Highlights

A Systematic Literature Review on the Impact of Formatting Elements on Program Understandability

Delano Oliveira, Reydne Bruno, Fernanda Madeiral, Hidehiko Masuhara, Fernando Castor

- We conducted a systematic literature review in which, from an initial set of 2,843 documents, we found and examined 12 scientific papers that compared code formatting elements and styles in order to find the most legible ones.
- We found 29 formatting elements (e.g., two-space indentation) that were compared with their equivalent alternatives (e.g., two-space indentation vs. four-space indentation). We organized these elements into five groups: formatting styles, spacing, block delimiters, long or complex code lines, and word boundary styles.
- We also found that, for half of the comparisons, researchers found exclusively statistically significant results, e.g., the camel case for identifier names is preferred over the snake case in terms of legibility. In other cases, researchers found different results, e.g., two-space indentation was considered positive for code understandability by one study, but other two studies did not find a statistical difference between the indentation levels.

A Systematic Literature Review on the Impact of Formatting Elements on Program Understandability

Delano Oliveira[a,b,*], Reydne Bruno[a,*], Fernanda Madeiral[c], Hidehiko Masuhara[d], Fernando Castor[e]

[a] *Federal University of Pernambuco, Recife, Brazil*
[b] *Federal Institute of Pernambuco, Recife, Brazil*
[c] *KTH Royal Institute of Technology, Stockholm, Sweden*
[d] *Tokyo Institute of Technology, Tokyo, Japan*
[e] *Utrecht University, Utrecht, Netherlands*

Abstract

Context: Software programs can be written in different but functionally equivalent ways. Even though previous research has compared specific formatting elements to find out which alternatives affect code legibility, seeing the bigger picture of what makes code more or less legible is challenging. **Goal:** We aim to find which formatting elements have been investigated in empirical studies and which ones were found to be more legible for human subjects. **Method:** We conduct a systematic literature review and identify 12 papers that compare alternative formatting elements. We organize these formatting elements using a card sorting method. **Results:** We identify 29 formatting elements, which are about formatting styles, spacing, block delimiters, long or complex code lines, and word boundary styles. While some formatting elements were found to be statistically better than other equivalent ones, e.g., camel case over underscore, others were not, e.g., pretty-printed vs. disrupted layout. For some alternatives, divergent results were found, e.g., one study found a significant difference in favor of 2-space indentation, but two other studies did not find it. **Conclusion:** Our results can serve as guidelines for developers in choosing ways to format their code and help researchers in developing, e.g., recommendation systems, to aid developers

[*] Authors contributed equally.

Email addresses: `dho@cin.ufpe.br` (Delano Oliveira), `reydne.bruno@gmail.com` (Reydne Bruno), `fer.madeiral@gmail.com` (Fernanda Madeiral), `masuhara@acm.org` (Hidehiko Masuhara), `f.j.castordelimafilho@uu.nl` (Fernando Castor)

to make their code more legible.

Keywords: Program understandability, code legibility, formatting elements.

1. Introduction

Program comprehension is the required activity for any software maintenance and evolution task. Even when documentation is available, software developers have to understand fine-grained elements of the source code to be able to modify it. These elements relate to visual, structural, and semantic characteristics of the source code of a program, and may hinder program comprehension.

Visual characteristics, such as spacing, impact the *legibility* of the source code and may affect the ability of developers to identify the elements of the code while reading it. Structural and semantic characteristics, such as programming constructs, impact the *readability* of the source code and may affect the ability of developers to understand it while reading the code. We use the term *understandability* to refer to the ease with which developers are able to extract information from a program that is useful for a software development- or maintenance-related task just by reading its source code. Understandability is impacted by legibility and readability.

There exist guidelines and coding standards for different programming languages, such as the Google Java Style Guide[1] for the Java language, which describes good practices and well-accepted conventions on how to write code. However, these guides are built based on the intuition and experience of the developers that elaborate them. In fact, *what makes the code more legible or readable is an open question.* Researchers have conducted empirical studies to compare different but functionally equivalent ways of writing code in terms of their legibility and readability. For instance, Miara et al. (1983) compared different levels of indentation (zero, two, four, and six spaces), and Binkley et al. (2013) investigated the usage of camel case vs. underscore for identifier names. These studies provide insight into the influence of different formatting elements on code legibility, but they are parts of a bigger whole that is still unclear. This paper aims to provide a more comprehensive view of our knowledge about the impact of different formatting elements on code legibility.

[1] `https://google.github.io/styleguide/javaguide.html`

We examine empirical studies performed by researchers with human subjects aiming to find which formatting alternatives are the best for the **legibility** of source code. We conduct a systematic literature review that, from an initial set of 2,843 documents, analyzes 12 papers. We select papers that directly compare alternative formatting elements and styles at the source code level and organize them through a card sorting process. Finally, for each formatting alternative, we examine the findings reported by the primary studies considering two aspects of human-centric studies that evaluate code legibility: the activities performed by human subjects (which relate to the cognitive skills required from them) and the response variables employed by the researchers to collect data from the studies (Oliveira et al., 2020).

We identify 29 formatting elements (e.g., two-space indentation), which were compared with equivalent alternatives (e.g., two-space indentation vs. four-space indentation) in 12 scientific papers. These formatting elements are about formatting styles (e.g., pretty-printed layout vs. disrupted layout), spacing (e.g., indentation levels), block delimiters (e.g., omitted vs. present block delimiters), long or complex code line (e.g., multiple statements per line vs. one statement per line), and word boundary styles (i.e, camel case vs. underscore). For half of the comparisons, researchers found exclusively statistically significant results. For instance, a block delimiter in its own separate line was found to be preferred in comparison to a block delimiter in the same line as the declaration to which it is associated (Arab, 1992; Santos and Gerosa, 2018). For many cases, no statistically significant difference was found, e.g., pretty-printed code layout vs. a disrupted layout (Siegmund et al., 2017). In comparisons involving horizontal spacing, Miara et al. (1983) found out that two indentation spaces lead to code that is easier to understand than four spaces. More recent work (Bauer et al., 2019; Santos and Gerosa, 2018) did not confirm this result, though, and produced inconclusive results. Identifier format is another topic where there are both conclusive and inconclusive results. Binkley et al. (2013) found out that the usage of camel case for identifier names is better than snake case in a study where 135 programmers should answer questions about static properties of the code. However, the same authors did not find a statistical difference between the two formatting alternatives in a second study with 15 programmers who had to perform tasks related to memorizing code elements and explaining the purpose of a program.

To sum up, our contribution is a comprehensive study of what we know about how source code formatting elements and their significance for code legibility. Our results provide fuel for future research on the impact of formatting elements on code legibility. They can also serve as guidelines for developers in choosing ways to format their code. Furthermore, other researchers can benefit from our study by developing automated support, e.g., recommendation systems, to aid developers during programming activities and make their code more legible.

2. Background

In a previous work (Oliveira et al., 2020), we conducted a systematic literature review to define how researchers measure code legibility and readability. To achieve that, we examined primary studies where human subjects are asked to perform programming-related tasks involving comparisons between alternative programming constructs, idioms, and styles. These alternatives are different but functionally equivalent ways of writing code, e.g., recursive vs. iterative code (Benander et al., 1996) and abbreviated vs. word identifier names (Hofmeister et al., 2019). The goal of those comparisons is to find the most legible or readable ways of writing code. We did not, however, investigate the alternatives themselves, which is the goal of the current work.

In this section, we summarize the main three elements of that paper that we build upon. First, we introduce the concepts of code legibility and readability in Section 2.1. We discuss how Software Engineering researchers define these two terms and we bring insights from other areas of knowledge to make a clear distinction between them. In Section 2.2, we provide background on the tasks performed by humans and response variables collected in studies that compare functionally-equivalent source code alternatives, as found by Oliveira et al. (2020). The combination of these tasks and response variables builds a landscape of the different ways in which researchers define and measure code legibility and readability in existing studies. Finally, in Section 2.3, we present how we leverage a learning taxonomy (Bloom et al., 1956; Fuller et al., 2007) to model program comprehension tasks so that researchers can clearly express the cognitive skills they intend to evaluate with a study. More specifically, we review a set of activities that decompose the tasks performed by humans in the primary studies of our previous study (Oliveira et al., 2020).

2.1. Legibility and Readability

In software engineering, the terms readability, legibility, understandability, and comprehensibility have overlapping meanings. For example, Buse and Weimer (2010) define "***readability*** *as a human judgment of how easy a text is to understand*". In a similar vein, Almeida et al. (2003) 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 (2008) 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. (2018) 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 (1986) 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 (1990) 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 (2004) 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 Tekfi (1987), 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. (2015) 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 (DuBay, 2004), design (Strizver, 2013), human-computer interaction (Zuffi et al., 2007), and education (Tekfi, 1987), 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. In this work, we focus mainly on code legibility.

2.2. Tasks and Response Variables Employed in Human-centric Studies on Code Legibility and Readability

In our previous study (Oliveira et al., 2020), we examined how researchers have investigated code legibility and readability by asking human subjects to perform tasks on source code and assessing their understanding or effort with response variables. In this section, we review the tasks and response variables we found during our previous study, which lay a foundation for the study presented in this paper.

Tasks. The essential code comprehension task is code reading. By construction, human-centric studies about code comprehension 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. In addition, the subjects are also expected to comprehend the code. However, there are different (kinds of) tasks that help researchers measure subject comprehension performance.

A large portion of the primary studies requires subjects to **provide information about the code**. For example, Benander et al. (1996) asked the subjects to explain using free-form text what a code snippet does, right after having read it. In some studies, the subjects are asked to answer questions about code characteristics. For example, Gopstein et al. (2017) and Ajami et al. (2019) presented subjects with multiple code snippets and asked them to guess the outputs of these snippets. 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 (1977) asked subjects to memorize a program for three minutes and rewrite the program as accurately as possible in the next four minutes.

Also, some studies required the subjects to **act on the code**. Among these studies, subjects were asked to find and fix bugs in the code. For example, Scanniello and Risi (2013) asked subjects to do so in programs with different identifier styles. In 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 (2014) asked subjects to implement a new feature in a program seen in a previous task. 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 with a comprehension task. For example, Wiese et al. (2019) first asked subjects to write a function that returns true if the input is seven and false otherwise so that they could identify what code style (novice, expert, or mixed) the subjects preferred when coding. Afterward, 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, subjects were also asked to give their **personal opinion**. In some studies, the subjects were inquired about their personal preferences or gut feeling without any additional task. For example, Buse and Weimer (2010) asked them to rate (from 1 to 5) how legible or readable a code snippet is. In other studies, the subjects were asked about their personal opinions while performing other tasks. For example, O'Neal and Edwards (1994) first asked subjects to read a code snippet, then to state if they understood the snippet and provide a description of its functionality. Similarly, Lawrie et al. (2007) 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, subjects were asked to rate the difficulty of the comprehension tasks they had to perform in some studies, e.g., in the study of Fakhoury et al. (2019).

Response variables. After human subjects perform tasks, researchers employ response variables to measure the subjects' performance. Depending on the study's goals, methodology, and subjects, response variables vary. The performance of subjects is measured in some studies in terms of whether they provide correct answers given a task. The **correctness** of answers may pertain to code structure, semantics, use of algorithms, or program behavior. For example, Bauer et al. (2019) measured the subjects' code understanding by asking them to fill in a questionnaire with multiple-choice questions referring to program output.

Studies that evaluate which code alternative is easier to read often collect the subjects' personal opinions. These response variables are grouped in a category called **opinion**. For instance, Santos and Gerosa (2018) presented pairs of functionally equivalent snippets to the subjects and asked them to choose which one they think is more readable or legible.

Some researchers measure the **time** subjects spend performing tasks. There is variety in the way the studies measure time. For Ajami et al. (2019), *"time is measured from displaying the code until the subject presses the button to indicate he is done"*. Hofmeister et al. (2019) computed the time subjects spent looking at specific parts of a program.

More recent studies collect information about the process of performing the task, instead of its outcomes. Multiple studies employ special equipment to track what the subjects see and to monitor the subjects' brain activities during the tasks. The response variables gathered during these processes are **visual metrics** and **brain metrics**. For example, Binkley et al. (2013) computed the visual attention, measured as the amount of time during which a subject is looking at a particular area of the screen. Siegmund et al. (2017) used functional magnetic resonance imaging (fMRI) to measure brain activity by detecting changes associated with blood flow.

2.3. Program Comprehension as a Learning Activity

The empirical studies analyzed in our previous work (Oliveira et al., 2020) involve a wide range of tasks to be performed by their subjects, as revised in the previous section. All these tasks are conducted to evaluate readability and legibility. However, they demand different cognitive skills from the subjects and, as a consequence, evaluate different aspects of readability and legibility. In our previous work, we attempted 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. Following along these lines, we treated the problem of program comprehension (or understanding) as a learning problem. We proposed an adaptation of the learning taxonomy devised by Fuller et al. (2007) to the context of program comprehension. Fuller et al. (2007)'s taxonomy is composed of a set of activities that build upon Bloom's revised taxonomy (Anderson et al., 2001) (for educational objectives) and a model that emphasizes that some activities in software development involve acting on knowledge, instead of just learning. Our central idea was to leverage the elements defined by the taxonomy of Fuller et al. (2007), with some adaptions, to identify and name the cognitive skills required by different tasks employed by code readability and legibility studies. The resulting activities

Table 1: Learning activities extended from Fuller et al. (2007)—Inspect and Memorize are not in the original taxonomy. Opinion is not included because it is not directly related to learning. The definitions, except where not applicable, are taken directly from Fuller et al. (2007). The examples are either extracted from tasks from our primary studies or general ones when no task involved that activity.

Activity	Description and example
Adapt	Modify a solution for other domains/ranges. This activity is about the modification of a solution to fit in a given context. Example: Remove preprocessor directives to reduce variability in a family of systems (Schulze et al., 2013).
Analyze	Probe the [time] complexity of a solution. Example: 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. Example: Reuse of off-the-shelf components.
Debug	Both detect and correct flaws in a design. Example: Given a program, identify faults and fix them (Scanniello and Risi, 2013).
Design	Devise a solution structure. The input to this activity is a problem specification. Example: Given a problem specification, devise a solution satisfying that specification.
Implement	Put into lowest level, as in coding a solution, given a completed design. Example: Write code using the given examples according to a specification (Stefik and Siebert, 2013).
Model	Illustrate or create an abstraction of a solution. The input is a design. Example: Given a solution, construct a UML model representing it.
Present	Explain a solution to others. Example: Read a program and then write a description of what it does and how (Chaudhary and Sahasrabuddhe, 1980).
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. Example: "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" (Tenny, 1988).
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. Example: 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. Example: Choose one out of three high-level descriptions that best describe the function of a previously studied application (Blinman and Cockburn, 2005).
Trace	Desk-check a solution. Simulate program execution while looking at its code. Example: Consider the fragment "x=++y": what is the final value of x if y is -10? (Dolado et al., 2003).
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. Example: "All variables in this program are global." [true/false] (Miara et al., 1983).
Memorize*	Memorize the code in order to reconstruct it later, partially or as a whole. Example: Given a program, memorize it in 3 minutes and then reconstruct it in 4 (Love, 1977).

are described with examples in Table 1. We introduced two activities (marked with "*") that stem directly from tasks performed by subjects in some of the primary studies we analyzed and require skills that are not covered by the original set of activities.

We analyzed the tasks that subjects performed in the primary studies and identified which activities from Fuller et al. (2007) they require. Table 2 presents the mapping of tasks (rows) to the learning activities (columns).

Table 2: Mapping of tasks (rows) to learning activities (columns), adapted from Oliveira et al. (2020).

	Adapt	Debug	Implement	Inspect*	Memorize*	Present	Recognize	Relate	Trace	Opinion**
provide information about the code										
explain what the code does						✓		✓		
answer questions about code				✓			✓	✓	✓	
remember (part of) the code			✓		✓	✓				
act on the code										
find and fix bugs in the code		✓								
modify the code	✓		✓	✓					✓	
write code			✓	✓						
provide personal opinion										
opinion about the code										✓
answer if understood the code										✓
rate confidence in her answer						✓			✓	✓
rate the task difficulty		✓	✓	✓		✓			✓	✓

Moreover, besides the activities in Table 1, we also considered Giving an Opinion (hereafter, "Opinion"), which is not part of the taxonomy because it is not a learning activity. The table 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 Giving an Opinion activity. In addition, some tasks may be connected to various activities. For instance, "Modify the code" may require subjects to Implement, Trace, Inspect, or Adapt the code to be modified. This makes sense; to modify 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 some studies (O'Neal and Edwards, 1994; Blinman and Cockburn, 2005) 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 non-intuitive relationships between tasks and activities. Chaudhary and Sahasrabuddhe (1980) asked the subjects to reconstruct a program after spending some time looking at it. This is a "Remember the code" task where the subjects have to explain what the program does by writing a similar program. It is a mix of Present and Implement. In another

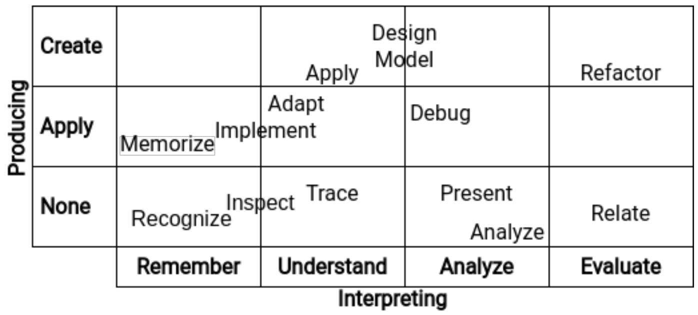

Figure 1: Learning activities model (Oliveira et al., 2020).

example, Lawrie et al. (2007) asked the subjects to describe what a code snippet does and rate their confidence in their answers. Rating one's confidence in an answer is about Opinion but it also must be associated with some other activity, since some answer must have been given in the first place. In this case, the activity is to Present the code snippet, besides giving an Opinion. A similar phenomenon can be observed in studies where subjects were asked to "Rate the task difficulty" they had performed, e.g., (Jbara and Feitelson, 2017) and (Fakhoury et al., 2019).

Besides the list of activities, Fuller et al. (2007) proposed a model adapted from the revised version of Bloom's taxonomy (Anderson et al., 2001). 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. Producing has three levels (None, Apply, and Create) and Interpreting has four (Remember, Understand, Analyze, and Evaluate). Figure 1 represents this two-dimensional model. According to Fuller et al. (2007), a level appearing more to the right of the figure along the Interpreting dimension (x-axis), 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 (y-axis).

3. Methodology

In this paper, we aim to examine what formatting elements have been investigated, and which ones were found to be more legible, in human-centric studies. We focus on studies that directly compare two or more functionally equivalent alternative ways of writing code. We address two research questions in this paper:

RQ1 What formatting elements at the source code level have been investigated in human-centric studies?

RQ2 Considering the subjects, tasks, and response variables in human-centric studies, which elements have been found to be more legible?

To answer our research questions, we conduct a systematic literature review, designed following the guidelines proposed by Kitchenham et al. (2015). Our methodology is similar to the one employed in our previous work (Oliveira et al., 2020)—the differences are explained in Section 3.6. Figure 2 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 use as seed papers so that a search string could be defined, and automatic search could be performed on search engines (Section 3.1). We retrieved 2,843 documents with the automatic search, which passed through a triage for study exclusion (Section 3.2), an initial study selection where inclusion criteria were applied (Section 3.3), and a final study selection where we evaluated the quality of the studies based on several criteria (Section 3.4). Then, the selected 12 papers were analyzed (Section 3.5). 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[2], because we were not aware of them at the time.

3.1. 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 between 2016 to June 2019 (the search was conducted in the second half of 2019), and selected the ones that would help us answer our

[2]`https://parsif.al/`

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 used the resulting terms to build the following search string:

$$\mathit{Title}(\mathit{ANY}(\mathit{terms}))\ \mathit{OR}\ \mathit{Keywords}(\mathit{ANY}(\mathit{terms})),$$
where $\mathit{terms} = \{$ "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 word in the title or keywords are related to linguistics or computational linguistics. In these fields, researchers 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[3], IEEE Explore[4], and Scopus[5]. 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.

3.2. 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 of writing code, neither direct nor indirect, or is clearly irrelevant to our research questions. For instance,

[3]`http://dl.acm.org/`
[4]`http://ieeexplore.ieee.org/`
[5]`http://www.scopus.com/`

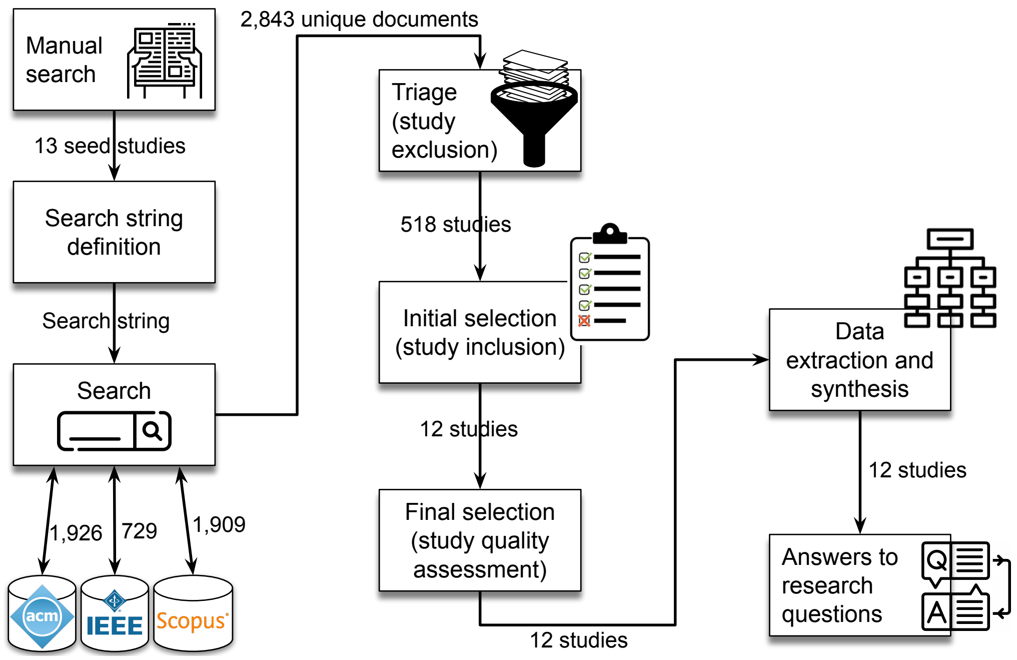

Figure 2: Systematic literature review roadmap.

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 (Kitchenham et al., 2015). As a rule of thumb, we consider that full papers must be at least 5 pages long.

EC3. The study is about legibility or 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. 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. At 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. We ended up with 518 documents.

3.3. Initial Selection (Study Inclusion)

After discarding clearly irrelevant documents in the triage step, we applied the following inclusion criteria to the 518 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 a controlled experiment, quasi-experiment, or survey *involving human subjects.*

IC3 (Comparison). The study must directly compare alternative formatting elements *in terms of code legibility* and the alternatives must be clearly identifiable.

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 (Cohen, 1960) for assessing the agreement between the two analyzes. We found

$k = 0.323$, which is considered a fair agreement strength (Kitchenham et al., 2015). At 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 all authors in live sessions to reach consensus. We ended up with 12 papers.

3.4. Study Quality Assessment

The selected 12 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 et al. (2007). 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 is 4.5 and no paper scored less than 5, no studies were removed because of bad quality.

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 paper of Binkley et al. (2008) was extended in a subsequent paper (Binkley et al., 2009). In this case, we consider the former to be deprecated and only take the latter into account.

3.5. Data Analysis

To answer the research questions, we analyzed a total of 12 papers. The analysis was composed of data extraction and synthesis.

Data extraction. Initially, we read all the papers in full and extracted the necessary data to answer our research questions. For this extraction, the papers were equally divided among the authors, and periodic meetings were held for discussion. For RQ1, we extracted the independent variables of the studies, which are the alternative formatting elements being compared. For RQ2, for each compared formatting element, we collected 1) the tasks the human subjects were required to perform in the experiment, which we mapped to learning activities (see Section 2.3), 2) the dependent variables, which are the response variables (see Section 2.2), 3) the results and statistical analysis, and 4) the characteristics of the human subjects.

Synthesis Process. To better understand and present the results in an organized way, we performed card sorting (Wood and Wood, 2008) on the examined alternative formatting elements. With this method, we transformed the formatting elements (topics) into cards and then grouped the cards based on their similarities. This process was bottom-up, i.e., alternatives (topics) $\rightarrow$ groups $\rightarrow$ category, and we present it this way as follows.

Formatting Elements (topics). We used the formatting elements extracted from the studies as topics in the card sorting process. Then, we created a card for each topic naming them in a comparison format, i.e., "a code alternative vs another code alternative", except when the topic compared more than two alternatives. For synonymous topics, we created one unique card to represent them. For instance, Santos and Gerosa (2018) evaluated whether using a block delimiter in the same line of their statements is better than the block having its own line. Similarly, Arab (1992) compared different presentation schemes (Peterson, Crider, and Gustafson) that define where the block delimiter should be placed. We created a unique card named "Block delimiter in the statement line vs Block delimiter in separate line" for these topics.

Groups and categories. The next step was to group similar cards. In live sessions, we discussed each card and included it in a representative group. When there was no representative group for a card, a new group was created. For example, the first card we analyzed was "Paragraphed vs Unparagraphed". We created the first group, named "Group 01", and included that card in it. The second card we analyzed was "Pretty-printed layout vs. Disrupted layout". Such a card is not similar to the card in "Group 1", so we created the second group named "Group 02" for it. The third card we analyzed was "Indentation level", which is similar to the card in "Group 01", so we included it in that group as well. After including all cards in some group, we gave meaningful names to each group. We also split groups that seemed too generic into smaller ones. In cases where two groups were very similar and small, we combined them.

The card sorting process was initially performed by the first two authors and later the results were refined with the collaboration of all authors in multiple live sessions.

3.6. Differences between this study and the previous one

This work is part of a bigger research project where we investigate coding idioms and constructs that affect code legibility and readability, what are they, which ones are the best, and how researchers and developers evaluate what "best" means. Thus, we reuse in this work the methodology we employed in our previous paper (Oliveira et al., 2020). We dedicate this section to explaining the differences between the two works.

In the previous paper, our goal was to examine and classify the tasks performed by human subjects in experiments and response variables employed by researchers to assess code legibility and readability. This paper builds upon our previous work, so much so that Section 2 presents a summary of that paper, and aims to investigate formatting elements that were compared in the literature and which ones are more legible. Because the goals of the two studies are different, the research questions are different. The data analysis (explained in Section 3.5) is also different.

The remainder of the methodology, which deals with article selection (i.e., the search strategy, study exclusion, study inclusion, and study quality assessment), was inherited from our previous work. However, some studies that were selected in our previous work are not adequate to answer the research questions of this study. For example, Buse and Weimer (2008) performed a study involving humans to predict readability judgments based on code features and human notions of readability. This study focuses on code readability, moreover, these coding alternatives were indirectly compared in their work. As a consequence, we are unable to determine which code alternatives were involved in the comparisons. Because of this, we changed an inclusion criterion from our previous methodology (IC3) to select only studies on legibility that directly compare formatting elements or styles (a group comprising multiple formatting elements), and the alternatives can be clearly identified.

Therefore, several studies that were examined in our previous work, for example (Buse and Weimer, 2008; Trockman et al., 2018; Scalabrino et al., 2018, 2019; Posnett et al., 2011; O'Neal and Edwards, 1994; Kasto and Whalley, 2013; Jbara and Feitelson, 2017) were not examined in this work, because we cannot use them to answer our current research questions.

3.7. 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`.

4. Results

In this section, we present the results of the study. We organize the subsections by categories of formatting elements being compared. In each section, we address both research questions. The first research question aims to identify the formatting elements that the researchers considered in their studies about code readability and legibility. These elements are organized into categories, subcategories, and groups, considering the similarities between them. Table 3 presents the results of this categorization. With the second research question, we aim to synthesize the studies' findings related to legibility by comparing the formatting elements. For that, we consider data such as the studies' answers, statistical tests, activities performed by humans, dependent variables, and the context of the human sample.

Legibility pertains to characteristics of the code that make it easier or harder to identify its elements. Although code readability is studied more often than code legibility, we identified multiple studies focusing on the latter. These studies are organized in five groups: Formatting styles (Section 4.1), spacing (Section 4.2), block delimiters (Section 4.3), long or complex code line (Section 4.4), and word boundary styles (Section 4.5).

4.1. Formatting Styles

This group gathers studies about different ways of formatting code, from a global, high-level perspective. The paper of Siegmund et al. (2017), aims to compare a pretty-printed code layout with a disrupted layout. More specifically, this paper compares the code comprehension of subjects considering

Table 3: Card Sorting results.

Groups	Compared alternatives
Formatting Styles	Lightspeed Pascal-style formatting vs. book format style
	Kernighan and Ritchie style vs. book format style
	Pretty-printed layout vs. disrupted layout
Spacing	Indentation level
	Paragraphed vs. unparagraphed
	Blank lines must be used to create a vertical separation between related instructions
Block Delimiters	Block delimiter in the statement line vs. block delimiter in a separate line
	Omitted vs. present block delimiters
	Use ENDIF/ENDWHILE (without BEGIN) vs. BEGIN-END in all blocks
	Use ENDIF/ENDWHILE (without BEGIN) vs. BEGIN-END only compound statements blocks
	Indentation blocked vs non-blocked
Long or Complex Code Line	Line lengths must be kept within the limit of 80 characters
	Multiple statements per line vs. one statement per line
Word Boundary Styles	Camelcase vs. underscore

code snippets following common coding conventions for layout, e.g., indentation, line breaks, correctly-placed scope delimiters, etc., and a code snippet with defective layout, e.g., line breaks in the middle of an expression or irregular use of indentation. An interesting differentiating aspect of this study is that it analyzes bottom-up program comprehension and the impact of beacons (Brooks, 1978) and pretty-printed layout using functional magnetic resonance imaging (fMRI). Eleven students and professionals took part in this study. First, these subjects participated in a training session in which they studied code snippets in Java including semantic cues to gain familiarity with them. Then, once in the fMRI scanner, the participants looked at other small code snippets to determine whether they implemented the same functionality as one of the snippets in the training session with a time limit of 30 seconds for each snippet (Relate). In this scanner, the level of blood oxygenation in various areas of the brain was measured. This process is called BOLD (Blood Oxygenation Level Dependent) (Chance et al., 1993). That metric is a proxy to assess the activation of the area of the brain. To evaluate the role of beacons and layout on comprehension based on semantic cues, they created four versions of the semantic-cues snippets: (i) beacons and pretty-printed layout, (ii) beacons and disrupted layout, (iii) no beacons and pretty-printed layout, (iv) no beacons and disrupted layout. Finally, they extracted the Random-effects Generalized Linear Model (GLM) beta values for each participant and condition to identify differences in brain activation for each program comprehension condition. For that evaluation, we considered the study with code snippets in which there are no beacons to avoid the effect of another variable, specifically we avoid beacons because they are not related to formatting elements. The authors did not find significant differences in activation values in the brain areas when comparing code versions with pretty-printed layout and disrupted layout.

Oman and Cook (1990) propose the notion of Book Format Style for structuring source code and compare it to two well-known styles for the Pascal and C languages: Lightspeed Pascal style (Johnson and Beekman, 1988) and Kernighan & Ritchie style (Ritchie et al., 1988). Book Format style, as the name implies, takes inspiration from how books are structured as an approach to organize source code. In this format, programs include a preface, table of contents, chapter divisions, pagination, code paragraphs, sentence structures, intramodule comments, among other elements. For the comparison with Lightspeed Pascal, the authors asked 36 students to answer fourteen multiple choice, short answer questions about the characteristics of the code snippet in a total of ten minutes (both Trace) and to provide a subjective opinion about the understandability of the code snippet (Giving an opinion). The subjects were randomly assigned to two treatment groups, one for each formatting alternative, and both received the same instructions. Then, they assessed (i) how many questions the students answered correctly (score of 1 to 14 points), (ii) the time to answer the questions (1 to 10 minutes), (iii) the performance score (number of correct answers per minute), and (iv) the subjective opinion (rating on a five-point). This study concluded that the Book Format element is considered a better formatting style compared to Lightspeed Pascal based on the significant differences for correctness score (ANOVA, $p < 0.005$), performance (ANOVA, $p < 0.01$), and subjective understandability opinion (ANOVA, $p < 0.05$), but not for time. In the comparison between the Book Format style and the Kerningham & Ritchie style, they prepared a very similar experiment with 44 different students. The only difference is that the questionnaire had ten questions instead of the fourteen from the previous experiment. They assessed the same response variables and found out that Book Format was statistically better for score (ANOVA, $p < 0.005$) and performance (ANOVA, $p < 0.005$), but there are no significant differences for time and subjective opinion. Table 4 presents a summary of the results. The table includes dependent variables used in each study, the corresponding learning activities (Section 2.3) performed by the

Table 4: Summary of results for the Formatting Styles group.

Comparison -Study	Dependent variables (Activities)	Results
Lightspeed Pascal-style formatting vs. Book Format style		
(Oman and Cook, 1990)	Correctness (Trace), Time (Trace) and Opinion	The book format style is the best for Correctness and Opinion, but the results for Time do not differ.
Kernighan and Ritchie style vs. Book Format style		
(Oman and Cook, 1990)	Correctness (Trace), Time (Trace) and Opinion	Book format style is better in Correctness, but in Time and Opinion no differences were found.
Pretty-printed layout vs. Disrupted layout		
(Siegmund et al., 2017)	Brain Metrics (Relate)	There is no significant difference.

subjects, and the main results found for each comparison.

4.2. Spacing

This group assembles studies that investigate different types of spacing in source code. Miara et al. (1983) conducted an experiment with 54 novice students and 32 experienced students to evaluate the influence of indentation levels and blocked code on program comprehension. In this experiment, the authors used the following independent variables: level of indentation (zero, two, four, and six spaces), level of experience (novice and expert), and method of block indentation (nonblocked and blocked). In the experiment, each subject received a program in Pascal using one of the levels of indentation and one of the methods of block indentation; in addition, a quiz with ten questions required subjects to (i) select the correct answer about code characteristics (Trace and Inspect), (ii) explain what the code does (Present), and (iii) to provide an opinion on the difficulty of the task (Giving an opinion). They considered two dependent variables: how many questions were correct (score between 1-10) and the rating of the subjective opinion. The results showed that there are significant differences between novices and experts (ANOVA, $p < 0.001$) and between indentation levels (ANOVA, $p = 0.013$), where two characters produced the best results.

Similarly, Bauer et al. (2019) conducted an experiment with 7 developers and 15 students to investigate the influence of the levels of indentation (zero, two, four, and eight spaces) on code comprehension. They designed an experiment similar to the experiment of Miara et al. (1983), with the differences that they asked for the output of snippets in an open box instead of providing multiple choices, and did not ask for a description of code functionality. They have also assessed code comprehension by (i) the exact correct answer

Table 5: Summary of results for the Spacing group.

Comparison -Study	Dependent variables (Activities)	Results
Indentation level		
(Miara et al., 1983)	Correctness (Trace, Inspect, Present) and Opinion	Two level of indentation was the best.
(Bauer et al., 2019)	Correctness (Trace), Time (Trace), Visual Metrics (Trace), Opinion	There is no significant difference.
(Santos and Gerosa, 2018)	Opinion	There is no significant difference.
Paragraphed vs Unparagraphed		
(Love, 1977)	Correctness (Memorize and Present)	There is no significant difference.
Blank lines must be used to create a vertical separation between related instructions		
(Santos and Gerosa, 2018)	Opinion	There is no significant difference.

of snippet output (Trace) and (ii) rank of perception of task difficulty (Giving an opinion). Differently from the work of Miara et al. (1983), this experiment includes eye-tracking to measure visual effort using fixations, which occur when the gaze is resting on a point, and saccades, which is the transition between two fixations. Furthermore, it measures the time required to provide an answer. The statistical analysis first applied Mauchly's sphericity test to evaluate which test was more adequate for each dependent variable, One-way ANOVA with repeated measures or Friedman's test. The results of this experiment were: correctness of answers (Friedman test, $p = 0.36$), the log-transformed response time (ANOVA, $p = 0.72$), perceived difficult (Friedman test, $p = 0.15$), fixation duration (ANOVA, $p = 0.045$), fixation rate (Fridman test, $p = 0.06$), saccadic amplitude (ANOVA, $p = 0.18$). Despite the $p < 0.05$ for fixation duration, they could not confirm this difference with a post hoc test. Therefore, there was not a statistically significant difference between indentation levels, unlike in the previous study.

Santos and Gerosa (2018) performed a survey with 55 students and 7 professionals to investigate the impact of a set of Java coding practices on code understandability. Specifically, among other practices, subjects had to choose (Giving an opinion) between two alternatives for indentation level: two and four spaces. They evaluated the results by comparing the proportions of the votes to each alternative. The results showed that there is no statistical difference (Two-tailed test for proportion, $p = 0.32$) between these alternatives.

Another coding practice analyzed by Santos and Gerosa (2018) pertains to the use of vertical spaces. More specifically, the authors asked the subjects (by presenting code examples) whether blank lines must be used to create a vertical separation between related instructions (Giving an opinion). The results also did not show a statistically significant difference (Two-tailed test for proportion, $p = 1.0$) between subjects who agree and disagree with this practice.

Still in the **Spacing** group, the paper by Love (1977) evaluates the use of paragraphed vs. unparagraphed code, i.e., the disciplined use of vertical and horizontal spacing to organize the code in text-like paragraphs. The author experimented with students (19 undergraduate and 12 graduate) to investigate the effect of program indentation (paragraphed or unparagraphed) and control flow (simple or complex) on code comprehension. The subjects were asked to memorize a program in Pascal in 3 minutes, rewrite the program in 4 minutes (Memorize), and describe the program functionality (Present). The reconstructed programs were scored based on the percentage of lines of source code recalled correctly in the proper order. The descriptions of the program functionality were rated with a 5-point scale, where one point means that the description has nothing right about the program functionality and five points means that the description shows a complete understanding of it. The author did not find a statistically significant difference between paragraphed and unparagraphed programs, considering Memorize (ANOVA, $p = 0.79$) and Present (ANOVA, $p = 0.6$) for both undergraduate and graduate students.

Table 5 presents a summary of the analyzed results about spacing.

4.3. Block Delimiters

This group gathers papers that analyze different types of block/scope delimiters. Arab (1992) compares the use of block delimiters in the same line as the associated statement, e.g., a loop or a conditional, vs. block delimiters in a separate line. This study consisted of an opinion survey with 30 subjects (22 novices and 8 experts) in which they evaluated three different code presentation schemes in Pascal that propose different patterns to place the block delimiters: (i) Peterson's scheme: BEGIN and END in their own lines (Peterson, 1977), (ii) Crider's scheme: BEGIN and END in the same line of the last statement or declaration (Crider, 1978), and (iii) Gustafson's scheme: BEGIN in the same line as the last statement but END in its own line (Gustafson, 1979). The subjects were asked to classify the three schemes in descending order (Giving an opinion). The results show that the block delimiter in its own line pattern, i.e., Peterson's scheme, was chosen as the best by most participants. However, they did not apply any statistical test to evaluate the results. In a similar vein, Santos and Gerosa (2018) asked subjects whether they prefer to have Java's opening curly braces in their own lines or in the same line as the corresponding class, method, block, etc. They found a statistically significant difference (Two-tailed test for proportion, $p = 0.006$) between the two groups, where most participants consider that opening braces in their own lines are easier to read, similar to results of Arab (1992).

Gopstein et al. (2017) compare the use of omitted and present block delimiters, in the specific context of small C programs. The authors conducted a broad-scoped experiment[6] with 73 programmers with 3 or more months of experience in C/C++ to evaluate small and isolated patterns in C code, named atoms of confusion, that may lead programmers to misunderstand code. They asked subjects to analyze tiny programs ($\sim$8 lines) where the control version contained a single atom of confusion candidate and the treatment version is a functionally-equivalent version where the atom is removed. For each program, the participants had to predict the output of the program (Trace). The authors compared the correctness of the answers of the version with the atom of confusion and their counterparts without the atoms. One was these atoms is called *Omitted Curly Braces*. In C programs, this is used when the body of an `if`, `while`, or `for` statement consists of a single statement. The study found a statistically significant difference (McNemar's test adjusted using Durkalski correction, $p < 0.05$) between the programs where the curly braces were omitted and the ones where they were not. Subjects analyzing programs with omitted curly braces committed more mistakes.

Similarly, the paper of Medeiros et al. (2019) investigates what they call misunderstanding code patterns, which are very similar to atoms of confusion. One study was a survey with 97 developers, where the subjects should determine how negative or positive is the impact of using one code snippet version including one misunderstanding pattern instead of a functionally-equivalent alternative without that pattern on a 5-point Likert scale (Giving an opinion). In another study, they submitted 35 pull requests suggesting developers to remove several misunderstanding patterns. In this section, we analyze the results of the misunderstanding pattern named Dangling Else that happens

[6]Arguably, it was an Internet-based survey, but with enough controls in place that we feel it is more appropriate to call it an experiment.

when an if statement without braces contains one if/else statement without braces. The developer needs to know that the else clause belongs to the innermost `if` statement. The first study found that Dangling Else is perceived as negative by 71.73% of the developers. In the second study, they submitted 10 pull requests to remove the Dangling Else pattern, where 2 of them were accepted, 1 was rejected, 3 were not answered, and 4 ignored by developers.

The work of Sykes et al. (1983) evaluates different styles of scope delimiters: (i) ENDIF, which uses `WHILE-ENDWHILE` and `IF-ENDIF` scope delimiters for the bodies of conditional and iterative statements, respectively; (ii) REQ-BE, which always uses `BEGIN-END` scope delimiters for single line and multiple line blocks; and (iii) BE, which uses `BEGIN-END` scope delimiters only for multiple line blocks. The study employed a questionnaire where most of the questions asked subjects to determine values of variables (Trace) after execution of program segments in Pascal with initial values given in the questions. In total, 36 students participated in that study, and they had 25 minutes to finish the questionnaire. The subjects were divided into advanced (at least 2 years of programming experience) and intermediate. They evaluated the responses to the questionnaire considering the type of delimiter and the experience of the subjects. ANOVA indicated that experience was significant ($p = 0.005$), and advanced subjects did much better than intermediate ones. Also, a paired t-test (considering an $\alpha = 0.1$) has shown that the subjects performed better using the ENDIF delimiters than using the REQ-BE ($p = 0.074$) and BE ($p = 0.073$) delimiters. The authors did not find a difference between the REQ-BE and BE (t-test, $p = 0.955$).

Miara et al. (1983) also evaluated the difference between two styles of block indentation: (i) non-blocked, where the code within the block appears one more indentation level to the right of the block delimiters, and (ii) blocked, where the code within a block is at the same level as the block delimiters. This comparison is different from what has been previously reported about indentation (Section 4.2) because this one evaluates when the indentation starts and where the block delimiters stay, while the other one focuses solely on the indentation level. The design of this study was explained in Section 4.2. The analysis of variance (ANOVA) of the quiz scores and the program ratings showed no significant effect with the non-blocked and blocked styles.

Table 6 presents a summary of the analyzed results for block delimiters.

Table 6: Summary of results for the Block Delimiters group.

Comparison -Study	Dependent variables (Activities)	Results
Block delimiter in the statement line vs Block delimiter in a separate line		
(Arab, 1992)	Opinion	Block delimiter in its own line is the best.
(Santos and Gerosa, 2018)	Opinion	Block delimiter in its own line is the best.
Omitted vs Present Block Delimiters		
(Gopstein et al., 2017)	Correctness (Trace)	Present block delimiters is the best.
(Medeiros et al., 2019)	Opinion	Present block delimiters is the best.
(Sykes et al., 1983)	Correctness (Trace), Opinion	There is no significant difference.
Use ENDIF/ENDWHILE (without BEGIN) vs Use BEGIN-END in all blocks		
(Sykes et al., 1983)	Correctness (Trace) Opinion	Use ENDIF/ENDWHILE is the best.
Use ENDIF/ENDWHILE (without BEGIN) vs Use BEGIN-END only compound statements blocks		
(Sykes et al., 1983)	Correctness (Trace) Opinion	Use ENDIF/ENDWHILE is the best.
Indentation Blocked vs non-blocked		
(Miara et al., 1983)	Correctness (Trace, Inspect, Present), Opinion	There is no significant difference.

4.4. Long or Complex Code Line

This group comprises studies that evaluate elements related to the size and complexity of code lines. More specifically, it covers two topics: whether lines are longer than 80 characters and whether they should include multiple statements. The first topic is analyzed in the study of Santos and Gerosa (2018). In this study, they found out that students and professionals find code snippets where every line consists of up to 80 characters more legible than ones where there are longer lines (Two-tailed test for proportion, $p < 0.001$).

Two studies tackle the question of whether lines of code including a single statement are more legible than ones with multiple statements. Santos and Gerosa (2018) found out that students and professionals prefer code snippets where lines of code do not include multiple statements (Two-tailed test for proportion, $p < 0.001$). As discussed before, the paper of Medeiros et al. (2019) (Section 4.3) introduces and evaluates what the authors call misunderstanding code patterns. Among the studied patterns, one specifically refers to a scenario where multiple variables are initialized on the same line. The study asked subjects their opinion about the use of this pattern (Giving an opinion). The authors found out that the use of *multiple initializations on the same line* was neither negative nor positive for most of the subjects. In the second study, they submitted 1 pull request to remove this pattern, which was rejected by developers.

Table 7: Summary of results for the Long or Complex Code Line group.

Comparison -Study	Dependent variables (Activities)	Results
Line lengths must be kept within the limit of 80 characters		
(Santos and Gerosa, 2018)	Opinion	The practice line lengths must be kept within the limit of 80 is the best.
Multiple statements per line vs. one statement per line		
(Medeiros et al., 2019)	Opinion	One statement per line is the best.
(Santos and Gerosa, 2018)	Opinion	There is no significant difference (neither negative nor positive).

Table 7 presents a summary of the analyzed results for block delimiters.

4.5. Word Boundary Styles

The fifth and last Legibility group we have identified comprises studies of **Word boundary styles** for identifiers. This group contains only one comparison, between the use of camel case and underscores to separate words the words in an identifier. Binkley et al. (2013) have performed five studies to evaluate the effect of identifier style on code comprehension. Only three are related to source code: Where's Waldo study, Eye Tracker Code study, and Read Aloud study. In the first study, 135 programmers and non-programmers attempted to find all occurrences of a particular identifier (Inspect) in code fragments written in C and Java. The authors measured the number of lines where the subjects misread occurrences of the identifier and the time spent on that task for the subjects that found all identifier occurrences. Using a Linear mixed-effects regression, they found a significant difference ($p = 0.0293$) in favor of camel case, to an estimated 0.24 fewer missed lines than underscore. For the time, it was marginally significant ($p = 0.0692$) and indicates that camel case takes on average 1.2 fewer seconds.

In the second study, they asked 15 programmers (undergraduate and graduate students) to study two C++ code snippets to reproduce them and answer questions that ask them to select the identifiers they remember that are in the code (Memorize). The authors used an eye tracker to measure eye fixations and gaze duration, the duration of a fixation within a specific area of interest (boxes around the identifiers with some extra padding). Also, the study assessed the correctness of the answers. The authors compared differences in visual effort using two pairs of similar identifiers, *row_sum* and *colSum*, and *c_sum* and *rSum*. They found out that *row_sum* requires significantly more fixations than *colSum* (1-tailed, $p = 0.007$), and that its average gaze duration was 704 ms longer than that of *colSum* (1-tailed, $p =$

Table 8: Summary of results for the Word Boundary Styles group.

Comparison -Study	Dependent variables (Activities)	Results
Camelcase vs Underscore		
(Binkley et al., 2013)	Correctness (Inspect, Memorize, Present), Time (Inspect, Present), Visual Metrics (Memorize)	Camelcase is the best except in the cases of: Correctness (Memorize and Present) and Time (Present)

0.005). No significant differences were found between *c_sum* and *rSum*. Furthermore, they did not find a statistical difference (Linear mixed-effects regression, $p = 0.451$) for correctness.

Finally, the third study asked 19 programmers to summarize (i.e., explain each step of the code) a Java code snippet (Present) verbally. The subjects were a subset of the Where's waldo study group in at least their second year of university. The authors collected the amount of time that each subject spent reviewing the code before summarizing it and the correctness of the answer, which was assessed on a 10-point Likert scale by the authors. They did not find a statistical difference between camel case and underscore for time (Linear mixed-effects regression, $p = 0.6129$) or correctness (Linear mixed-effects regression, $p = 0.3048$).

Table 8 presents a summary of the analyzed results for block delimiters.

5. Discussion

In this section, we discuss aspects of this study that pertain to more than one of the investigated papers. We also highlight gaps we have identified in the literature and potential directions for future work.

5.1. Addressing the two research questions.

Section 3 presents the two research questions that this work aims to answer:

RQ1 What formatting elements at the source code level have been investigated in human-centric studies?

RQ2 Considering the subjects, tasks, and response variables in human-centric studies, which elements have been found to be more legible?

Based on the results of our study, for **RQ1** we have identified 12 scientific papers in which researchers compared alternative formatting elements with human subjects. We found 29 formatting elements, which are about code formatting styles (e.g., pretty-printed layout vs. disrupted layout), code spacing (e.g., paragraphed vs unparagraphed), block delimiters (e.g., omitted vs present block delimiters), long or complex code line (e.g., multiple statements per line vs. one statement per line), and word boundary styles (e.g., camel case vs underscore).

The 29 formatting elements (e.g., two-space indentation) were compared with equivalent ones (e.g., two-space indentation vs. four-space indentation), comprising 14 comparisons (see Table 3). For **RQ2**, out of the 14 comparisons, seven had exclusively statistically significant results. For these comparisons, the best alternatives were book format style (cf. Lightspeed Pascal-style formatting and Kernighan and Ritchie style), the placement of a block delimiter in its own line (cf. block delimiter in the same line as the declaration), the usage of ENDIF/ENDWHILE for block delimiters (cf. use BEGIN-END in all blocks and use BEGIN-END only for compound statement blocks), the camel case convention for identifier names (cf. snake case), and the practice that line lengths must be kept within the limit of 80 (cf. line lengths exceeding the 80 character limit). Four of the comparisons did not exhibit statistically significant results (e.g., Pretty-printed layout vs. Disrupted layout and Paragraphed vs Unparagraphed). In addition, three comparisons showed divergent results between different studies, e.g., indentation levels.

5.2. Contrasting empirical results with existing style guides

In this section, we summarize the main results of the study for each of the five groups. Since our ultimate goal is to devise a coding style guide based on empirical evidence, we examine the results in the light of four existing style guides for Java, JavaScript, Python, and C and contrast their recommendations against the findings of this study.

Formatting Styles. We found three comparisons within this group, but only two of them showed statistically significant results. In particular, the Book format style (Oman and Cook, 1990) exhibited better results compared to the Lightspeed Pascal and Kernighan & Ritchie styles. However, these results are from a study that used Pascal, not a popular programming language in 2022[7][8], and at the time this comparison was performed, formatting tools were still scarce. Such results may not be applicable in our current context because considerable advances have been made in programming languages and tools. Furthermore, no existing code style guide supports the use of the Book format style.

Spacing. Most studies that evaluated spacing did not produce statistically significant results. Overall, the results indicate that the disciplined use of vertical and horizontal spacing to organize code into text-like paragraphs is not relevant. We hypothesize that this may stem from a lack of statistical power: different spacing approaches are not likely to yield big effect sizes and the power required to detect such small effect sizes implies large sample sizes (Section 5.3). Two studies (Bauer et al., 2019; Santos and Gerosa, 2018) did not find a significant difference between different indentation levels, e.g., two vs. four spaces. An older study (Miara et al., 1983) comparing zero, two, four, and six spaces found that two spaces exhibited the best result. This is consistent with some coding guides such as Google Java Style Guide[9], the AirBNB JavaScript Style Guide[10], and the Style Guide for Python Code[11]. Others, such as the Linux Kernel Coding Style[12] recommend eight spaces and explicitly argue against two or four spaces. Currently, the developer community uses spacing patterns, both vertical and horizontal, because they believe in their usefulness.

Block Delimiters. The majority of the studies that evaluated block delimiters found statistically significant results. They have evaluated the position, visibility, and naming of block delimiters. The studies we investigated suggest that block delimiters are relevant to legibility and should be visible and stay in their own line. The Linux Kernel Coding Style agrees with this for functions but disagrees for statements. The AirBNB JavaScript Style Guide [13] prescribes that the opening brace of a statement or declaration should be placed in the same line as the statement or declaration and not in a separate

[7]https://www.tiobe.com/tiobe-index/

[8]https://redmonk.com/sogrady/2022/03/28/language-rankings-1-22/

[9]`https://google.github.io/styleguide/javaguide.html`

[10]`https://github.com/airbnb/javascript`

[11]`https://peps.python.org/pep-0008/`

[12]`https://www.kernel.org/doc/html/v4.10/process/coding-style.html`

[13]`https://github.com/airbnb/javascript`

line.

Some languages allow omitting the block delimiters in some cases, e.g., when the block consists of only one line, to make the code less verbose. The Linux Kernel Coding Style argues in favor of this practice. The Google Java Style Guide and the AirBNB JavaScript Style Guide argue in the opposite direction. The results obtained by Gopstein et al. (2017) suggest that omitting braces may be bug-prone. Furthermore, using names to block delimiters that indicate the context of the block seems to increase code comprehension. It becomes apparent when the code has nested blocks, where the information about the block in the name of delimiters could avoid misunderstanding. To the best of our knowledge, no programming language in widespread use supports this approach.

Long or complex code line vs short or simple code line. Comparisons in this group show that keeping line lengths within the 80-character limit is considered positive for legibility (Santos and Gerosa, 2018). On the other hand, Santos and Gerosa (2018) and Medeiros et al. (2019) investigated the use of one vs. multiple statements per line and obtained different results. Medeiros et al. (2019) found out that one statement per line is preferred by the participants, but Santos and Gerosa (2018) did not find a significant difference between one or more statements per line. Both studies use only the subjects' opinions as the dependent variable. The Style Guide for Python Code[14] and Linux Kernel Coding Style[15] agree with the 80-character limit. The Google Java Style Guide[16] recommends one statement per line but recommends 100-character lines. The AirBNB JavaScript Style Guide also establishes 100 characters as the limit for lines. It makes no prescription about number of statements per line.

Word boundary styles. The only comparison (Binkley et al., 2013) in this group reveals that camel case is a positive standard for legibility, compared to snake case. This pattern is adopted in the examined coding style guides for Java and JavaScript. Python uses camel case for class names and for method names *"where that's already the prevailing style [..] to retain backwards compatibility."*. For most names, it suggests the use of snake case, with additional trailing or leading underscores for special identifiers. The Linux

[14]`https://peps.python.org/pep-0008/`
[15]`https://www.kernel.org/doc/html/v4.10/process/coding-style.html`
[16]`https://google.github.io/styleguide/javaguide.html`

Kernel Coding Style explicitly argues against camel case and prescribes the use of snake case.

5.3. Statistical power of the analyzed studies.

The power of a statistical test describes the probability that a statistical test will correctly identify a genuine effect (Ellis, 2010). In other words, a statistical test with sufficient power is unlikely to accept a null hypothesis when it should be rejected. A scenario where a null hypothesis is rejected when it should not have been rejected is called a Type II error (Cohen, 1992). The calculation of the sample size required to achieve certain level of statistical power is called power analysis. The latter depends on the significance criterion, the sample size, and the population effect size.

For seven of the comparisons we have investigated, it was not possible to reject the null hypothesis. However, it is not clear whether these results stem from the absence of a statistically significant difference or from a lack of power. Among the analyzed studies, only the works of Bauer et al. (2019) and Gopstein et al. (2017) report a concern with this aspect. In both cases, the authors have calculated the sample sizes that would yield what they considered an acceptable level of statistical power. Null results reported by other studies are difficult to judge because they have no associated confidence level, due to the lack of power analysis and reporting of effect sizes and the generally low sample sizes. These low sample sizes imply that these studies are only able to detect large effect sizes. The study of Siegmund et al. (2017) presents an illustrative example. This study attempts to compare the differences in brain activation of subjects exposed to pretty-printed code and code whose layout is disrupted. Since this study involved only 11 participants, its power is very low and it can only detect statistically significant differences if the effect size is large.

Another example is the pair of studies performed by Miara et al. (1983) and Bauer et al. (2019). Both compare different indentation levels, with the latter being inspired by the former. On the one hand, the study of Miara et al. involved 86 subjects and found a statistically significant difference between different indentation levels and also between novices and experts. On the other hand, the study conducted by Bauer et al. had only 22 participants and could not find statistically significant differences. Although still not common in Software Engineering, meta-analyses (Ellis, 2010) could be leveraged to combine the results of these different studies. In this manner, it would be possible to identify subtler differences, if they exist.

5.4. Limitations of our study and of existing studies.

The studies analyzed in this work have been conducted between 1977 and 2019 and cover a wide range of formatting-related issues. Nevertheless, there are other issues beyond what we have managed to capture. For example, in our work, we did not consider studies that investigate the influence of typography, colors, contrast, or dynamic presentation of program elements on the ability of developers to identify program elements. In summary, if it falls outside what can be tinkered with at the source code level, in an ASCII text editor, it is beyond the scope of this work. Investigating aspects that go beyond these limits is left for future work.

Even within the strictly-defined boundaries we adhere to, much is still unknown. As pointed out when discussing statistical power, many previous studies are inconclusive and it is still not clear if well-established practices actually have any effect on legibility. For example, we have identified only a single paper that compares different types of block delimiters (Sykes et al., 1983) and that paper was published almost 40 years ago, at a time when software development was very different from what it is today. One point that can be raised against this kind of study is whether such choices matter at all, especially for experienced developers, since these can be seen as minute details. Previous work (Gopstein et al., 2017, 2018; Medeiros et al., 2019; Stefik and Siebert, 2013) suggests that they do, though, even in very mature, high-complexity projects Gopstein et al. (2018).

Considering the learning activities model (Section 2.3) proposed by Fuller et al. (2007) and extended by Oliveira et al. (2020), the analyzed studies required participants to Inspect, Trace, Present, Memorize, Relate, or Giving an opinion. This means that no study required the participants to apply the knowledge obtained from understanding the program, e.g., by implementing a new feature or fixing a bug. More generally, these studies focus on “lower” cognitive skills according to the model of Fuller et al. (2007). Program comprehension aims to support other activities such as performing maintenance, fixing bugs, writing tests, etc. The absence of studies requiring participants to conduct these activities points to a possible direction for future work.

6. Threats to validity

This section discusses some of the threats to the validity of this study.

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. Journal papers were considered in the actual review. 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 and we did not have the resources to analyze so many papers.

Internal validity. This study was conducted by multiple 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 perform. Because of that, we might have misclassified these studies when mapping studies to task types. Also, another threat is the period when primary study searches were conducted. Relevant new work may have been published recently and its results may impact our conclusions. To mitigate this threat, we intend to perform a search with the same search string considering only the subsequent years of the first one.

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.

7. Conclusion

In this paper, through a systematic review of the literature, we investigated what formatting elements have been shown by previous studies to have a positive impact on code legibility when compared to functionally equivalent alternatives in human-centric studies. We present a comprehensive categorization of the elements found. In addition, we analyzed the results considering the subjects, tasks, and response variables.

The studies we identified were categorized in five groups: formatting styles, spacing, block delimiters, long or complex code lines, and word boundary styles. Our results show that book format style, the placement of the block delimiter on its own line, the use of ENDIF/ENDWHILE for block delimiters, the camel case convention for identifier naming, and the practice that line lengths should be kept within the limit of 80 exhibited positive results in the investigated studies. On the other hand, we showed that 4 of the comparisons found were not statistically significant and another 3 comparisons show divergence between the authors who investigated them.

Our study focused only on code formatting elements. Nonetheless, other studies on code comprehension investigated aspects related to structural and semantic characteristics of the code. Thus, we plan to investigate in future work what these elements are and which of them are positive for the understanding of the code. Furthermore, we want to investigate which main learning activities are being used in these studies.

References

R. J. Miara, J. A. Musselman, J. A. Navarro, B. Shneiderman, Program Indentation and Comprehensibility, Communications of the ACM 26 (1983) 861–867. URL: `http://doi.acm.org/10.1145/182.358437`. doi:`10.1145/182.358437`.

D. Binkley, M. Davis, D. Lawrie, J. I. Maletic, C. Morrell, B. Sharif, The impact of identifier style on effort and comprehension, Empirical Software Engineering 18 (2013) 219–276. URL: `http://dx.doi.org/10.1007/s10664-012-9201-4`. doi:`10.1007/s10664-012-9201-4`.

D. Oliveira, R. Bruno, F. Madeiral, F. Castor, Evaluating code readability and legibility: An examination of human-centric studies, in: 2020 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE, 2020, pp. 348–359.

M. Arab, Enhancing Program Comprehension: Formatting and Documenting, ACM SIGPLAN Notices 27 (1992) 37–46. URL: `http://doi.acm.org/10.1145/130973.130975`. doi:`10.1145/130973.130975`.

R. M. a. d. Santos, M. A. Gerosa, Impacts of Coding Practices on Readability, in: Proceedings of the 26th Conference on Program Comprehension (ICPC '18), ACM, New York, NY, USA, 2018, pp. 277–285. URL: `http://doi.acm.org/10.1145/3196321.3196342`. doi:`10.1145/3196321.3196342`.

J. Siegmund, N. Peitek, C. Parnin, S. Apel, J. Hofmeister, C. Kästner, A. Begel, A. Bethmann, A. Brechmann, Measuring Neural Efficiency of Program Comprehension, in: Proceedings of the 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE '17), ACM, New York, NY, USA, 2017, pp. 140–150. URL: `http://doi.acm.org/10.1145/3106237.3106268`. doi:`10.1145/3106237.3106268`.

J. Bauer, J. Siegmund, N. Peitek, J. C. Hofmeister, 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), IEEE Press, Piscataway, NJ, USA, 2019, pp. 154–164. URL: `https://doi.org/10.1109/ICPC.2019.00033`. doi:`10.1109/ICPC.2019.00033`.

A. C. Benander, B. A. Benander, H. Pu, Recursion vs. Iteration: An Empirical Study of Comprehension, Journal of Systems and Software 32 (1996) 73–82. URL: `http://www.sciencedirect.com/science/article/pii/0164121295000437`. doi:`https://doi.org/10.1016/0164-1212(95)00043-7`.

J. C. Hofmeister, J. Siegmund, D. V. Holt, Shorter Identifier Names Take Longer to Comprehend, Empirical Software Engineering 24 (2019) 417–443. URL: `https://doi.org/10.1007/s10664-018-9621-x`. doi:`10.1007/s10664-018-9621-x`.

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

U. Fuller, C. G. Johnson, T. Ahoniemi, D. Cukierman, I. Hernán-Losada, J. Jackova, E. Lahtinen, T. L. Lewis, D. M. Thompson, C. Riedesel, E. Thompson, Developing a Computer Science-specific Learning Taxonomy, ACM SIGCSE Bulletin 39 (2007) 152–170. URL: `https://doi.org/10.1145/1345375.1345438`. doi:`10.1145/1345375.1345438`.

R. P. L. Buse, W. R. Weimer, Learning a Metric for Code Readability, IEEE Transactions on Software Engineering 36 (2010) 546–558. URL: `https://doi.org/10.1109/TSE.2009.70`. doi:`10.1109/TSE.2009.70`.

J. R. d. Almeida, J. B. Camargo, B. A. Basseto, S. M. Paz, Best practices in code inspection for safety-critical software, IEEE software 20 (2003) 56–63.

J.-C. Lin, K.-C. Wu, Evaluation of Software Understandability Based On Fuzzy Matrix, in: Proceedings of the 2008 IEEE International Conference on Fuzzy Systems (IEEE World Congress on Computational Intelligence), 2008, pp. 887–892.

X. Xia, L. Bao, D. Lo, Z. Xing, A. E. Hassan, S. Li, Measuring Program Comprehension: A Large-Scale Field Study with Professionals, IEEE Transactions on Software Engineering 44 (2018) 951–976.

P. B. Gough, W. E. Tunmer, Decoding, Reading, and Reading Disability, Remedial and special education 7 (1986) 6–10.

W. A. Hoover, P. B. Gough, The simple view of reading, Reading and writing 2 (1990) 127–160.

W. H. DuBay, The principles of readability., Online Submission (2004).

C. Tekfi, Readability Formulas: An Overview, Journal of documentation 43 (1987) 261–273.

E. Daka, J. Campos, G. Fraser, J. Dorn, W. Weimer, Modeling Readability to Improve Unit Tests, in: Proceedings of the 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE '15), Association for Computing Machinery, New York, NY, USA, 2015, pp. 107–118. URL: `https://doi.org/10.1145/2786805.2786838`. doi:`10.1145/2786805.2786838`.

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

S. Zuffi, C. Brambilla, G. Beretta, P. Scala, Human computer interaction: Legibility and contrast, in: Proceedings of the 14th International Conference on Image Analysis and Processing (ICIAP '07), IEEE, 2007, pp. 241–246.

D. Gopstein, J. Iannacone, Y. Yan, L. DeLong, Y. Zhuang, M. K.-C. Yeh, J. Cappos, Understanding Misunderstandings in Source Code, in: Proceedings of the 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE '17), ACM, New York, NY, USA, 2017, pp. 129–139. URL: http://doi.acm.org/10.1145/3106237.3106264. doi:10.1145/3106237.3106264.

S. Ajami, Y. Woodbridge, D. G. Feitelson, Syntax, predicates, idioms – what really affects code complexity?, Empirical Software Engineering 24 (2019) 287–328. URL: https://doi.org/10.1007/s10664-018-9628-3. doi:10.1007/s10664-018-9628-3.

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 11 (1977) 105–113. URL: http://doi.acm.org/10.1145/390018.808317. doi:10.1145/390018.808317.

G. Scanniello, 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), IEEE Computer Society, USA, 2013, pp. 190–199. URL: https://doi.org/10.1109/ICSM.2013.30. doi:10.1109/ICSM.2013.30.

A. Jbara, D. G. Feitelson, On the Effect of Code Regularity on Comprehension, in: Proceedings of the 22nd International Conference on Program Comprehension (ICPC '14), ACM, New York, NY, USA, 2014, pp. 189–200. URL: http://doi.acm.org/10.1145/2597008.2597140. doi:10.1145/2597008.2597140.

E. S. Wiese, A. N. Rafferty, 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), IEEE Press, Piscataway, NJ, USA, 2019, pp. 84–94. URL: https://doi.org/10.1109/ICSE-SEET.2019.00017. doi:10.1109/ICSE-SEET.2019.00017.

M. B. O'Neal, W. R. Edwards, Complexity Measures for Rule-Based Programs, IEEE Transactions on Knowledge and Data Engineering 6 (1994) 669–680. URL: https://doi.org/10.1109/69.317699. doi:10.1109/69.317699.

D. Lawrie, C. Morrell, H. Feild, D. Binkley, Effective identifier names for comprehension and memory, Innovations in Systems and Software Engineering 3 (2007) 303–318. URL: https://doi.org/10.1007/s11334-007-0031-2. doi:10.1007/s11334-007-0031-2.

S. Fakhoury, D. Roy, Y. Ma, V. Arnaoudova, O. Adesope, Measuring the impact of lexical and structural inconsistencies on developers' cognitive load during bug localization, Empirical Software Engineering (2019) 1–39. URL: https://doi.org/10.1007/s10664-019-09751-4. doi:10.1007/s10664-019-09751-4.

L. Anderson, B. Bloom, D. Krathwohl, P. Airasian, K. Cruikshank, R. Mayer, P. Pintrich, J. Raths, M. Wittrock, A Taxonomy for Learning, Teaching, and Assessing: A Revision of Bloom's Taxonomy of Educational Objectives, Longman, 2001. URL: https://books.google.com.br/books?id=EMQlAQAAIAAJ.

S. Schulze, J. Liebig, J. Siegmund, 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), ACM, New York, NY, USA, 2013, pp. 65–74. URL: http://doi.acm.org/10.1145/2517208.2517215. doi:10.1145/2517208.2517215.

A. Stefik, S. Siebert, An Empirical Investigation into Programming Language Syntax, ACM Transactions on Computing Education (TOCE) 13 (2013) 19:1–19:40. URL: http://doi.acm.org/10.1145/2534973. doi:10.1145/2534973.

B. D. Chaudhary, H. V. Sahasrabuddhe, Meaningfulness as a Factor of Program Complexity, in: Proceedings of the ACM 1980 Annual Conference (ACM '80), ACM, New York, NY, USA, 1980, pp. 457–466. URL: http://doi.acm.org/10.1145/800176.810001. doi:10.1145/800176.810001.

T. Tenny, Program Readability: Procedures Versus Comments, IEEE Transactions on Software Engineering 14 (1988) 1271–1279. doi:10.1109/32.6171.

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

J. J. Dolado, M. Harman, M. C. Otero, L. Hu, An Empirical Investigation of the Influence of a Type of Side Effects on Program Comprehension, IEEE Transactions on Software Engineering 29 (2003) 665–670. URL: https://doi.org/10.1109/TSE.2003.1214329. doi:10.1109/TSE.2003.1214329.

A. Jbara, D. G. Feitelson, How programmers read regular code: a controlled experiment using eye tracking, Empirical Software Engineering 22 (2017) 1440–1477. URL: https://doi.org/10.1007/s10664-016-9477-x. doi:10.1007/s10664-016-9477-x.

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

J. Cohen, A Coefficient of Agreement for Nominal Scales, Educational and Psychological Measurement 20 (1960) 37–46. URL: `https://doi.org/10.1177/001316446002000104`. doi:`10.1177/001316446002000104`. `arXiv:https://doi.org/10.1177/001316446002000104`.

S. Keele, et al., Guidelines for performing systematic literature reviews in software engineering, Technical Report, Technical report, Ver. 2.3 EBSE Technical Report. EBSE, 2007.

D. Binkley, D. Lawrie, S. Maex, C. Morrell, Impact of limited memory resources, in: 2008 16th IEEE International Conference on Program Comprehension, IEEE, 2008, pp. 83–92.

D. Binkley, D. Lawrie, S. Maex, C. Morrell, Identifier length and limited programmer memory, Science of Computer Programming 74 (2009) 430–445. URL: `http://dx.doi.org/10.1016/j.scico.2009.02.006`. doi:`10.1016/j.scico.2009.02.006`.

J. R. Wood, L. E. Wood, Card sorting: current practices and beyond, Journal of Usability Studies 4 (2008) 1–6.

R. P. Buse, W. R. Weimer, A Metric for Software Readability, in: Proceedings of the 2008 International Symposium on Software Testing and Analysis (ISSTA '08), ACM, New York, NY, USA, 2008, pp. 121–130. URL: `http://doi.acm.org/10.1145/1390630.1390647`. doi:`10.1145/1390630.1390647`.

A. Trockman, K. Cates, M. Mozina, T. Nguyen, C. Kästner, B. Vasilescu, "Automatically Assessing Code Understandability" Reanalyzed: Combined Metrics Matter, in: Proceedings of the 15th International Conference on Mining Software Repositories (MSR '18), ACM, New York, NY, USA, 2018, pp. 314–318. URL: `http://doi.acm.org/10.1145/3196398.3196441`. doi:`10.1145/3196398.3196441`.

S. Scalabrino, M. Linares-Vásquez, R. Oliveto, D. Poshyvanyk, A Comprehensive Model for Code Readability, Journal of Software: Evolution and Process 30 (2018) e1958. URL: `https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.1958`. doi:`10.1002/smr.1958`. `arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/smr.1958`.

S. Scalabrino, G. Bavota, C. Vendome, M. Linares-Vásquez, D. Poshyvanyk, R. Oliveto, Automatically Assessing Code Understandability, IEEE Transactions on Software Engineering (2019) 1–1. doi:`10.1109/TSE.2019.2901468`.

D. Posnett, A. Hindle, P. Devanbu, A Simpler Model of Software Readability, in: Proceedings of the 8th Working Conference on Mining Software Repositories (MSR '11), Association for Computing Machinery, New York, NY, USA, 2011, pp. 73–82. URL: `https://doi.org/10.1145/1985441.1985454`. doi:`10.1145/1985441.1985454`.

N. Kasto, 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), Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 2013, pp. 59–65. URL: `http://dl.acm.org/citation.cfm?id=2667199.2667206`.

R. Brooks, Using a behavioral theory of program comprehension in software engineering, in: Proceedings of the 3rd international conference on Software engineering, 1978, pp. 196–201.

B. Chance, Z. Zhuang, C. UnAh, C. Alter, L. Lipton, Cognition-activated low-frequency modulation of light absorption in human brain., Proceedings of the National Academy of Sciences 90 (1993) 3770–3774.

P. W. Oman, C. R. Cook, Typographic Style is More than Cosmetic, Communications of the ACM 33 (1990) 506–520. URL: `http://doi.acm.org/10.1145/78607.78611`. doi:`10.1145/78607.78611`.

M. Johnson, G. Beekman, Oh! THINK's lightspeed Pascal!, WW Norton, 1988.

D. M. Ritchie, B. W. Kernighan, M. E. Lesk, The C programming language, Prentice Hall Englewood Cliffs, 1988.

J. L. Peterson, On the formatting of pascal programs, ACM Sigplan Notices 12 (1977) 83–86.

J. E. Crider, Structured formatting of pascal programs, ACM Sigplan Notices 13 (1978) 15–22.

G. Gustafson, Some practical experiences formatting pascal programs, ACM Sigplan Notices 14 (1979) 42–49.

F. Medeiros, G. Lima, G. Amaral, S. Apel, C. Kästner, M. Ribeiro, R. Gheyi, An investigation of misunderstanding code patterns in C open-source software projects, Empirical Software Engineering 24 (2019) 1693–1726. URL: `https://doi.org/10.1007/s10664-018-9666-x`. doi:`10.1007/s10664-018-9666-x`.

F. Sykes, R. T. Tillman, B. Shneiderman, The Effect of Scope Delimiters on Program Comprehension, Software: Practice and Experience 13 (1983) 817–824. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.4380130908. doi:10.1002/spe.4380130908. arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.4380130908.

P. D. Ellis, The Essential Guide to Effect Sizes: Statistical Power, Meta-Analysis, and the Interpretation of Research Results, Cambridge University Press, 2010.

J. Cohen, Statistical power analysis, Current directions in psychological science 1 (1992) 98–101.

D. Gopstein, H. H. Zhou, P. Frankl, J. Cappos, Prevalence of Confusing Code in Software Projects: Atoms of Confusion in the Wild, in: Proceedings of the 15th International Conference on Mining Software Repositories (MSR '18), ACM, New York, NY, USA, 2018, pp. 281–291. URL: http://doi.acm.org/10.1145/3196398.3196432. doi:10.1145/3196398.3196432.