Evaluating Code Readability and Legibility: An Examination of Human-centric Studies

Delano Oliveira*, Reydne Bruno*, Fernanda Madeiral†, Fernando Castor*
*Federal University of Pernambuco, Recife, Brazil, {dho, rbs8, fjclf}@cin.ufpe.br
†KTH Royal Institute of Technology, Stockholm, Sweden, fer.madeiral@gmail.com

Abstract—**Reading code is an essential activity in software maintenance and evolution. Several studies with human subjects have investigated how different factors, such as the employed programming constructs and naming conventions, can impact code readability, i.e., what makes a program easier or harder to read and apprehend by developers, and code legibility, i.e., what influences the ease of identifying elements of a program. These studies evaluate readability and legibility by means of different comprehension tasks and response variables. In this paper, we examine these tasks and variables in studies that compare programming constructs, coding idioms, naming conventions, and formatting guidelines, e.g., recursive vs. iterative code. To that end, we have conducted a systematic literature review, where we found 54 relevant papers. Most of these studies evaluate code readability and legibility by measuring the correctness of the subjects' results (83.3%) or simply asking their opinions (55.6%). Some studies (16.7%) rely exclusively on the latter variable. There are still few studies that monitor subjects' physical signs, such as brain activation regions (5%). Moreover, our study shows that some variables are multi-faceted. For instance, correctness can be measured as the ability to predict the output of a program, answer questions about its behavior, or recall parts of it. These results make it clear that different evaluation approaches require different competencies from subjects, e.g., tracing the program vs. summarizing its goal vs. memorizing its text. To assist researchers in the design of new studies and improve our comprehension of existing ones, we model program comprehension as a learning activity by adapting a preexisting learning taxonomy. This adaptation indicates that some competencies, e.g., tracing, are often exercised in these evaluations whereas others, e.g., relating similar code snippets, are rarely targeted.**

Index Terms—**Code readability, code legibility, code understandability, code understanding, program comprehension**

I. INTRODUCTION

Understanding a program usually requires reading code. A program might be easier or harder to be read depending on its *readability*, i.e., what makes a program easier or harder to read and apprehend by developers, and *legibility*, i.e., what influences the ease of identifying elements of a program. Different factors can influence code readability and legibility, such as which constructs are employed [1], [2], how the code is formatted with whitespaces [3]–[5], and identifier naming conventions [6]–[13]. Researchers have conducted empirical studies to investigate several of those factors, where different but functionally equivalent ways of writing code are compared, e.g., recursive vs. iterative code [14] and long vs. abbreviated identifiers [10], [15]. These studies involve asking subjects to perform one or more tasks related to source code and assessing their understanding or the effort involved in the tasks.

There are systematic literature reviews about program comprehension studies [16], [17]. To the best of our knowledge, however, no previous work has investigated how code readability and legibility are evaluated in studies comparing different ways of writing code, in particular, what kinds of tasks these studies conduct and what response variables they employ. Analyzing these tasks and variables can help researchers identify limitations in the evaluations of previous studies, gauge their comprehensiveness, and improve our understanding of what said tasks and variables evaluate.

In this paper, we investigate how code readability and legibility are evaluated in studies that aim to identify approaches that are more readable or legible than others. Our goal is to study what types of tasks are performed by human subjects in those studies, what cognitive skills are required from them, and what response variables these studies employ. To achieve this goal, we carried out a systematic literature review, which started with 2,843 documents, where 54 papers are relevant for our research—these papers are the primary studies of our study. Our results show that 40 primary studies involve asking subjects to provide information about a program, for example, to predict its output or the values of its variables, or to provide a high level description of its functioning. In addition, 30 studies involve asking subjects to provide personal opinions, and 15 studies involve asking subjects to act on code. Most studies employ combinations of these types of tasks.

When considering response variables, the most common approach is to verify if the subjects are able to correctly provide information about the code or act on it. Correctness is the response variable of choice in 45 studies. Moreover, 27 studies measure the time required to perform tasks as an imprecise proxy to the effort required by these tasks. Besides, mirroring the aforementioned tasks where subjects are required to provide their personal opinions, 30 studies employ opinion as a response variable. Additionally, 9 of the 54 studies use opinion as the sole response variable. Furthermore, studies evaluating legibility tend to employ opinion as a response variable proportionally more often than studies focusing on readability. Also, there are still relatively few studies that monitor subjects' physical signs, such as brain activation regions (5%) [18]–[20]. The analysis of the response variables reveals that they are multi-faceted. For example, correctness can be measured as the ability to predict the output of a program, answer questions about its general behavior, precisely recall specific parts, identify the cause of a bug, among other things.

To assist researchers in the design of new studies and improve our comprehension of existing ones, we adapt an existing learning taxonomy to the context of program comprehension. This adapted taxonomy indicates that some competencies, e.g., tracing, are often exercised in these evaluations whereas others, e.g., relating similar code snippets, are rarely targeted. The taxonomy also indicates that 37% of the primary studies have a narrow focus, requiring a single cognitive skill from the subjects, even though program comprehension is a complex activity. Additionally, the taxonomy highlights the tendency of the studies evaluating readability and legibility to focus on comprehension tasks that do not contextualize the information that is apprehended. In software development practice, program comprehension is often associated with other tasks such as reengineering an existing program or extending it. Existing studies rarely go this far.

II. CODE READABILITY AND LEGIBILITY

In software engineering, the terms readability, legibility, understandability, and comprehensibility have overlapping meanings. For example, Buse and Weimer [21] define "***readability as a human judgment of how easy a text is to understand***". In a similar vein, Almeida et al. [22] affirm that "*legibility is fundamental to code maintenance; if source code is written in a complex way, understanding it will require much more effort*". In addition, Lin and Wu [23] state that ""*Software **understandability**" determines whether a system can be understood by other individuals easily, or whether artifacts of one system can be easily understood by other individuals*". Xia et al. [24] treat comprehension and understanding as synonyms, expressing that "*Program **comprehension** (aka., program understanding, or source code comprehension) is a process where developers actively acquire knowledge about a software system by exploring and searching software artifacts, and reading relevant source code and/or documentation*".

In linguistics the concept of text comprehension is similar to program comprehension in software engineering. Gough and Tunmer [25] state that "*comprehension (not reading comprehension, but rather linguistic comprehension) is the process by which given lexical (i.e., word) information, sentences and discourses are interpreted*". However, Hoover and Gough [26] further elaborate on that definition and claim that "*decoding and linguistic comprehension are separate components of reading skill*". This claim highlights the existence of two separate processes during text comprehension: (i) decoding the words/symbols and (ii) interpreting them and sentences formed by them. DuBay [27] separates these two processes and defines them as **legibility**, which concerns typeface, layout, and other aspects related to the identification of elements in text, and **readability**, that is, what makes some texts easier to read than others. In a similar vein, for Tekif [28], legibility studies are mainly concerned with typographic and layout factors while readability studies concentrate on the linguistic factors.

These two perspectives also apply to programs. We can find both the visual characteristics and linguistic factors in source code, although with inconsistent terminology. For example,

Daka et al. [29] affirm that "*the visual appearance of code in general is referred to as its readability*". The authors clearly refer to legibility (in the design/linguistics sense) but employ the term "readability" possibly because it is more often used in the software engineering literature.

Based on the differences between the terms "readability" and "legibility" that are well-established in other areas such as linguistics [27], design [30], human-computer interaction [31], and education [28], we believe that the two terms should have clear, distinct, albeit related, meanings also in the area of software engineering. On the one hand, the structural and semantic characteristics of the source code of a program that affect the ability of developers to understand it while reading the code, e.g., programming constructs, coding idioms, and meaningful identifiers, impact its **readability**. On the other hand, the visual characteristics of the source code of a program, which affect the ability of developers to identify the elements of the code while reading it, such as line breaks, spacing, alignment, indentation, blank lines, identifier capitalization, impact its **legibility**. Hereafter, we employ these two terms according to these informal definitions.

III. METHODOLOGY

Our goal is to investigate how human-centric studies evaluate whether a certain approach is more readable or legible than another one. More specifically, we investigate what tasks are performed by human subjects and how their performance is evaluated in empirical studies aiming to evaluate readability and legibility. We focus on studies that directly compare two or more different approaches and have a focus on low-level source code elements, without accounting for tools, documentation, or higher level issues (details are further presented in this section). We address two research questions in this paper:

RQ1 What are the tasks performed by human subjects in empirical studies?

RQ2 What are the response variables of these studies?

To answer to our research questions, we conducted a systematic literature review, which was designed following the guidelines proposed by Kitchenham et al. [32]. Figure 1 presents the roadmap of our review including all steps we followed. First, we performed the selection of studies. We started with a manual search for papers to further be used as seed papers, so that a search string could be defined, and automatic search could be performed on search engines (Section III-A). We retrieved 2,843 documents with the automatic search, which passed through 1) a triage for study exclusion (Section III-B), 2) an initial study selection where inclusion criteria should hold true for each study (Section III-C), and 3) a final study selection where we evaluated the quality of the studies based on a number of criteria (Section III-D). Then, the selected 54 papers were analyzed (Section III-E). We extracted data from the papers and synthesized it to answer our research questions. We detail these steps in the following sections. It is worth mentioning that we did not leverage systematic review tools, like Parsifal [33], because we were not aware of them at the time.

A. Search Strategy

Our search strategy is composed of three parts: a manual search to gather seed studies, the definition of a generic search string, and the automatic search in search engines. First, we performed the manual search for seed studies aiming to find terms for the definition of a generic search string. For that, we chose the following top-tier software engineering conferences: ICSE, FSE, MSR, ASE, ISSTA, OOPSLA, ICSME, ICPC, and SANER. Then, the first two authors of this paper looked at the papers published in these conferences in the last three years and a half (from 2016 to June 2019), and selected the ones that would help us answer our research questions. We also included one paper from TSE, which we already knew is relevant to our research. This process resulted in 13 seed papers.

The title and keywords of the seed papers were analyzed, and then we extracted the general terms related to our research questions. We chose general terms as a conservative way to gather as many papers as possible that can fit within the scope of our study; otherwise, we would delimit our set of studies based on specific topics. We used the resulting terms to build the following search string:

$$Title(ANY(terms))\ OR\ Keywords(ANY(terms)),$$
where $terms = \{$ "code comprehension", "code understandability",
"code understanding", "code readability",
"program comprehension", "program understandability",
"program understanding", "program readability",
"programmer experience" $\}$

We did not include terms with "legibility" in the search string. Most of the papers with this in the title, abstract, or keywords are related to linguistics or computational linguistics. Therefore, they use this term with a different meaning than what would be expected in a software engineering paper. Using it in our search string would drastically increase the number of false positives. Also, we did not include terms with "software", e.g., "software readability", because software is broader than source code and program.

Finally, we performed an automatic search for studies using our generic search string adapted for three search engines: ACM Digital Library [34], IEEE Explore [35], and Scopus [36]. We retrieved 1,926, 729, and 1,909 documents, respectively, in September 2, 2019. Since a given document might be retrieved from more than one engine, we unified the output of the engines to eliminate duplicates, which resulted in 2,843 unique documents. The 13 seed papers were returned by the automatic search.

B. Triage (Study Exclusion)

The 2,843 documents retrieved with our automatic search passed through a triage process so that we could discard clearly irrelevant documents. We first defined five exclusion criteria:

- EC1: The study is outside the scope of this study. It is not primarily related to source code comprehension, readability, or legibility, does not involve any comparison of different ways to write programs, neither direct nor indirect, or is clearly irrelevant to our research questions. For instance,
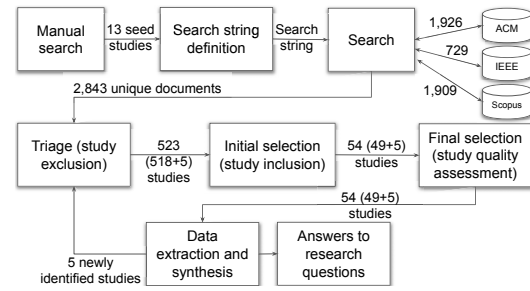

Fig. 1: Systematic literature review roadmap.

we exclude studies focusing on high-level design, documentation, and dependencies (higher-level issues).
- EC2: The study is not a full paper (e.g. MSc dissertations, PhD theses, course completion monographs, short papers) or is not written in English: not considering these types of documents in systematic reviews is a common practice [32]. As a rule of thumb, we consider that full papers must be at least 5 pages long.
- EC3: The study is about readability metrics without an experimental evaluation.
- EC4: The study is about program comprehension aids, such as visualizations or other forms of analysis or sensory aids (e.g., graphs, trace-based execution, code summarization, specification mining, reverse engineering).
- EC5: The study focuses on accessibility, e.g., targets individuals with visual impairments or neurodiverse developers.

Each of the 2,843 documents was analyzed by an author of this paper, who checked the title and abstract of the document, and in some cases the methodology, against the exclusion criteria. The documents that do not meet any of the exclusion criteria were directly accepted to enter the next step (described in the next section). The documents that meet at least one exclusion criterion passed through a second round in the triage process, where each document was analyzed by a different author. In the end of the two rounds, we discarded all documents that were annotated with at least one exclusion criterion in both rounds. We followed this two-round process to mitigate the threat of discarding potentially relevant studies in the triage. Because only rejected papers in the first round were analyzed in the second round, i.e., raters in the second round knew that the documents had been marked for exclusion, no inter-rater agreement analysis was conducted. We ended up with 523 documents.

C. Initial Selection (Study Inclusion)

After discarding clearly irrelevant documents in the triage step, we applied the following inclusion criteria in the 523 papers to build our initial set of papers:

- IC1 (Scope): The study must be primarily related to the topics of code comprehension, readability, legibility, or hard-to-understand code.
- IC2 (Methodology): The study must be or contain at least one empirical study, such as controlled experiment, quasi-experiment, case study, or survey *involving human subjects*.

- IC3 (Comparison): The study must compare alternative programming constructs, coding idioms, or coding styles *in terms of code readability or legibility.*
- IC4 (Granularity): The study must target fine-grained program elements and low-level/limited-scope programming activities. Not design or comments, but implementation.

The application of the inclusion criteria to a paper often requires reading not only the title and abstract as in the triage step, but also sections of introduction, methodology, and conclusion. If a given paper violates at least one inclusion criterion, the paper is annotated with "not acceptable". When there are doubts about the inclusion of a paper, the paper is annotated with "maybe" for further discussion. We also performed this step in two rounds, but differently from the triage, all papers in this step were independently analyzed by two different authors. We calculated the Kappa coefficient [37] for assessing the agreement between the two analyzes. We found $k = 0.323$, which is considered a fair agreement strength [32]. In the end of this step, the papers annotated with "acceptable" in both rounds were directly selected, and papers annotated with "not acceptable" in both rounds were rejected. All the other cases were discussed by the four authors in live sessions to reach consensus. We ended up with 54 papers.

D. Study Quality Assessment

After the search, exclusion, and inclusion of papers, the remaining 54 papers passed through a final selection step, aiming to assess their quality. In this step, we aim to identify low-quality papers for removal. To do so, we elaborated nine questions that were answered for each paper. We adapted these questions from the work of Keele [38]. There were three questions about study design, e.g., "are the aims clearly stated?"; four questions about analysis, e.g., "are the data collection methods adequately described?"; and two questions about rigor, e.g., "do the researchers explain the threats to the study validity?". There are three possible answers for each question: yes (1), partially (0.5), and no (0). The sum of the answers for a given paper is its score. The maximum is, therefore, 9. If a paper scores 0.5 or better in all questions, its overall score is 4.5 or more. Thus, we defined that a paper should score at least 4.5 to be kept in our list of papers.

Each paper was assessed by one of the authors. At the beginning of this step, each author selected one paper, performed the quality assessment, and justified to the other authors the given score for each question in a live discussion. This procedure allows us to align our understanding of the questions and avoid misleading assessments. The scores of the studies were: $min = 5$, $median = 8$, $max = 9$. Since the minimum score for keeping a paper was 4.5 and no paper scored less than 5, no studies were removed because of bad quality.

Inclusion of additional studies. Five relevant studies were not captured by our automatic search because our search string did not cover them, and they were unknown for us when we built the set of seed papers. We found some of these studies when we were reading already included studies in the final selection step—those studies were cited as "related works" to the ones included in our list. The five studies were discussed by all the authors in a live session and, after reaching agreement about their pertinence, they were subjected to exclusion and inclusion criteria and quality evaluation.

Deprecated studies. We identified some papers that we refer to as *deprecated*. A paper is deprecated if it was extended by another paper that we selected. For instance, the work of Buse and Weimer published in 2008 [39] was extended in a subsequent paper [21]. In this case, we consider the former to be deprecated and only take the latter into account.

E. Data Analysis

In this step of the study, we analyzed a total of 54 papers. Initially, we read all the papers in full and extracted from each one the data necessary to answer our research questions. This activity was carried out through a questionnaire that had to be filled in for each paper. For this extraction, the papers were divided equally among the researchers, and periodic meetings were held to discuss the extracted data. At this stage we seek to extract information about the characteristics of the studies, for example, whether they pertain to readability or legibility, the evaluation method, the tasks the subjects were required to perform, and information about the results.

After extracting the data, we analyzed it so as to address the two research questions. For the first research question, we collected all the tasks extracted from the 54 studies. The first two authors examined these tasks together, identifying commonalities and grouping them together. All these tasks were then subject to discussion among all the authors. After we reached a consolidated list of tasks performed by the subjects of these studies, we organized them in three large groups, considering what is required from the subjects: (i) to provide information about the code; (ii) to act on the code; and (iii) to provide a personal opinion about the code.

For the second research question, we adopted a similar procedure. Initially, we collected the extracted response variables. Since they were very diverse, we created groups that include multiple response variables with similar characteristics. For example, response variables related to "correctness" include the predicted output of a program, a recalled part of it, or a general description of its function. At the end, we elicited five categories of response variables in the studies. The initial analysis of the data was conducted by the first two authors and later the results were refined with the collaboration of all the authors.

Data availability. Raw data, such as the list of the 2,843 documents returned by our automatic search, and additional detail about our methodology, such as the questionnaires we used for study quality assessment and data extraction, are available at https://github.com/reydne/code-comprehension-review.

IV. RESULTS

In this section we attempt to answer our two research questions, based on the obtained data.

A. Tasks Performed by Human Subjects (RQ1)

The essential code comprehension task is code reading. By construction, all selected studies have at least one reading task where the subject is required to read a code snippet, a set of snippets, or even large, complete programs. Since all the studies compare two or more approaches to write code, the subjects often have multiple code reading tasks. In addition, the subjects are also expected to comprehend the code. However, there are different ways to measure subject comprehension performance. Subjects are asked to provide information about the code, act on the code, or provide personal opinion. In most studies, more than one kind of task was employed. Table I summarizes the identified tasks.

A large portion of the primary studies, 40 out of 54, require subjects to **provide information about the code**. For example, Benander et al. [14] asked the subjects to explain using free-form text what a code snippet does, right after having read it. Blinman et al. [11] asked the subjects to choose the best description for a code snippet among multiple options. Explaining what the code does is not an objective method to evaluate comprehension because someone has to judge the answer. We found 18 studies that employed this task.

The subjects of 27 studies were asked to answer questions about characteristics of the code. Some studies asked the subjects to predict the behavior of a source code just by looking at it. For example, the studies of Gopstein et al. [1] and Ajami and Feitelson [2] presented subjects with multiple short code snippets and asked them to guess the outputs of these snippets. Dolado et al. [40] asked the subjects to answer a set of questions about expression results, final values of variables, and how many times loops were executed. Other studies ask the subjects questions about higher-level issues related to the source code. For example, the study of Scalabrino et al. [41] asked the subjects if they recognize an element of a specific domain or about the purpose of using an external component in the snippet (e.g., JDBC APIs). Similarly, Binkley et al. [9] inquired the subjects about the kind of application or the industry where a specific line of code might be found. In addition, some studies ask subjects to localize code elements of interest. For example, Binkley et al. [15] asked subjects to find identifiers in a snippet, marking each code line that has that identifier. Ceccato et al. [12] asked the subjects to pinpoint the part of the code implementing a specific functionality.

Some studies make the assumption that code that is easy to understand is also easy to memorize. Therefore, they attempt to measure how much subjects remember the code. For example, Love [42] asked participants to memorize a program for three minutes and rewrite the program as accurately as possible in the next four minutes. Lawrie et al. [7] first presented a code snippet to the subjects. Then, in a second step, they listed six possible identifiers and the subjects had to select the ones that they recalled appearing in the code. Overall, seven studies asked the subjects to remember the code.

On the other hand, some studies require the subjects to **act on the code**. In ten studies subjects were asked to find and

TABLE I: Task types and their corresponding studies. A study may involve more than one type of task.

Task type	Studies
provide information about the code (40 studies)	
explain what the code does	18 studies: [7], [8], [10], [11], [14], [15], [19], [42], [44], [46]–[54]
answer questions about code characteristics	27 studies: [1], [2], [4], [9], [12], [13], [15], [18], [20], [40], [41], [45], [46], [49], [50], [53]–[64]
remember (part of) the code	7 studies: [7], [9], [15], [42], [47], [51], [52]
act on the code (15 studies)	
find and fix bugs in the code	10 studies: [6], [10], [18], [19], [43]–[45], [65]–[67]
modify the code	8 studies: [10], [12], [44], [45], [55], [61], [65], [66]
write code	3 studies: [60], [61], [68]
provide personal opinion (30 studies)	
opinion about the code (readability or legibility)	23 studies: [3]–[5], [8], [10], [12], [21], [44], [48], [54], [58], [60], [61], [63], [64], [66], [68]–[74]
answer if understood the code	4 studies: [41], [54], [58], [59]
rate confidence in her answer	3 studies: [7], [20], [52]
rate the task difficult	7 studies: [4], [12], [19], [20], [44], [48], [50]

fix bugs in the code. Scanniello et al. [43] asked participants to do so in two programs with different identifier styles. In eight other studies the subjects were asked to modify the code of a working program, i.e., without the need to fix bugs. For example, Jbara and Feitelson [44] asked subjects to implement a new feature in a program seen in a previous task. Likewise, Schulze et al. [45] requested that participants modify and delete annotated code with specific preprocessor directives, which also requires understanding the respective source code.

In a few studies, subjects were asked to write code from a description. Writing code *per se* is not a comprehension task, but it may be associated to a comprehension task. For example, Wiese et al. [60] first asked subjects to write a function that returns true if the input is 7 and false otherwise so that they could identify what code style (novice, expert, or mixed) the subjects preferred when coding. Afterwards, they asked the subjects to choose the most readable among three versions of a function, each one with a different code style. One of the goals of this study was to determine if the subjects write code in the same style that they find more readable.

Lastly, 30 studies require that subjects give their **personal opinion**. Nine studies only inquired the subjects about their personal preferences or gut feeling, without any additional task. For example, Buse et al. [21] asked them to rate (from 1 to 5) how legible or readable a code snippet is. Similarly, Arab [5] asked subjects to classify the legibility of three presentation schemes of Pascal code in descending order. In the study of Santos and Gerosa [73], the subjects chose the most readable between two functionally-equivalent snippets. Other studies, 21 in total, asked subjects about their personal opinions while requiring them to perform other tasks. In the study conducted by O'Neal et al. [54], subjects first read a code snippet. They then stated whether they understood the snippet or not and provided a written description of its functionality. Similarly, Lawrie et al. [7] asked subjects to provide a free-form written description of the purpose of a function and to rate their confidence in their description. In addition, some studies ask the subjects to rate the difficulty of the comprehension tasks they had to perform, e.g., the work of Fakhoury et al. [19].

Summary: To assess code readability and legibility, human-centric studies ask subjects to provide information about the code, act on the code, or give their personal opinion. We found out that 16.7% of the studies only ask the subjects to provide personal opinion. Moreover, although most of the analyzed studies require subjects to provide information about source code, that information varies widely in nature. Subjects may be asked to predict the output of programs, identify code elements, or explain high-level functionality.

B. Response Variables (RQ2)

Depending on the goals, methodology, and subjects of a study, response variables vary. This section presents the results obtained in the analysis of the response variables used in the selected studies. Since there is considerable diversity of response variables, we have organized the response variables into five categories. Figure 2 presents the frequency of all response variables. The numbers do not add up to 54, the overall number of analyzed studies, because most studies employ more than one response variable. In Table II, we present a detailed synthesis of the identified response variables.

Some studies measure the performance of subjects in terms of whether they are able to correctly provide information about programs just by looking at the source code. The response variables may pertain to code structure, semantics, use of algorithms, or program behavior, for instance. We aggregated response variables like these into a category called **correctness**. Correctness variables may be objective or subjective. In the former case, correctness can be objectively determined. For example, Bauer et al. [4] measured the subjects' code understanding by asking them to fill in a questionnaire with multiple-choice questions referring to program output. Gopstein et al. [1] and Ajami and Feitelson [2] asked the subjects to predict the outputs of the execution of short programs. In studies that employ subjective correctness response variables, there is some margin to interpret whether the results produced by a subject are correct. For example, Blinman et al. [11] evaluated if the subjects' textual description of a program is correct. Similarly, Love [42] asked subjects to write a textual description of a program and scored it based on a subjective evaluation. Correctness response variables are employed by 83.3% of the primary studies.

The second most often employed response variable, present in 30 studies, is the subjects' personal opinion. What is common to all these studies is the use of the preferences and gut feeling of the subjects, instead of the results of what they do, to assess readability and legibility. We grouped these response variables in a category called **opinion**. Scalabrino et al. [41], for example, asked the subjects to state whether they understood a code snippet or not. Santos and Gerosa [73] presented pairs of functionally-equivalent snippets to the subjects and asked them to choose which one they think is more readable or legible. Stefik and Gellenbeck [72] requested that subjects rate lists of words/symbols associated to programming constructs (e.g., conditionals and loops) based on how intuitive
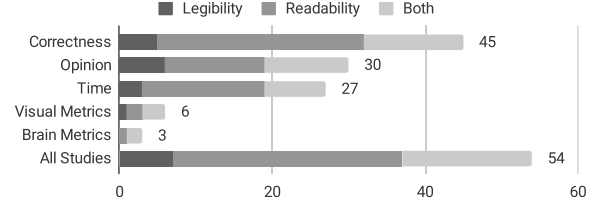

Fig. 2: Frequency of response variables.

TABLE II: Response variables and their corresponding studies.

Category and type	Sub-type	Studies
Correctness (45 studies)		
Objective	Binary	37 studies: [1], [2], [4], [6], [7], [9], [11]–[13], [15], [18]–[20], [40]–[44], [46], [50]–[67]
	Scale	3 studies: [10], [45], [68]
Subjective	Binary	11 studies: [8], [14], [19], [46], [47], [49]–[52], [60], [61]
	Rate	7 studies: [7], [10], [15], [42], [44], [48], [53]
Opinion (30 studies)		
Personal preference	Rate	9 studies: [21], [63], [64], [68]–[72], [74]
	Choice	6 studies: [3], [8], [10], [60], [61], [73]
	Ranking	1 study: [5]
On understandability	Binary	2 studies: [41], [59]
	Rate	2 studies: [54], [58]
Professional opinion	Acceptability	2 studies: [66], [74]
Rate answer confidence	Rate	3 studies: [7], [20], [52]
Rate task difficulty	Ranking	1 study: [4]
	Rate	6 studies: [12], [19], [20], [44], [48], [50]
Time (27 studies)		
Time to complete task		20 studies: [2], [4], [6], [10]–[12], [14], [15], [40], [43]–[45], [48], [54], [56], [63]–[67]
Time reading code		5 studies: [8], [15], [19], [41], [46]
Time per question		4 studies: [9], [46], [55], [57]
Number of attempts		2 studies: [57], [66]
Visual Metrics (6 studies)		
Eye tracking		4 studies: [4], [15], [19], [48]
Letterboxing		2 studies: [6], [67]
Brain Metrics (3 studies)		
fMRI		1 study: [18]
fNIRS		1 study: [19]
EEG		1 study: [20]

they think they are. Lawrie et al. [7] asked the subjects to rate their confidence in their understanding of the code.

Multiple studies measure the **time** spent to perform tasks. This response variable category is the third most often used in the analyzed studies, with 27 instances. There is variety in the way different studies measure time. For example, for Ajami and Feitelson [2], *"time is measured from displaying the code until the subject presses the button to indicate he is done"*. Hofmeister et al. [67] compute the time subjects spend looking at specific parts of a program. Geffen et al. [55] measure the response time for each question when subjects answer a multiple-choice questionnaire about the code. Instead of directly measuring time, Malaquias et al. [66] count the attempts to fix syntactic and semantic errors in the code.

Many studies collect information about the process of the task instead of its outcomes. In particular, multiple studies employ special equipment to track what the subjects see. These studies employ **visual metrics** as response variables. Six of the analyzed studies employ some form of eye tracking. For example, Blinkley et al. [15] computed the visual attention, measured as the amount of time during which a subject is

looking at a particular area of the screen. Bauer et al. [4] computed three visual metrics: fixation duration, fixation rate, i.e., number of fixations per second, and saccadic amplitude, i.e., the spatial length of a saccade, that is, the transition between two fixations. Hofmeister et al. [67] employ a software tool that limits the subjects' view of the code to a few lines at a time. This frame can be shifted up and down using the arrow keys to reveal different parts of the code. This approach is called "letterboxing". Hofmeister et al. [67] called each code frame an area of interest (AOI), and measured the time subjects spent on an AOI, first-pass reading times, and AOI visits.

Recently, some researchers have resorted to leveraging brain monitoring tools to understand what happens in the brain of the subjects during program comprehension. In total, three of the analyzed studies employ response variables based on brain monitoring. Siegmund et al. [18] used functional magnetic resonance imaging (fMRI) to measure brain activity by detecting changes associated with blood flow. Fakhoury et al. [19] employed functional near-infrared spectroscopy (fNIRS) to measure brain activity through the hemodynamic response within physical structures of the brain. Yeh et al. [20] leveraged electroencephalography (EEG) to monitor the electrical activity of the brain. These response variables were grouped into the **brain metrics** category.

The analyzed studies differ in the response variables they employ, depending on whether they aim to assess readability, legibility, or both. As shown in Figure 2, readability studies leverage all the response variable categories whereas no legibility study leverages brain metrics. This is not surprising, considering the much lower number of legibility studies. In addition, a high proportion of legibility studies (86%) employ opinion response variables. This is not the case for studies only about readability (43%) or studies about both (65%).

Most studies, 38 in total, use more than one variable to validate their hypotheses. The response variables time and correctness are the ones that most often appear together (27 papers), followed by opinion and correctness (21 papers). In a different manner, 16 papers use only one response variable, correctness and opinion were the only response variables employed in isolation. Among these, 9 papers (16.7%) use only personal opinion. The use of this response variable category in isolation is a clear threat to the validity of these studies. Furthermore, no study uses time as the only response variable. This makes sense since, even though time is often used as a proxy for effort, it is not significant by itself.

Summary: There are five categories of response variables. Correctness is the most widely used, being employed in 83.3% of the analyzed studies. Time and opinion are also used often (50% and 55.6% of the studies, respectively). A significant number of studies use the variable time and/or opinion associated with correctness. On the other hand, 30% of the studies employ a single response variable. The readability and readability+legibility papers used all response variable categories while almost all legibility studies used the opinion response variable.

V. PROGRAM COMPREHENSION AS A LEARNING ACTIVITY

The empirical studies discussed in Section IV-A involve a wide range of tasks to be performed by their subjects. For example, they may ask subjects to memorize a program, follow its execution step by step, answer questions about it, or write a high-level explanation of what it does. All these tasks are conducted with the goal of evaluating readability and legibility. However, they demand different cognitive skills from the subjects and, as a consequence, evaluate different aspects of readability and legibility. We attempt to shed a light on this topic by analyzing the cognitive skill requirements associated with each kind of task.

According to the Merriam-Webster Thesaurus, to learn something is *"go gain an understanding of"* it. We follow this definition by treating the problem of program comprehension (or understanding) as a learning problem. In this section we propose an adaptation of the learning taxonomy devised by Fuller and colleagues [75] to the context of program comprehension. This taxonomy is itself an adaptation of Bloom's taxonomy of educational objectives [76] to the context of Software Development. Fuller et al. [75] state that *"learning taxonomies [..] describe the learning stages at which a learner is operating for a certain topic."* A learning taxonomy supports educators in establishing intended learning outcomes for courses and evaluate the students' success in meeting these goals. According to Biggs [77] and Fuller et al. [75], learning taxonomies can help with "understanding about understanding" and "communicating about understanding". Based on this description, they seem to be a good fit to help researchers better understand studies about program understanding and communicate about them. Our central idea is to use the elements defined by the taxonomy of Fuller et al. [75], with some adaptions, to identify and name the cognitive skills required by different tasks employed by code readability and legibility studies.

A. A Learning Taxonomy

Bloom's taxonomy [76] consists of three hierarchical models aiming to classify educational learning objectives in terms of complexity and specificity. In the context of this work, the cognitive domain of this taxonomy is the most relevant. It recognizes that learning is a multi-faceted process that requires multiple skills with different levels of complexity and which build upon each other. This taxonomy has been later refined by Anderson et al. [78]. The most visible aspect of the refined taxonomy is a list of six activities that define progressively more sophisticated levels of learning: remember, understand, apply, analyze, evaluate, and create.

Fuller et al. [75] proposed an adaptation of the refined version of Bloom's taxonomy for the area of Computer Science, with a particular emphasis on software development. They developed a set of activities that build upon Bloom's revised taxonomy and organized them in a model that emphasizes that some activities in software development involve acting on knowledge, instead of just learning. We leverage this taxonomy and apply it in the context of program comprehension.

TABLE III: Learning activities extended from Fuller et al. [75]—Inspect and Memorize are not in the original taxonomy. Opinion is not included because it is not directly related to learning.

Activity	Description	Example
Adapt	Modify a solution for other domains/ranges. This activity is about the modification of a solution to fit in a given context.	Remove preprocessor directives to reduce variability in a family of systems [45]
Analyze	Probe the [time] complexity of a solution.	Identify the function where the program spends more time running
Apply	Use a solution as a component in a larger problem. Apply is about changing a context so that an existing solution fits in it.	Reuse of off-the-shelf components
Debug	Both detect and correct flaws in a design	Given a program, identify faults and fix them [43]
Design	Devise a solution structure. The input to this activity is a problem specification.	Given a problem specification, devise a solution satisfying that specification
Implement	Put into lowest level, as in coding a solution, given a completed design	Write code using the given examples according to a specification [68]
Model	Illustrate or create an abstraction of a solution. The input is a design.	Given a solution, construct a UML model representing it
Present	Explain a solution to others.	Read a program and then write a description of what it does and how [47]
Recognize	Base knowledge, vocabulary of the domain. In this activity, the subject must identify a concept or code structure obtained before the task to be performed.	"The sorting algorithm (lines 46-62) can best be described as: (A) bubble sort (B) selection sort (C) heap sort (D) string sort (E) partition exchange sort" [62]
Refactor	Redesign a solution (as for optimization). The goal is to modify non-functional properties of a program or, at a larger scale, reengineer it	Rewrite a function so as to avoid using conditional expressions
Relate	Understand a solution in context of others. This activity is about identifying distinctions and similarities, pros and cons of different solutions.	Choose one out of three high-level descriptions that best describe the function of a previously studied application [11]
Trace	Desk-check a solution. Simulate program execution while looking at its code.	Consider the fragment "x=++y": what is the final value of x if y is -10? [40]
Inspect*	Examine code to find or understand fine-grain static elements. Inspect is similar to Analyze, but it happens at compile time instead of run time.	"All variables in this program are global." [true/false] [50]
Memorize*	Memorize the code in order to reconstruct it later, partially or as a whole	Given a program, memorize it in 3 minutes and then reconstruct it in 4 [42]

Table III presents the activities (i.e., cognitive skills) devised by Fuller et al. [75]. Also, we introduce two activities (marked with "*") that stem directly from tasks performed by subjects in some of our primary studies and require skills that are not covered by the original set of activities. Table III also presents examples of the activities, which are either extracted from tasks from our primary studies or general examples when no task involved that activity. In the next section we leverage these activities to gain a better understanding of the tasks conducted by subjects in our primary studies.

B. Mapping Tasks to Learning Activities

We analyzed each task in the primary studies and identified which activities from Table III they performed. A task can require subjects to conduct one or more activities. For example, Avidan and Feitelson [8] asked subjects to explain what a function does. This is an instance of the Present activity. Miara et al. [50] applied a questionnaire about code characteristics that asked subjects to identify specific code elements and also inquired them about the results of expression evaluations. In this case, both Inspect and Trace activities were executed. Besides the activities in Table III, we also considered Giving an Opinion (hereafter, "Opinion"), as a special case that is not part of the taxonomy because it is not a learning activity. We include Opinion to reflect the intrinsically biased and unreliable nature of tasks that ask subjects to provide their opinion. Previous work has shown that evidence in software engineering studies [79], [80] often contradicts opinions. This kind of analysis falls outside the scope of this paper.

Figure 3 presents a confusion matrix to show the frequency of studies in which tasks and learning activities co-occur. For instance, there are six studies that involves the "Remember the code" task in the context of the Memorize learning activity. The matrix shows that there is a direct correspondence between some tasks and activities. For example, all the instances of the "Find and fix bugs in the code" task involve the Debug activity, and all the tasks that require subjects to provide an opinion are connected to the Opinion activity. In addition,
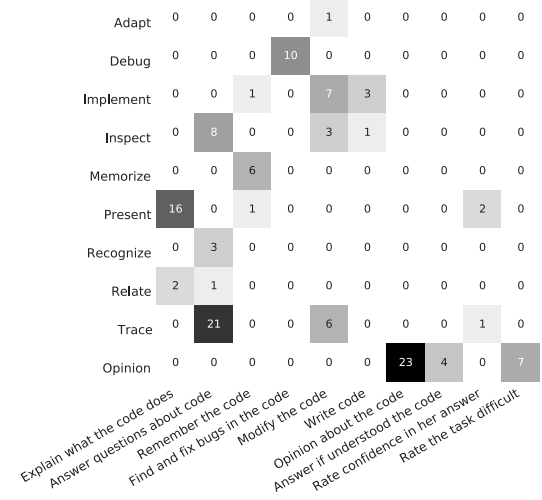

Fig. 3: Confusion matrix of tasks (columns) and learning activities (rows).

some tasks may be connected to various activities. For instance, "Modify the code" may require subjects to Implement, Trace, Adapt, or Inspect the code to be modified. This makes sense; to implement a modification in a program, one may have to understand its static elements and its behavior, as well as adapt code elements to be reused. Another example is "Answer questions about code", which often requires subjects to Trace and Inspect code. Furthermore, "Explain what the code does" is usually related to Present. Notwithstanding, in two studies [11], [54] subjects were presented with multiple descriptions for the same code snippet and asked which one is the most appropriate. This requires the subject to Relate different programs and descriptions. Finally, there are some nonintuitive relationships between tasks and activities. Chaudhary [47] asked the subjects of the study to reconstruct a program after spending some time looking at it. This is a "Remember the code" task where the subject has to explain what the program does by writing a similar program. It is a

mix of Present and Implement. In another example, Lawrie et al. [7] asked the subjects to describe what a code snippet does and rate their confidence in their answers. Rating one's confidence in a response must be associated with some other activity, since some response must have been given in the first place. In this case, the activity is to Present the code snippet.

The two most widely used activities from the taxonomy require that subject extract information from programs. 25 studies require that subjects Trace a program; these studies ask the subjects to predict the output value of a program or the value of a variable. Coming in second, 16 studies employ the Present activity, usually to ask the subjects to explain in general terms what the program does. It is also common for subjects to be asked to Inspect the code, so as to determine, e.g., how many loops are there in the program. Overall, 11 studies employ this activity. Subjects are required to Implement or Debug code in 10 studies each.

Figure 4 shows how many times each activity is required, separating readability and legibility studies, as well as those targeting both attributes. The figure shows that Opinion is required by more studies than any activity from the taxonomy. The figure also highlights that studies focusing solely on legibility and on legibility and readability combined require subjects to give an Opinion more than any other activity. Even considering the small number of legibility studies, this may be a hint that researchers have yet to figure out the effective ways to meaningfully assess legibility-related attributes.

Figure 5 highlights how many studies use combinations of two activities. More than one activity is employed by 34 of the studies, which combine mainly the most popular activities, i.e., Trace, Present, Implement and Inspect. The numbers in the diagonal indicate how many studies require their subjects to use a single activity. In total, 20 studies employ a single activity. This number amounts to 37% of our primary studies. The use of a single activity in an evaluation of readability or legibility suggests a narrow focus, since these attributes are multi-faceted, as highlighted by Table III.

C. A Two-Dimensional Model for Program Comprehension

Besides the list of activities, Fuller et al. [75] proposed a taxonomy adapted from the revised version of Bloom's taxonomy [78]. It is represented by two semi-independent dimensions, Producing and Interpreting. Each dimension defines hierarchical linear levels where a deeper level requires the competencies from the previous ones. Production has three levels (None, Apply, and Create) and Interpretation has four (Remember, Understand, Analyze, and Evaluate). Figure 6 represents this two-dimensional model. In this section we do not consider Opinion, as it is not part of the learning taxonomy.

The activities in Table III were positioned in the figure in conformance to their required competencies. According to Fuller et al. [75], a level appearing more to the right of the figure along the Interpreting dimension, e.g., Evaluate, requires more competencies than one appearing more to the left, e.g., Remember. The same applies to the Levels appearing nearer the top along the Producing dimension. We complement the
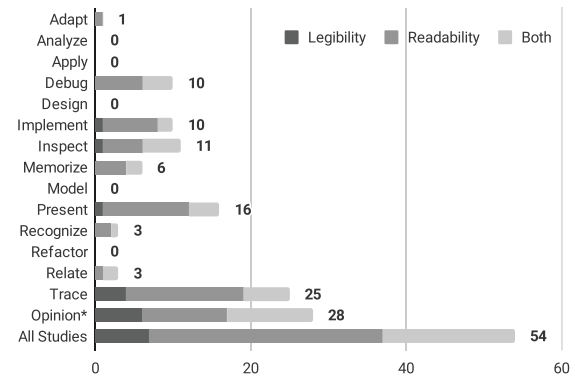

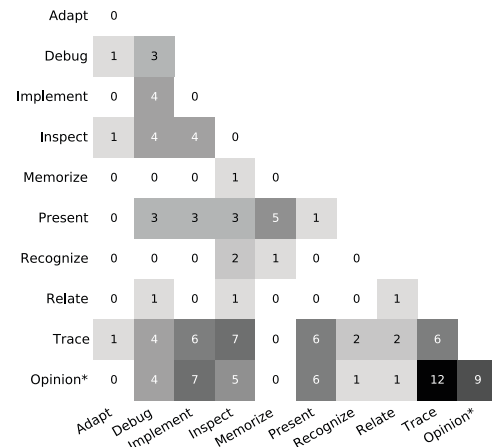Fig. 4: Frequency of learning activities and subject opinion.

Fig. 5: Co-occurrence of learning activities and subject opinion—the main diagonal represents the number of studies that an activity or opinion is used alone.

two-dimensional model by including the two activities that we introduced, Inspect and Memorize. Figure 6 is a heatmap that presents how frequently each competence was required by studies analyzed in this work. Dark regions in the figure are associated to more significant activities.

The Interpreting dimension (x-axis) of Figure 6 indicates that most activities employed in studies evaluating readability and legibility occur at the Understand level, followed by Analyze and, to a lesser extent, Remember. Evaluate, the higher competence level of the Interpreting dimension, is almost never used. Considering the Producing dimension (y-axis), where the ability to design and build a new product, e.g., a program, is evaluated, we notice that None is the most representative, followed by Apply. This is to be expected since most of these studies focus on comprehension-related activities. However, program comprehension is often required for work that involves the Apply and Create levels. Nevertheless, these levels are rarely tackled by our primary studies. This reinforces the point raised in Section V-B, that many of these studies have a narrow evaluation focus.

We build three additional heatmaps to emphasize the differences between legibility and readability studies when consider-

ing the two-dimensional model. Figure 7 presents the heatmaps for readability studies, legibility studies, and both studies in sequence. Figure 7a is very similar to Figure 6. In contrast, Figure 7b shows that legibility studies are concentrated in the Understand level, where Trace is the main activity. Finally, Figure 7c presents the heatmap for studies that tackle both readability and legibility together. Albeit similar to Figure 6, the area in the intersection between the Analyze level of the Interpreting dimension and the Apply level of the Producing dimension is darker in this figure. This reflects the proportionally higher number of studies that employ the Debug activity.

VI. THREATS TO VALIDITY

Construct validity. Our study was built on the selected primary studies, which stem from the search and selection processes. The search for studies relies on a search string, which was defined based on the seed papers. We chose only conferences to search for seed papers. A paper published in a journal is often an extension of a conference paper and, in Computer Science, the latest research is published in conferences. Furthermore, we focused on conferences only as a means to build our search string. The actual survey did not exclude journals. Additionally, we only used three search engines for our automatic search. Other engines, such as Springer and Google Scholar, could return different results. However, the majority of our seed studies were indexed by ACM and IEEE, and then we used Scopus to expand our search. Moreover, while searching on the ACM digital library, we used the *Guide to the Computing Literature*, which retrieves resources from other publishers, such as Springer. Finally, we avoided Google Scholar because it returned more than 17 thousand documents for our search string.

Internal validity. This study was conducted by four researchers. We understand that this could pose a threat to its internal validity since each researcher has a certain knowledge and way of conducting her research activities. However, each researcher conducted her activities according to the established protocol, and periodic discussions were conducted between all researchers. Another threat to validity is the value of Cohen's Kappa in the study inclusion step ($k = 0.323$), which is considered fair. This value stems from the use of three possible evaluations ("acceptable", "not acceptable", and "maybe") in that step. However, we employed "maybe" to avoid having to choose between "acceptable" and "not acceptable" when we had doubts about the inclusion of a paper—all papers marked with at least one "maybe" were discussed between all authors. Moreover, a few primary studies do not report in detail the tasks the subjects performed. Because of that, we might have misclassified these studies when mapping studies to task types.

External validity. Our study focuses on studies that report on comparisons of alternative ways of writing code, considering low-level aspects of the code. Our findings might not apply to other works that evaluate code readability or legibility.

Conclusion validity. According to Kitchenham et al. [32], conclusion validity is concerned with how reliably we can draw conclusions about the relationship between a treatment
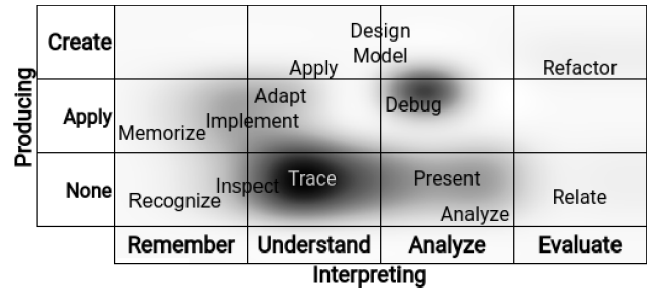

Fig. 6: Learning activities and their frequency.

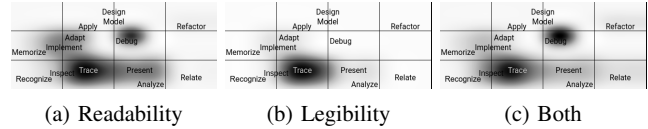(a) Readability   (b) Legibility   (c) Both

Fig. 7: Learning activities by readability and legibility. This figure decomposes Figure 6 to show spatial disposition.

and the outcomes of an empirical study. In a systematic review, that relates to the data synthesis and how well this supports the conclusions of the review. The threats of our study related to it are therefore presented as internal validity.

VII. CONCLUSION

We presented a systematic literature review on empirical studies about code readability and legibility comprehension, in particular those that compare two or more ways of writing the same code. Our goal was to study what types of task are performed by human subjects, what are the response variables of these studies, and what cognitive skills are required by those tasks. We presented comprehensive classifications for both tasks and response variables. We also introduced a separation between code readability and legibility. This separation is common in other areas such as linguistics, design, human-computer interaction, and education, but, to the best of our knowledge, we are the first to propose it in the context of software engineering. Moreover, we have adapted a well-known learning taxonomy to program comprehension, mapping the tasks identified in the literature review to the cognitive skills (activities) that comprise the taxonomy. This study highlighted limitations of primary studies: (i) 37% of them exercise a single cognitive skill; (ii) 16.7% only employ personal opinion as a response variable; and (iii) few studies evaluate readability and legibility in a way that simulates real-world scenarios, where program comprehension is part of a more complex task, requiring higher-level cognitive skills. We are currently performing a systematic literature review focusing on the different ways of writing code and which ones improve or hinder readability and legibility.

R<small>EFERENCES</small>

[1] D. Gopstein, J. Iannacone, Y. Yan, L. DeLong, Y. Zhuang, M. K.-C. Yeh, and J. Cappos, "Understanding Misunderstandings in Source Code," in *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE '17)*. New York, NY, USA: ACM, 2017, pp. 129–139.

[2] S. Ajami, Y. Woodbridge, and D. G. Feitelson, "Syntax, predicates, idioms – what really affects code complexity?" *Empirical Software Engineering*, vol. 24, no. 1, pp. 287–328, Feb. 2019.

[3] X. Wang, L. Pollock, and K. Vijay-Shanker, "Automatic Segmentation of Method Code into Meaningful Blocks: Design and Evaluation," *Journal of Software: Evolution and Process*, vol. 26, no. 1, pp. 27–49, 2014.

[4] J. Bauer, J. Siegmund, N. Peitek, J. C. Hofmeister, and S. Apel, "Indentation: Simply a Matter of Style or Support for Program Comprehension?" in *Proceedings of the 27th International Conference on Program Comprehension (ICPC '19)*. Piscataway, NJ, USA: IEEE Press, 2019, pp. 154–164.

[5] M. Arab, "Enhancing Program Comprehension: Formatting and Documenting," *ACM SIGPLAN Notices*, vol. 27, no. 2, pp. 37–46, Feb. 1992.

[6] A. Schankin, A. Berger, D. V. Holt, J. C. Hofmeister, T. Riedel, and M. Beigl, "Descriptive Compound Identifier Names Improve Source Code Comprehension," in *Proceedings of the 26th International Conference on Program Comprehension (ICPC '18)*. New York, NY, USA: ACM, 2018, pp. 31–40.

[7] D. Lawrie, C. Morrell, H. Feild, and D. Binkley, "Effective identifier names for comprehension and memory," *Innovations in Systems and Software Engineering*, vol. 3, no. 4, pp. 303–318, Dec 2007.

[8] E. Avidan and D. G. Feitelson, "Effects of Variable Names on Comprehension: An Empirical Study," in *Proceedings of the 25th International Conference on Program Comprehension (ICPC '17)*. Piscataway, NJ, USA: IEEE Press, 2017, pp. 55–65.

[9] D. Binkley, D. Lawrie, S. Maex, and C. Morrell, "Identifier length and limited programmer memory," *Science of Computer Programming*, vol. 74, no. 7, pp. 430–445, May 2009.

[10] G. Beniamini, S. Gingichashvili, A. K. Orbach, and D. G. Feitelson, "Meaningful Identifier Names: The Case of Single-Letter Variables," in *Proceedings of the 25th International Conference on Program Comprehension (ICPC '17)*. Piscataway, NJ, USA: IEEE Press, 2017, pp. 45–54.

[11] S. Blinman and A. Cockburn, "Program Comprehension: Investigating the Effects of Naming Style and Documentation," in *Proceedings of the 6th Australasian User Interface Conference (AUIC '05)*. Newcastle, Australia: ACS, 2005, pp. 73–78.

[12] M. Ceccato, M. Di Penta, J. Nagra, P. Falcarin, F. Ricca, M. Torchiano, and P. Tonella, "The Effectiveness of Source Code Obfuscation: an Experimental Assessment," in *Proceedings of the IEEE 17th International Conference on Program Comprehension (ICPC '09)*. IEEE, 2009, pp. 178–187.

[13] B. E. Teasley, "The effects of naming style and expertise on program comprehension," *International Journal of Human-Computer Studies*, vol. 40, no. 5, pp. 757–770, May 1994.

[14] A. C. Benander, B. A. Benander, and H. Pu, "Recursion vs. Iteration: An Empirical Study of Comprehension," *Journal of Systems and Software*, vol. 32, no. 1, pp. 73–82, 1996.

[15] D. Binkley, M. Davis, D. Lawrie, J. I. Maletic, C. Morrell, and B. Sharif, "The impact of identifier style on effort and comprehension," *Empirical Software Engineering*, vol. 18, no. 2, pp. 219–276, Apr. 2013.

[16] J. Siegmund, "Program comprehension: Past, present, and future," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 5, 2016, pp. 13–20.

[17] I. Schröter, J. Krüger, J. Siegmund, and T. Leich, "Comprehending studies on program comprehension," in *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*. IEEE, 2017, pp. 308–311.

[18] J. Siegmund, N. Peitek, C. Parnin, S. Apel, J. Hofmeister, C. Kästner, A. Begel, A. Bethmann, and A. Brechmann, "Measuring Neural Efficiency of Program Comprehension," in *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE '17)*. New York, NY, USA: ACM, 2017, pp. 140–150.

[19] S. Fakhoury, D. Roy, Y. Ma, V. Arnaoudova, and O. Adesope, "Measuring the impact of lexical and structural inconsistencies on developers' cognitive load during bug localization," *Empirical Software Engineering*, pp. 1–39, Aug. 2019.

[20] M. K.-C. Yeh, D. Gopstein, Y. Yan, and Y. Zhuang, "Detecting and Comparing Brain Activity in Short Program Comprehension Using EEG," in *2017 IEEE Frontiers in Education Conference (FIE)*. Los Alamitos, CA, USA: IEEE Computer Society, 2017, pp. 1–5.

[21] R. P. L. Buse and W. R. Weimer, "Learning a Metric for Code Readability," *IEEE Transactions on Software Engineering*, vol. 36, no. 4, pp. 546–558, Jul. 2010.

[22] J. R. de Almeida, J. B. Camargo, B. A. Basseto, and S. M. Paz, "Best practices in code inspection for safety-critical software," *IEEE software*, vol. 20, no. 3, pp. 56–63, 2003.

[23] Jin-Cherng Lin and Kuo-Chiang Wu, "Evaluation of software understandability based on fuzzy matrix," in *2008 IEEE International Conference on Fuzzy Systems (IEEE World Congress on Computational Intelligence)*, 2008, pp. 887–892.

[24] X. Xia, L. Bao, D. Lo, Z. Xing, A. E. Hassan, and S. Li, "Measuring program comprehension: A large-scale field study with professionals," *IEEE TSE*, vol. 44, no. 10, pp. 951–976, 2018.

[25] P. B. Gough and W. E. Tunmer, "Decoding, reading, and reading disability," *Remedial and special education*, vol. 7, no. 1, pp. 6–10, 1986.

[26] W. A. Hoover and P. B. Gough, "The simple view of reading," *Reading and writing*, vol. 2, no. 2, pp. 127–160, 1990.

[27] W. H. DuBay, "The principles of readability." *Online Submission*, 2004.

[28] C. Tekfi, "Readability formulas: An overview," *Journal of documentation*, 1987.

[29] E. Daka, J. Campos, G. Fraser, J. Dorn, and W. Weimer, "Modeling readability to improve unit tests," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: Association for Computing Machinery, 2015, p. 107–118.

[30] I. Strizver, *Type Rules: The designer's guide to professional typography*. John Wiley & Sons, 2013.

[31] S. Zuffi, C. Brambilla, G. Beretta, and P. Scala, "Human computer interaction: Legibility and contrast," in *14th International Conference on Image Analysis and Processing (ICIAP 2007)*. IEEE, 2007, pp. 241–246.

[32] B. A. Kitchenham, D. Budgen, and P. Brereton, *Evidence-Based Software Engineering and Systematic Reviews*. Chapman & Hall/CRC, 2015.

[33] *Parsifal*, https://parsif.al/.

[34] *ACM Digital Library*, http://dl.acm.org/.

[35] *IEEE Explore*, http://ieeexplore.ieee.org/.

[36] *Scopus*, http://www.scopus.com/.

[37] J. Cohen, "A Coefficient of Agreement for Nominal Scales," *Educational and Psychological Measurement*, vol. 20, no. 1, pp. 37–46, 1960.

[38] S. Keele *et al.*, "Guidelines for performing systematic literature reviews in software engineering," Technical report, Ver. 2.3 EBSE Technical Report. EBSE, Tech. Rep., 2007.

[39] R. P. Buse and W. R. Weimer, "A Metric for Software Readability," in *Proceedings of the 2008 International Symposium on Software Testing and Analysis (ISSTA '08)*. New York, NY, USA: ACM, 2008, pp. 121–130.

[40] J. J. Dolado, M. Harman, M. C. Otero, and L. Hu, "An Empirical Investigation of the Influence of a Type of Side Effects on Program Comprehension," *IEEE Transactions on Software Engineering*, vol. 29, no. 7, pp. 665–670, Jul. 2003.

[41] S. Scalabrino, G. Bavota, C. Vendome, M. Linares-Vásquez, D. Poshyvanyk, and R. Oliveto, "Automatically Assessing Code Understandability," *IEEE Transactions on Software Engineering*, pp. 1–1, 2019.

[42] T. Love, "An Experimental Investigation of the Effect of Program Structure on Program Understanding," *ACM SIGOPS Operating Systems Review – Proceedings of an ACM conference on Language design for reliable software*, vol. 11, no. 2, pp. 105–113, Mar. 1977.

[43] G. Scanniello and M. Risi, "Dealing with Faults in Source Code: Abbreviated vs. Full-Word Identifier Names," in *Proceedings of the 2013 IEEE International Conference on Software Maintenance (ICSM '13)*. USA: IEEE Computer Society, 2013, pp. 190–199.

[44] A. Jbara and D. G. Feitelson, "On the Effect of Code Regularity on Comprehension," in *Proceedings of the 22nd International Conference*

on Program Comprehension (ICPC '14). New York, NY, USA: ACM, 2014, pp. 189–200.

[45] S. Schulze, J. Liebig, J. Siegmund, and S. Apel, "Does the Discipline of Preprocessor Annotations Matter? A Controlled Experiment," in *Proceedings of the 12th International Conference on Generative Programming: Concepts & Experiences (GPCE '13).* New York, NY, USA: ACM, 2013, pp. 65–74.

[46] J. P. Boysen and R. F. Keller, "Measuring Computer Program Comprehension," in *Proceedings of the 11th Technical Symposium on Computer Science Education (SIGCSE '80).* New York, NY, USA: ACM, 1980, pp. 92–102.

[47] B. D. Chaudhary and H. V. Sahasrabuddhe, "Meaningfulness as a Factor of Program Complexity," in *Proceedings of the ACM 1980 Annual Conference (ACM '80).* New York, NY, USA: ACM, 1980, pp. 457–466.

[48] A. Jbara and D. G. Feitelson, "How programmers read regular code: a controlled experiment using eye tracking," *Empirical Software Engineering*, vol. 22, no. 3, pp. 1440–1477, Jun. 2017.

[49] N. Kasto and J. Whalley, "Measuring the difficulty of code comprehension tasks using software metrics," in *Proceedings of the 15th Australasian Computing Education Conference - Volume 136 (ACE '13).* Darlinghurst, Australia, Australia: Australian Computer Society, Inc., 2013, pp. 59–65.

[50] R. J. Miara, J. A. Musselman, J. A. Navarro, and B. Shneiderman, "Program Indentation and Comprehensibility," *Communications of the ACM*, vol. 26, no. 11, pp. 861–867, Nov. 1983.

[51] S. Wiedenbeck, "Beacons in computer program comprehension," *International Journal of Man-Machine Studies*, vol. 25, no. 6, pp. 697–709, Dec. 1986.

[52] ——, "The initial stage of program comprehension," *International Journal of Man-Machine Studies*, vol. 35, no. 4, pp. 517–540, Nov. 1991.

[53] S. N. Woodfield, H. E. Dunsmore, and V. Y. Shen, "The Effect of Modularization and Comments on Program Comprehension," in *Proceedings of the 5th International Conference on Software Engineering (ICSE '81).* Piscataway, NJ, USA: IEEE Press, 1981, pp. 215–223.

[54] M. B. O'Neal and W. R. Edwards, "Complexity Measures for Rule-Based Programs," *IEEE Transactions on Knowledge and Data Engineering*, vol. 6, no. 5, pp. 669–680, Oct. 1994.

[55] Y. Geffen and S. Maoz, "On Method Ordering," in *Proceedings of the 24th International Conference on Program Comprehension (ICPC '16).* IEEE, 2016, pp. 1–10.

[56] E. R. Iselin, "Conditional statements, looping constructs, and program comprehension: an experimental study," *International Journal of Man-Machine Studies*, vol. 28, no. 1, pp. 45–66, Jan. 1988.

[57] M. Maćkowiak, J. Nawrocki, and M. Ochodek, "On Some End-User Programming Constructs and Their Understandability," *Journal of Systems and Software*, vol. 142, pp. 206–222, 2018.

[58] F. Sykes, R. T. Tillman, and B. Shneiderman, "The Effect of Scope Delimiters on Program Comprehension," *Software: Practice and Experience*, vol. 13, no. 9, pp. 817–824, 1983.

[59] A. Trockman, K. Cates, M. Mozina, T. Nguyen, C. Kästner, and B. Vasilescu, ""Automatically Assessing Code Understandability" Reanalyzed: Combined Metrics Matter," in *Proceedings of the 15th International Conference on Mining Software Repositories (MSR '18).* New York, NY, USA: ACM, 2018, pp. 314–318.

[60] E. S. Wiese, A. N. Rafferty, and A. Fox, "Linking Code Readability, Structure, and Comprehension among Novices: It's Complicated," in *Proceedings of the 41st International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET '19).* Piscataway, NJ, USA: IEEE Press, 2019, pp. 84–94.

[61] E. S. Wiese, A. N. Rafferty, D. M. Kopta, and J. M. Anderson, "Replicating Novices' Struggles with Coding Style," in *Proceedings of the 27th International Conference on Program Comprehension (ICPC '19).* Piscataway, NJ, USA: IEEE Press, 2019, pp. 13–18.

[64] ——, "Typographic Style is More than Cosmetic," *Communications of the ACM*, vol. 33, no. 5, pp. 506–520, May 1990.

[62] T. Tenny, "Program Readability: Procedures Versus Comments," *IEEE Transactions on Software Engineering*, vol. 14, no. 9, pp. 1271–1279, Sep. 1988.

[63] P. W. Oman and C. R. Cook, "A Paradigm for Programming Style Research," *ACM SIGPLAN Notices*, vol. 23, no. 12, pp. 69–78, Dec. 1988.

[65] S. Kleinschmager, R. Robbes, A. Stefik, S. Hanenberg, and E. Tanter, "Do static type systems improve the maintainability of software systems? an empirical study," in *Proceedings of the 20th IEEE International Conference on Program Comprehension (ICPC '12)*, 2012, pp. 153–162.

[66] R. Malaquias, M. Ribeiro, R. Bonifácio, E. Monteiro, F. Medeiros, A. Garcia, and R. Gheyi, "The Discipline of Preprocessor-Based Annotations Does #ifdef TAG n't #endif Matter," in *Proceedings of the 25th International Conference on Program Comprehension (ICPC '17).* Piscataway, NJ, USA: IEEE Press, 2017, pp. 297–307.

[67] J. C. Hofmeister, J. Siegmund, and D. V. Holt, "Shorter Identifier Names Take Longer to Comprehend," *Empirical Software Engineering*, vol. 24, no. 1, pp. 417–443, Feb. 2019.

[68] A. Stefik and S. Siebert, "An Empirical Investigation into Programming Language Syntax," *ACM Transactions on Computing Education (TOCE)*, vol. 13, no. 4, pp. 19:1–19:40, Nov. 2013.

[69] V. Arnaoudova, M. Di Penta, and G. Antoniol, "Linguistic Antipatterns: What They Are and How Developers Perceive Them," *Empirical Software Engineering*, vol. 21, no. 1, pp. 104–158, Feb. 2016.

[70] D. Posnett, A. Hindle, and P. Devanbu, "A Simpler Model of Software Readability," in *Proceedings of the 8th Working Conference on Mining Software Repositories (MSR '11).* New York, NY, USA: Association for Computing Machinery, 2011, pp. 73–82.

[71] S. Scalabrino, M. Linares-Vásquez, R. Oliveto, and D. Poshyvanyk, "A Comprehensive Model for Code Readability," *Journal of Software: Evolution and Process*, vol. 30, no. 6, p. e1958, 2018.

[72] A. Stefik and E. Gellenbeck, "Empirical studies on programming language stimuli," *Software Quality Journal*, vol. 19, no. 1, pp. 65–99, Mar. 2011.

[73] R. M. a. dos Santos and M. A. Gerosa, "Impacts of Coding Practices on Readability," in *Proceedings of the 26th Conference on Program Comprehension (ICPC '18).* New York, NY, USA: ACM, 2018, pp. 277–285.

[74] F. Medeiros, G. Lima, G. Amaral, S. Apel, C. Kästner, M. Ribeiro, and R. Gheyi, "An investigation of misunderstanding code patterns in C open-source software projects," *Empirical Software Engineering*, vol. 24, no. 4, pp. 1693–1726, Aug. 2019.

[75] U. Fuller, C. G. Johnson, T. Ahoniemi, D. Cukierman, I. Hernán-Losada, J. Jackova, E. Lahtinen, T. L. Lewis, D. M. Thompson, C. Riedesel *et al.*, "Developing a computer science-specific learning taxonomy," *ACM SIGCSE Bulletin*, vol. 39, no. 4, pp. 152–170, 2007.

[76] B. Bloom, M. Engelhart, E. Furst, W. H. Hill, and D. R. Krathwohl, *Taxonomy of educational objectives: The classification of educational goals. Handbook I: Cognitive domain.* New York: David McKay Company, 1956.

[77] J. Biggs, *Teaching for quality learning at university.* Buckingham: Open University Press, 1999.

[78] L. Anderson, B. Bloom, D. Krathwohl, P. Airasian, K. Cruikshank, R. Mayer, P. Pintrich, J. Raths, and M. Wittrock, *A Taxonomy for Learning, Teaching, and Assessing: A Revision of Bloom's Taxonomy of Educational Objectives.* Longman, 2001.

[79] C. J. Rossbach, O. S. Hofmann, and E. Witchel, "Is transactional programming actually easier?" in *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '10).* New York, NY, USA: Association for Computing Machinery, 2010, pp. 47–56.

[80] F. Castor, J. a. P. Oliveira, and A. L. Santos, "Software transactional memory vs. locking in a functional language: A controlled experiment," in *Proceedings of the Compilation of the Co-Located Workshops on DSM'11, TMC'11, AGERE! 2011, AOOPES'11, NEAT'11, & VMIL'11.* Association for Computing Machinery, 2011, pp. 117–122.