasets reported in this section can be used to enable and support new empirical studies testing the relationship between source-code metrics and cognitive load.

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 road for the future research (cf. Section 5.3).

5.1. Conceptual framework

The analysis of the existing literature reviews provides interesting insights on source-code metrics. These insights can be synthesized into the conceptual framework depicted in Fig. 5. Therein, a metric has a granularity defining the level of source-code at which it can be collected. As presented in Section 4.4, 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 4.3.2 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). Moreover, metrics can be specific to a source-code language paradigm or quality attribute.

Metrics are used estimate the quality of source-code (i.e., internal or external, cf. Section 4.3.2). In turn, source-code quality, can influence cogntive load. As mentioned in Section 4.3.3, Isong and Obeten (2013), provided a reasonable explanation of this relationship in their literature review by describing how structural source-code properties (captured by metrics) can affect cognitive load, which itself can hinder developers' performance and lead to produce low-quality source-code. However, while Isong

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

Project name	Link	Type	Language	Refs
Apache ANT	https://github.com/apache/ant	source-code	Java	Nuñez-Varela et al. (2017), Malhotra (2015), Kaur (2020)
Apache Camel	https://github.com/apache/camel	source-code	Java	Kaur (2020), Malhotra (2015)
Apache Ivy	https://github.com/apache/ant-ivy	source-code	Java	Kaur (2020), Malhotra (2015)
Apache Lucene	https://github.com/apache/lucene	source-code	Java	Nuñez-Varela et al. (2017), Malhotra (2015)
Apache Synapse	https://github.com/apache/synapse	source-code	Java	Kaur (2020), Malhotra (2015)
Apache Tomcat	https://github.com/apache/tomcat	source-code	Java	Kaur (2020), Malhotra (2015)
Argo uml	https://github.com/argouml-tigris-org	source-code	Java	Malhotra and Bansal (2015), Malhotra (2015), Nuñez-Varela et al. (2017)
Eclipse	https://git.eclipse.org/c/	source-code	Java	Malhotra and Bansal (2015), Nuñez-Varela et al. (2017), Wu et al. (2016), Malhotra (2015), Kaur (2020), Li et al. (2018)
FreeCol	https://github.com/FreeCol/freecol	source-code	Java	Malhotra and Bansal (2015), Kaur (2020)
GanttProject	https://www.ganttproject.biz	source-code	Java	Nuñez-Varela et al. (2017), Kaur (2020)
Hibernate	https://github.com/hibernate	source-code	Java	Malhotra and Bansal (2015), Nuñez-Varela et al. (2017), Wu et al. (2016)
Jabref	https://github.com/JabRef/jabref	source-code	Java	Kaur (2020), Nuñez-Varela et al. (2017)
JBoss	https://github.com/wildfly/wildfly	source-code	Java	Kaur (2020), Wu et al. (2016)
jEdit	https://github.com/albfan/jEdit	source-code	Java	Kaur (2020), Nuñez-Varela et al. (2017)
JFreeChart	https://github.com/jfree/jfreechart	source-code	Java	Kaur (2020), Malhotra and Bansal (2015)
JHotDraw	https://github.com/wumpz/jhotdraw	source-code	Java	Kaur (2020), Nuñez-Varela et al. (2017)
Mozilla Porject	https://github.com/mozilla/	source-code	Several languages inc. Java	Malhotra (2015), Wu et al. (2016)
PROMISE repository	http://promise.site.uottawa.ca/SERepository/	pre-computed metrics		Li et al. (2018), Nuñez-Varela et al. (2017)

and Obeten considered only structural source-code properties (e.g., captured using cognitive metrics Shao and Wang, 2003 or object-oriented metrics Chidamber and Kemerer, 1994; Li and Henry, 1993; Lorenz and Kidd, 1994; Abreu and Carapuça, 1994; Bansiya and Davis, 2002), textual features (e.g., captured using readability metrics Scalabrino et al., 2016; Buse and Weimer, 2009; Dorn, 2012) could also be seen as source-code properties with a possible influence on developers' cognitive load (Fakhoury et al., 2018, 2020; Siegmund et al., 2017).

Delving into a more fine-grained level of the conceptual framework, the structural and textual properties can be related to the source-code essential and accidental complexities (Buse and Weimer, 2009), which themselves can respectively affect the intrinsic and extraneous components of cognitive load (cf. Section 4.3.3).

Overall, through its different attributes, the conceptual framework synthesized in this work provides a characterization of source-code metrics with a particular focus on their potential relationship to cognitive load based on the literature findings. This framework 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. Furthermore, the proposed framework aims at setting the grounds for theorizing the relationship between source-code metrics and cognitive load by bridging the gap between source-code essential/accidental complexities and the intrinsic/extraneous components of cognitive load.

5.2. Literature gaps

The findings of this tertiary review hint toward a set of gaps in the literature. Starting with the covered literature reviews, the relationship between source-code metrics and quality attributes such as fault proneness, code smells and maintainability received high attention in the secondary studies. Conversely, the link between source-code metrics and cognitive load remained undercovered.

The lack of literature reviews bridging the gap between source-metrics and cognitive load does not necessarily imply that no studies have looked into this relationship. In fact, a handful of studies from the last couple of years have investigated this topic (e.g., Peitek et al., 2021, 2018; Medeiros et al., 2019, 2021; Couceiro et al., 2019). A possible explanation for the reason why these studies were not covered in the recent literature reviews could be that the topic of "source-code metrics and cognitive load" was never a core subject of the existing literature reviews and thus no interest was given to this research track.

When it comes to the primary studies covered by the identified literature reviews, metrics were associated with different quality attributes. However, there are still no clear theoretical foundations explaining these associations, especially from a cognitive perspective. The framework reported in Isong and Obeten (2013) provides a good starting point for developing these theoretical foundations but still lacks depth when it comes to discerning the different components of cognitive load and linking them with metrics reflecting the essential and accidental complexities of the source-code (Weiser, 1981; Ott and Thuss, 1993).

The analysis of the findings in Section 4.3.3 suggests that some classes of source-code metrics could potentially relate to cognitive load. Yet, the granularity level at which metrics are computed can pose a challenge when evaluating their alignment

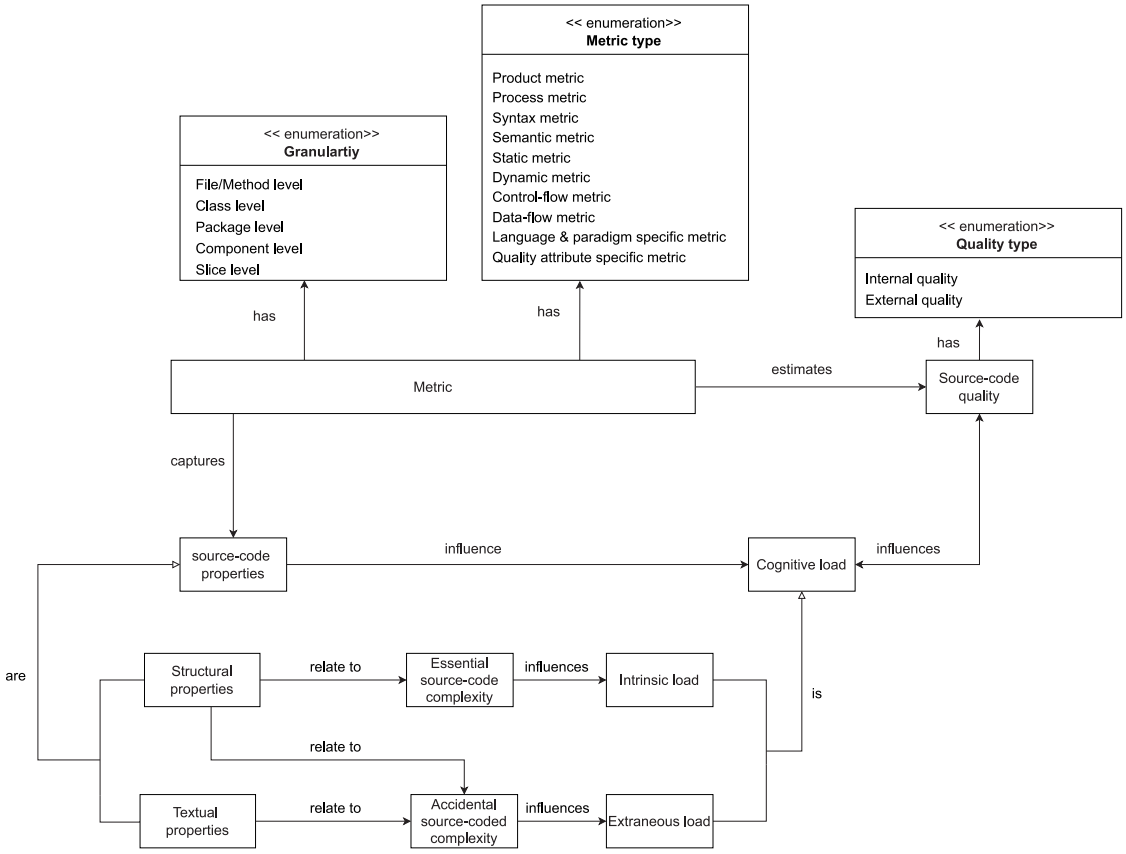


Fig. 5. A conceptual framework synthesizing the findings of the tertiary review.

with cognitive load. As mentioned in Section 3.1 and pointed out by Petrusel and Mendling (2013) (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 mostly on the source-code fragments that are relevant for that task (assuming that they can identify them). Although many of 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. Slice-based metrics can be used to overcome this limitation assuming that the task-relevant parts of code share some common control-flow and data-flow dependencies forming slices of which the properties can be quantified using specific metrics. However, the limited number of existing slice-based metrics (Weiser, 1981; Ott and Thuss, 1993) do not capture many of the structural and textual aspects affecting the source-code essential and accidental complexities.

5.3. Research agenda

This section suggests a research agenda based on the literature gaps identified in Section 5.2. First and foremost, it is important for the upcoming literature reviews to shed more light on the relationship between source-code metrics and cognitive load. To this end, the upcoming reviews need to engage with the recent literature on that topic (e.g., Peitek et al., 2021, 2018; Medeiros et al., 2019, 2021; Couceiro et al., 2019) and explore the wide stream of research on program comprehension and cognitive aspects in software engineering (e.g., Müller and Fritz, 2016; Abbad-Andaloussi et al., 2022).

Secondly, the chain linking source-code metrics, cognitive load and source-code quality needs to be further examined and theorized to explain their interplay. It is also important to investigate the granularity of source-code metrics and how existing metrics can be tailored to capture the quality of the task-relevant code only. Applying slicing techniques (Weiser, 1984) can help to attain this objective. In this context, the slice-based metrics proposed in the literature (Weiser, 1981; Ott and Thuss, 1993) 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 should measure the same properties as those captured by the existing structural and textual metrics (e.g., Shao and Wang, 2003; Chidamber and Kemerer, 1994; Li and Henry, 1993; Lorenz and Kidd, 1994; Abreu and Carapuça, 1994; Bansiya and Davis, 2002; Scalabrino et al., 2016; Buse and Weimer, 2009; Dorn, 2012) with the only constraint to cover just the task-relevant source-code.

Given that the aforementioned objectives are attained, the relationship between source-code metrics and cognitive load needs to be validated empirically. The few attempts that have been made in this regard (e.g., Peitek et al., 2021, 2018; Medeiros et al., 2019, 2021; Couceiro et al., 2019) used only small source-code snippets in their experiments which do not reflect the real-world settings where developers typically engage with large source-code repositories when performing different software development tasks. The datasets and metric extraction tools identified in Section 4.5 can be used in this vein to conduct large-scale experiments but also serve as benchmarks to compare the findings of the upcoming studies.

6. Threats to validity

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

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 3.4) and several search iterations were conducted, i.e., pilot search, main search and snowballing search (cf. Sections Sections 3.2, 3.7 and 3.8). However, as this is a single-authored article, it was not possible to reiterate the search and data extraction procedures by a second researcher nor to discuss every single article before its inclusion or exclusion in the STR. However, as mentioned in Section 3, the criteria based on which the papers were selected have been discussed with the author's colleagues and were refined to ensure a good consistency and coverage of the literature. Moreover, the articles perceived as being borderline by the author were discussed with his colleagues prior to their inclusion in the paper. Nevertheless, it is worthwhile to mention that the search process was completed in August 2022 and therefore all work published after this date was not covered.

Descriptive validity. Descriptive validity denotes the extent to which the findings reported in the study are described objectively and accurately. To mitigate this threat, a well-defined data extraction scheme (cf. Table 3) was applied to all the literature reviews identified during the literature search. Nonetheless, the subjectivity of the qualitative analysis procedure might have influenced the reported findings. To minimize this effect, the author has followed a systematic data analysis protocol, where initial, focused and axial coding (Corbin and Strauss, 2014) were conducted iteratively to extract and synthesize the findings of this study. This procedure has been used in other studies and yielded interesting results (Zimmermann et al., 2022; Debois et al., 2020; Abbad Andaloussi et al., 2021). In addition, the findings of this STR were discussed with the author's colleagues who have provided constructive feedback to improve the quality of the paper.

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 were documented in detail following well-recognized guidelines (Kitchenham, 2007) (cf. Section 3).

7. Conclusion

This paper presents an STR study investigating source-code metrics with a particular focus on their relationship to cognitive load. Following the approach presented in Section 3, the finding of this study yielded 53 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), capturing different internal and external quality attributes of the source-code. These metrics were characterized differently in the literature (e.g., product vs. process, static vs. dynamic, control vs. data-flow, syntax vs. semantics; cf. Section 4.3). They were also collected at different levels of granularity (i.e., file/method, class, package, component, slice; cf. Section 4.4), Moreover, as part of the analysis, the relationship between source-code metrics and cognitive load was investigated. Therein, different classes of metrics were related to the intrinsic and extraneous components of cognitive load. Furthermore, a set of popular tools and projects were identified to extract source-code metrics and create benchmarking datasets for future experiments (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 and highlighting the potential relationship between source-code metrics and the intrinsic and extraneous components of cognitive load (Section 5.1).

Last but not least, a set of gaps identified in the literature have been reported and discussed. Namely, the existing literature reviews have granted little attention to the studies investigating the link between source-code metrics and cognitive load. Moreover, the primary studies covered by these literature reviews did not provide clear theoretical foundations explaining how their metrics relate to the quality attributes they capture, especially from a cognitive perspective. When it comes to the metrics themselves, although some of them could in principle reflect developers' intrinsic and extraneous load, the granularity level at which they are collected could hinder their precision if not tailored to capture the task-relevant code only. The aforementioned limitations were discussed (cf. Section 5.2) and a research agenda paving the road for the future work was proposed (Section 5.3). In brief, the research agenda encourages future literature reviews to search and cover more articles on the link between source-code metrics and cognitive load. It also sheds light on code-slicing techniques which can be adapted to propose a new generation of slice-based metrics that capture the task-relevant code in their calculations. In addition, it raises the need to develop more advanced theories on source-code metrics in relation to cognitive load and validate them in rigorous empirical studies.

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.

CRediT authorship contribution statement

Amine Abbad-Andaloussi: Literature search, Analysis, Conceptualization, Writing of the paper.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

A link to the literature search data is provided in the article.

Acknowledgments

Work supported by the International Postdoctoral Fellowship (IPF), Switzerland 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

Abbad-Andaloussi, A., Sorg, T., Weber, B., 2022. Estimating developers' cognitive load at a fine-grained level using eye-tracking measures. In: Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension. pp. 111–121.

Abbad Andaloussi, A., Zerbato, F., Burattin, A., Slaats, T., Hildebrandt, T.T., Weber, B., 2021. Exploring how users engage with hybrid process artifacts based on declarative process models: a behavioral analysis based on eye-tracking and think-aloud. Softw. Syst. Model. 20 (5), 1437–1464.

Abdellatief, M., Sultan, A.B.M., Ghani, A.A.A., Jabar, M.A., 2013. A mapping study to investigate component-based software system metrics. J. Syst. Softw. 86 (3), 587–603.

Abreu, F.B., Carapuça, R., 1994. Object-oriented software engineering: Measuring and controlling the development process. In: Proceedings of the 4th International Conference on Software Quality. vol. 186.

AbuHassan, A., Alshayeb, M., Ghouti, L., 2021. Software smell detection techniques: A systematic literature review. J. Softw.: Evol. Process 33 (3), e2320.

Alkharabsheh, K., Crespo, Y., Manso, E., Taboada, J.A., 2019. Software Design Smell Detection: a systematic mapping study. Softw. Qual. J. 27 (3), 1069–1148.

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

Anon, 1990. IEEE standard glossary of software engineering terminology. In: IEEE Std 610.12-1990. pp. 1–84. http://dx.doi.org/10.1109/IEEESTD.1990.101064.

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

Ardito, L., Coppola, R., Barbato, L., Verga, D., 2020. A tool-based perspective on software code maintainability metrics: A systematic literature review. Sci. Program. 2020.

Arvanitou, E.M., Ampatzoglou, A., Chatzigeorgiou, A., Galster, M., Avgeriou, P., 2017. A mapping study on design-time quality attributes and metrics. J. Syst. Softw. 127, 52–77.

Baddeley, A.D., Hitch, G., 1974. Working memory. In: Psychology of Learning and Motivation. vol. 8, Elsevier, pp. 47–89.

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

Bansiya, J., Davis, C.G., 2002. A hierarchical model for object-oriented design quality assessment. IEEE Trans. Softw. Eng. 28 (1), 4–17.

Bellini, C.G.P., Pereira, R.D.C.D.F., Becker, J.L., 2008. Measurement in software engineering: From the roadmap to the crossroads. Int. J. Softw. Eng. Knowl. Eng. 18 (01), 37–64.

Bexell, A., 2020. Software source code readability: A mapping study.

Boehm, B.W., Brown, J.R., Lipow, M., 1976. Quantitative evaluation of software quality. In: Proceedings of the 2nd International Conference on Software Engineering. pp. 592–605.

Borchert, T., 2008. Code profiling: Static code analysis.

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

Briand, L.C., Wüst, J., Ikonomovski, S., Lounis, H., 1998. A comprehensive investigation of quality factors in object-oriented designs. An industrial case study.

Brooks, F., Kugler, H., 1987. No Silver Bullet.

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

Burrows, R., Garcia, A., Taïani, F., 2009. Coupling metrics for aspect-oriented programming: A systematic review of maintainability studies. Eval. Nov. Approaches Softw. Eng. 277–290.

Buse, R.P., Weimer, W.R., 2009. Learning a metric for code readability. IEEE Trans. Softw. Eng. 36 (4), 546–558.

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

Catal, C., Diri, B., 2009. A systematic review of software fault prediction studies. Expert Syst. Appl. 36 (4), 7346–7354.

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

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

Chidamber, S.R., Kemerer, C.F., 1994. A metrics suite for object oriented design. IEEE Trans. Softw. Eng. 20 (6), 476–493.

Colakoglu, F.N., Yazici, A., Mishra, A., 2021. Software product quality metrics: A systematic mapping study. IEEE Access.

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

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

Crnkovic, I., Larsson, M.P.H., 2002. Building Reliable Component-Based Software Systems. Artech House.

Debois, S., López, H.A., Slaats, T., Andaloussi, A.A., Hildebrandt, T.T., 2020. Chain of events: modular process models for the law. In: International Conference on Integrated Formal Methods. Springer, pp. 368–386.

Dias Canedo, E., Valença, K., Santos, G.A., 2019. An analysis of measurement and metrics tools: A systematic literature review.

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

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

Dumke, R., Ebert, C., 2007. Software measurement: Establish-extract-evaluate-execute. Chapter 8.3 Measurements for Project Control.

Fakhoury, S., Ma, Y., Arnaoudova, V., Adesope, O., 2018. 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, pp. 286–28610.

Fakhoury, S., Roy, D., Ma, Y., Arnaoudova, V., Adesope, O., 2020. Measuring the impact of lexical and structural inconsistencies on developers' cognitive load during bug localization. Empir. Softw. Eng. 25 (3), 2140–2178.

Fenton, N., Pfleeger, S., 1997. Software metrics. In: A Rigorous and Practical Approach, second ed. PWS Publishing Co, Boston.

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

Fowler, M., 2018. Refactoring: Improving the Design of Existing Code. Addison-Wesley Professional.

Fregnan, E., Baum, T., Palomba, F., Bacchelli, A., 2019. A survey on software coupling relations and tools. Inf. Softw. Technol. 107, 159–178.

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

Gezici, B., Özdemir, N., Yılmaz, N., Coşkun, E., Tarhan, A., Chouseinoglou, O., 2019. Quality and success in open source software: A systematic mapping. In: 2019 45th Euromicro Conference on Software Engineering and Advanced Applications. SEAA, IEEE, pp. 363–370.

Glinz, M., 2014. Software quality: Software product quality, teaching material.

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

Gonçales, L., Farias, K., da Silva, B.C., 2021. Measuring the cognitive load of software developers: An extended Systematic Mapping Study. Inf. Softw. Technol. 106563.

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

Gupta, V., Chhabra, J.K., 2012. Package level cohesion measurement in object-oriented software. J. Braz. Comput. Soc. 18 (3), 251–266.

Hall, T., Beecham, S., Bowes, D., Gray, D., Counsell, S., 2011. A systematic literature review on fault prediction performance in software engineering. IEEE Trans. Softw. Eng. 38 (6), 1276–1304.

Halstead, M.H., 1977. Elements of Software Science (Operating and Programming Systems Series). Elsevier Science Inc..

Hansen, M., Goldstone, R.L., Lumsdaine, A., 2013. What makes code hard to understand? arXiv preprint arXiv:1304.5257.

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

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

Hutton, D., 2009. Clean code: A handbook of agile software craftsmanship. Kybernetes.

Isong, B., Obeten, E., 2013. A systematic review of the empirical validation of object-oriented metrics towards fault-proneness prediction. Int. J. Softw. Eng. Knowl. Eng. 23 (10), 1513–1540.

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

Kaur, A., 2020. A systematic literature review on empirical analysis of the relationship between code smells and software quality attributes. Arch. Comput. Methods Eng. 27 (4), 1267–1296.

Khan, Y.A., Elish, M.O., El-Attar, M., 2012. 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, pp. 258–273.
Kitchenham, B., 2007. Guidelines for performing systematic literature reviews in software engineering. Tech. rep..
Kitchenham, B., 2010. What's up with software metrics?–A preliminary mapping study. J. Syst. Softw. 83 (1), 37–51.
Li, W., Henry, S., 1993. Object-oriented metrics that predict maintainability. J. Syst. Softw. 23 (2), 111–122.
Li, Z., Jing, X.-Y., Zhu, X., 2018. Progress on approaches to software defect prediction. IET Softw. 12 (3), 161–175.
Lloyd, J., 1994. Practical advantages of declarative programming. In: Proc. of Joint Conference on Declarative Programming, GULDPRODE94. Peniscola, Spain, pp. 3–17.
Lorenz, M., Kidd, J., 1994. Object-Oriented Software Metrics: A Practical Guide. Prentice-Hall, Inc..
Malhotra, R., 2015. A systematic review of machine learning techniques for software fault prediction. Appl. Soft Comput. 27, 504–518.
Malhotra, R., Bansal, A., 2015. Predicting change using software metrics: A review. In: 2015 4th International Conference on Reliability, Infocom Technologies and Optimization (ICRITO)(Trends and Future Directions). IEEE, pp. 1–6.
Malhotra, R., Chug, A., 2016. Software maintainability: Systematic literature review and current trends. Int. J. Softw. Eng. Knowl. Eng. 26 (08), 1221–1253.
McCabe, T.J., 1976. A complexity measure. IEEE Trans. Softw. Eng. (4), 308–320.
McCall, J.A., Matsumoto, M.T., 1980. Software quality measurement manual. Tech. rep., vol. 2, General Electric Co Sunnyvale CA.
McKiernan, K.A., Kaufman, J.N., Kucera-Thompson, J., Binder, J.R., 2003. A parametric manipulation of factors affecting task-induced deactivation in functional neuroimaging. J. Cogn. Neurosci. 15 (3), 394–408.
Medeiros, J., Couceiro, R., Castelhano, J., Branco, M.C., Duarte, G., Duarte, C., Durães, J., Madeira, H., Carvalho, P., Teixeira, C., 2019. Software code complexity assessment using EEG features. In: 2019 41st Annual International Conference of the IEEE Engineering in Medicine and Biology Society. EMBC, IEEE, pp. 1413–1416.
Medeiros, J., Couceiro, R., Duarte, G., Durães, J., Castelhano, J., Duarte, C., Castelo-Branco, M., Madeira, H., de Carvalho, P., Teixeira, C., 2021. Can EEG be adopted as a neuroscience reference for assessing software programmers' cognitive load? Sensors 21 (7), 2338.
Mehboob, B., Chong, C.Y., Lee, S.P., Lim, J.M.Y., 2021. Reusability affecting factors and software metrics for reusability: A systematic literature review. Softw. - Pract. Exp. 51 (6), 1416–1458.
Mendling, J., 2007. Detection and prediction of errors in EPC business process models (Ph.D. thesis). Wirtschaftsuniversität Wien Vienna.
Miara, R.J., Musselman, J.A., Navarro, J.A., Shneiderman, B., 1983. Program indentation and comprehensibility. Commun. ACM 26 (11), 861–867.
Mijač, M., Stapić, Z., 2015. Reusability metrics of software components: survey. In: Proceedings of the 26th Central European Conference on Information and Intelligent Systems. pp. 221–231.
Miller, G.A., 1956. The magical number seven, plus or minus two: Some limits on our capacity for processing information. Psychol. Rev. 63 (2), 81.
Misra, S., Adewumi, A., Fernandez-Sanz, L., Damasevicius, R., 2018. A suite of object oriented cognitive complexity metrics. IEEE Access 6, 8782–8796.
Montagud, S., Abrahão, S., Insfran, E., 2012. A systematic review of quality attributes and measures for software product lines. Softw. Qual. J. 20 (3), 425–486.
Moser, R., Pedrycz, W., Succi, G., 2008. 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. pp. 181–190.
Müller, S.C., Fritz, T., 2016. Using (bio) metrics to predict code quality online. In: 2016 IEEE/ACM 38th International Conference on Software Engineering. ICSE, IEEE, pp. 452–463.
Murillo-Morera, J., Quesada-López, C., Jenkins, M., 2015. Software fault prediction: A systematic mapping study. In: CIbSE. p. 446.
Narasimhan, V.L., Hendradjaya, B., 2007. Some theoretical considerations for a suite of metrics for the integration of software components. Inform. Sci. 177 (3), 844–864.
Nguyen, V., Deeds-Rubin, S., Tan, T., Boehm, B., 2007. A SLOC counting standard. In: Cocomo Ii Forum. vol. 2007, Citeseer, pp. 1–16.
Nuñez-Varela, A.S., Pérez-Gonzalez, H.G., Martínez-Perez, F.E., Soubervielle-Montalvo, C., 2017. Source code metrics: A systematic mapping study. J. Syst. Softw. 128, 164–197.
Nurdiani, I., Börstler, J., Fricker, S.A., 2016. The impacts of agile and lean practices on project constraints: A tertiary study. J. Syst. Softw. 119, 162–183.
Olszak, A., Jørgensen, B.N., 2014. Modularization compass navigating the white waters of feature-oriented modularity. In: 2014 9th International Conference on Software Engineering and Applications (ICSOFT-EA). IEEE, pp. 48–59.
Ott, L.M., Thuss, J.J., 1993. Slice based metrics for estimating cohesion. In: [1993] Proceedings First International Software Metrics Symposium. IEEE, pp. 71–81.
Özakıncı, R., Tarhan, A., 2018. Early software defect prediction: A systematic map and review. J. Syst. Softw. 144, 216–239.
Paas, F., Tuovinen, J.E., Tabbers, H., Van Gerven, P.W., 2003. Cognitive load measurement as a means to advance cognitive load theory. Educ. Psychol. 38 (1), 63–71.
Park, R.E., Goethert, W.B., Florac, W.A., 1996. Goal-driven software measurement. A guidebook. Tech. rep., Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Inst.
Peitek, N., Apel, S., Parnin, C., Brechmann, A., Siegmund, J., 2021. Program comprehension and code complexity metrics: An fMRI study. In: 2021 IEEE/ACM 43rd International Conference on Software Engineering. ICSE, IEEE, pp. 524–536.
Peitek, N., Siegmund, J., Apel, S., Kästner, C., Parnin, C., Bethmann, A., Leich, T., Saake, G., Brechmann, A., 2018. A look into programmers' heads. IEEE Trans. Softw. Eng. 46 (4), 442–462.
Petrusel, R., Mendling, J., 2013. Eye-tracking the factors of process model comprehension tasks. In: International Conference on Advanced Information Systems Engineering. Springer, pp. 224–239.
Piotrowski, P., Madeyski, L., 2020. Software defect prediction using bad code smells: A systematic literature review. Data-Centric Bus. Appl. 77–99.
Radjenović, D., Heričko, M., Torkar, R., Živkovič, A., 2013. Software fault prediction metrics: A systematic literature review. Inf. Softw. Technol. 55 (8), 1397–1418.
Rasool, G., Arshad, Z., 2015. A review of code smell mining techniques. J. Softw.: Evol. Process 27 (11), 867–895.
Rattan, D., Kaur, J., 2016. Systematic mapping study of metrics based clone detection techniques. In: Proceedings of the International Conference on Advances in Information Communication Technology & Computing. pp. 1–7.
dos Reis, J.P., e Abreu, F.B., de Figueiredo Carneiro, G., Anslow, C., 2021. Code smells detection and visualization: A systematic literature review. Arch. Comput. Methods Eng. 1–48.
Riaz, M., Mendes, E., Tempero, E., 2009. A systematic review of software maintainability prediction and metrics. In: 2009 3rd International Symposium on Empirical Software Engineering and Measurement. IEEE, pp. 367–377.
Riedl, R., Fischer, T., Léger, P.-M., Davis, F.D., 2020. A decade of neurois research: progress, challenges, and future directions. ACM SIGMIS Database: The DATABASE Adv. Inf. Syst. 51 (3), 13–54.
Riedl, R., Léger, P.-M., 2016. Fundamentals of neurois. Stud. Neurosci. Psychol. Behav. Econ. 127.
Rotaru, O.P., Dobre, M., 2005. Reusability metrics for software components. In: The 3rd ACS/IEEE International Conference OnComputer Systems and Applications, 2005.. IEEE, p. 24.
Sabir, F., Palma, F., Rasool, G., Guéhéneuc, Y.-G., Moha, N., 2019. A systematic literature review on the detection of smells and their evolution in object-oriented and service-oriented systems. Softw. - Pract. Exp. 49 (1), 3–39.
Sagar, K., Saha, A., 2017. A systematic review of software usability studies. Int. J. Inf. Technol. 1–24.
Salman, N., 2006. Complexity metrics as predictors of maintainability and integrability of software components. Cankaya Univ. J. Arts Sci. 1 (5), 39–50.
Santos, M., Afonso, P., Bermejo, P.H., Costa, H., 2016. 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, pp. 1–11.
Saraiva, J., Barreiros, E., Almeida, A., Lima, F., Alencar, A., Lima, G., Soares, S., Castor, F., 2012. Aspect-oriented software maintenance metrics: A systematic mapping study. In: 16th International Conference on Evaluation & Assessment in Software Engineering (EASE 2012). IET, pp. 253–262.
Scalabrino, S., Linares-Vasquez, M., Poshyvanyk, D., Oliveto, R., 2016. Improving code readability models with textual features. In: 2016 IEEE 24th International Conference on Program Comprehension. ICPC, IEEE, pp. 1–10.
Schryen, G., 2010. Preserving knowledge on IS business value. Bus. Inf. Syst. Eng. 2 (4), 233–244.
Seref, B., Tanriover, O., 2016. Software code maintainability: a literature review. Int. J. Softw. Eng. Appl..
Shao, J., Wang, Y., 2003. A new measure of software complexity based on cognitive weights. Can. J. Electr. Comput. Eng. 28 (2), 69–74.
Siegmund, J., Peitek, N., Parnin, C., Apel, S., Hofmeister, J., Kästner, C., Begel, A., Bethmann, A., Brechmann, A., 2017. Measuring neural efficiency of program comprehension. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering. pp. 140–150.
Sima, V., Gheorghe, I.G., Subić, J., Nancu, D., 2020. Influences of the industry 4.0 revolution on the human capital development and consumer behavior: A systematic review. Sustainability 12 (10), 4035.
Snyder, H., 2019. Literature review as a research methodology: An overview and guidelines. J. Bus. Res. 104, 333–339.
Sternberg, R.J., Sternberg, K., 2016. Cognitive Psychology. Nelson Education.
Sweller, J., 2011. Cognitive load theory. In: Psychology of Learning and Motivation. vol. 55, Elsevier, pp. 37–76.

Taba, S.E.S., Khomh, F., Zou, Y., Hassan, A.E., Nagappan, M., 2013. Predicting bugs using antipatterns. In: 2013 IEEE International Conference on Software Maintenance. IEEE, pp. 270–279.
Tahir, A., MacDonell, S.G., 2012. A systematic mapping study on dynamic metrics and software quality. In: 2012 28th IEEE International Conference on Software Maintenance. ICSM, IEEE, pp. 326–335.
Tiwari, S., Rathore, S.S., 2018. Coupling and cohesion metrics for object-oriented software: a systematic mapping study. In: Proceedings of the 11th Innovations in Software Engineering Conference. pp. 1–11.
Tomas, P., Escalona, M.J., Mejias, M., 2013. Open source tools for measuring the Internal Quality of Java software products. A survey. Comput. Stand. Interfaces 36 (1), 244–255.
Torgerson, W.S., 1958. Theory and methods of scaling.
Tripathi, A., Kushwaha, D., 2015. A metric for package level coupling. CSI Trans. ICT 2 (4), 217–233.
Valença, K., Canedo, E.D., Figueiredo, R.M.D.C., 2018. 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, pp. 1852–1859.
Vanitha, N., ThirumalaiSelvi, R., 2014. A report on the analysis of metrics and measures on software quality factors–a literature study.
Veltman, J., Jansen, C., 2005. The role of operator state assessment in adaptive automation. TNO Defence Security and Safety Soesterberg (Netherlands).
Wang, D., Ueda, Y., Kula, R.G., Ishio, T., Matsumoto, K., 2021. Can we benchmark Code Review studies? A systematic mapping study of methodology, dataset, and metric. J. Syst. Softw. 111009.
Weber, B., Fischer, T., Riedl, R., 2021. Brain and autonomic nervous system activity measurement in software engineering: A systematic literature review. J. Syst. Softw. 110946.
Webster, J., Watson, R.T., 2002. Analyzing the past to prepare for the future: Writing a literature review. MIS Q. xiii–xxiii.
Wedyan, F., Abufakher, S., 2020. Impact of design patterns on software quality: a systematic literature review. IET Softw. 14 (1), 1–17.
Weiser, M.D., 1981. Program slicing. In: Proceedings of the 5th International Conference on Software Engineering. pp. 439–449.
Weiser, M., 1984. Program slicing. IEEE Trans. Softw. Eng. (4), 352–357.
Winograd, T., 1975. Frame representations and the declarative/procedural controversy. In: Representation and Understanding. Elsevier, pp. 185–210.
Wu, H., Shi, L., Chen, C., Wang, Q., Boehm, B., 2016. Maintenance effort estimation for open source software: A systematic literature review. In: 2016 IEEE International Conference on Software Maintenance and Evolution. ICSME, IEEE, pp. 32–43.
Yan, M., Xia, X., Zhang, X., Xu, L., Yang, D., Li, S., 2019. Software quality assessment model: A systematic mapping study. Sci. China Inf. Sci. 62 (9), 1–18.
Zaidi, M.A., Colomo-Palacios, R., 2019. Code smells enabled by artificial intelligence: a systematic mapping. In: International Conference on Computational Science and Its Applications. Springer, pp. 418–427.
Zheng, R.Z., 2017. Cognitive Load Measurement and Application: A Theoretical Framework for Meaningful Research and Practice. Routledge.
Zimmermann, L., Zerbato, F., Weber, B., 2022. Process mining challenges perceived by analysts: an interview study. In: International Conference on Business Process Modeling, Development and Support, International Conference on Evaluation and Modeling Methods for Systems Analysis and Development. Springer, pp. 3–17.

Amine Abbad-Andaloussi is a postdoctoral researcher at St. Gallen University in Switzerland. He received his Ph.D. degree in Computer Science from the Technical University of Denmark in 2021. His research is about the understandability of software artifacts (e.g., process models, source-code). With an academic background in computer science and a strong research interest in cognitive psychology, Amine investigates the way users engage with software artifacts in order to improve their understandability and enhance the existing software development and modeling tools. Amine has Published in several venues including Information Systems, IEEE Transactions on Learning Technologies and Software and System Modeling.