A systematic literature review on the impact of formatting elements on code legibility☆

Delano Oliveira [a,b,1], Reydne Santos [a,1], Fernanda Madeiral [c], Hidehiko Masuhara [d], Fernando Castor [e,a,*]

[a] *Federal University of Pernambuco, Recife, Brazil*
[b] *Federal Institute of Pernambuco, Recife, Brazil*
[c] *Vrije Universiteit Amsterdam, Amsterdam, Netherlands*
[d] *Tokyo Institute of Technology, Tokyo, Japan*
[e] *Utrecht University, Utrecht, Netherlands*

ARTICLE INFO

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 alternatives were found to be more legible for human subjects.

Method: We conducted a systematic literature review and identified 15 papers containing human-centric studies that directly compared alternative formatting elements. We analyzed and organized these formatting elements using a card-sorting method.

Results: We identified 13 formatting elements (e.g., indentation) and 33 levels of formatting elements (e.g., two-space indentation), which are about formatting styles, spacing, block delimiters, long or complex code lines, and word boundary styles. While some levels were found to be statistically better than other equivalent ones in terms of code legibility, e.g., appropriate use of indentation with blocks, others were not, e.g., formatting layout. For identifier style, we found divergent results, where one study found a significant difference in favor of camel case, while another study found a positive result in favor of snake case.

Conclusion: The number of identified papers, some of which are outdated, and the many null and contradictory results emphasize the relative lack of work in this area and underline the importance of more research. There is much to be understood about how formatting elements influence code legibility before the creation of guidelines and automated aids to help developers make their code more legible.

1. Introduction

Program comprehension is a 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.

Formatting elements, such as spacing, are *factors* that impact the *legibility* of the source code and, consequently, 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.

Ensuring the understandability of software projects is a challenging task. Some software organizations (e.g., Google[2] and Sun

---

☆ Editor: Gabriele Bavota.
* Corresponding author.
E-mail addresses: dho@cin.ufpe.br (D. Oliveira), reydne.bruno@gmail.com (R. Santos), fer.madeiral@gmail.com (F. Madeiral), masuhara@acm.org (H. Masuhara), f.j.castordelimafilho@uu.nl (F. Castor).
[1] Authors contributed equally.

[2] https://google.github.io/styleguide/, last access May 22, 2023.

Microsystems[3]) have adopted code conventions to enforce the usage of software development best practices by their developers. Smit et al. (2011) explain that *"code conventions are a body of advice on lexical and syntactic aspects of code, aiming to standardize low-level code design under the assumption that such a systematic approach will make code easier to read, understand, and maintain"*.

There exist guidelines and coding standards for different programming languages, such as the Google Java Style Guide[4] for the Java language, which describe 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.

In this paper, we aim to provide a more comprehensive view of the existing knowledge about the impact of different formatting elements on code legibility. To do so, we examined empirical studies performed by researchers with human subjects aiming to find which formatting alternatives are the best for the **legibility** of source code. We conducted a systematic literature review that, from an initial set of 4,914 documents, analyzes 15 papers. We selected papers that directly compared alternative levels of formatting elements and organized them through a card-sorting process. Finally, for each formatting element, we examined 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 identified 13 formatting elements (e.g., indentation) and 33 alternative levels of these factors (e.g., two-space indentation) in 27 comparisons (e.g., one comparison considered zero, two, four, and six indentation levels). These formatting elements are about formatting styles (e.g., formatting layout), spacing (e.g., indentation), block delimiters (e.g., block delimiter visibility), long or complex code line (e.g., statements per line), and word boundary styles (i.e., identifier styles).

All comparisons for four factors showed statistically significant results in favor of the same level (e.g., book format for formatting style). For other four factors, there were comparisons where statistically significant differences between the levels were found, while other comparisons did not show differences. For instance, in comparisons involving horizontal spacing, Miara et al. (1983) found out that two-space indentation leads to code that is easier to understand than zero, four, or six spaces. However, recent works (Santos and Gerosa, 2018; Bauer et al., 2019) did not find differences between indentation levels, though the studied source code snippets of these studies were from different programming languages. For the identifier style factor, there were divergent results. Sharif and Maletic (2010) found that snake case is the best alternative for identifier names. However, Binkley et al. (2013) found out that the usage of camel case is better than snake case in one study, and no statistical difference between the two formatting alternatives was found in a second study. For the

remaining four factors, e.g., blank space around operators and parameters (Sampaio and Barbosa, 2016), no statistically significant differences were found.

This study highlights that our current understanding of code legibility is limited. The area is **immature**, i.e., there are few studies on the topic, the studies that do exist are **narrow in scope**, i.e., they employ a restricted and small set of approaches to evaluate alternative solutions, and the results of many studies are **inconclusive**, in part due to lack of statistical power. Furthermore, 5/15 papers we analyzed are **outdated**, having been published 30 or more years ago. This indicates the need for more research in this area. Ultimately, understating the impact of formatting elements on code legibility would allow the creation of guidelines and automated aids, e.g., linters and recommendation systems, to help 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 compared 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 and legibility 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 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 the 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 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) state 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 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. A recent study by Feitelson (2022) also presented the tasks and metrics adopted in studies about code comprehension. Different from our previous study, their focus was narrower: they performed an in-depth analysis of the factors involved in only one of the levels of learning skills ("understanding") presented by Oliveira et al. (2020), and on concrete activities performed by developers. 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 studies required 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 were 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.

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.

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.

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 reviewed 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 light on this topic by analyzing the cognitive skill requirements associated with each kind of task.

Table 1

Learning activities from our previous study (Oliveira et al., 2020), 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. We took the definitions directly from Fuller et al. (2007), except when a definition is not applicable to our code understanding context. The examples are either extracted from tasks from the primary studies of our previous work (Oliveira et al., 2020) or general ones when no task involved that activity.

Activity	Description and example
Adapt	Modify a solution for other domains/ranges, i.e., change 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 complexity of a solution by reading the source code. Example: Identify the function where the program would spend more time running.
Apply	Use an existing solution as a component to implement a part of a program. Example: Reuse of off-the-shelf components.
Debug	Detect and correct flaws in a design. Example: Given a program, identify faults and fix them (Scanniello and Risi, 2013).
Design	Understand a program and devise a solution for a new problem specification. Example: Given a program, sketch a solution for a new functionality.
Implement	Read (part of) a program and implement 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 by reading the source code. Example: Given a program and the design of a new functionality, 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	Identify a concept or code structure from a base knowledge or vocabulary of the domain 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 to modify non-functional properties of a program or, at a larger scale, re-engineer it. Example: Rewrite a function so as to avoid using conditional expressions.
Relate	Understand different solutions and identify distinctions and similarities, pros and cons of these 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, i.e., 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 a source code to find or understand fine-grain static elements. Inspect is similar to Analyze, but the inspected aspect of the program is static instead of dynamic. Example: "All variables in this program are global [true/false] " (Miara et al., 1983).
Memorize*	Memorize a source 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).

According to the Merriam-Webster Thesaurus, to learn something is *"to 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 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 required 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). 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 tasks listed in the table were extracted from the previously analyzed studies. Furthermore, the activities to which they are mapped are the ones that were required from subjects in those studies, i.e., there may be other activities associated with those tasks that have not been investigated.

Table 2 shows that there is a direct correspondence between some tasks and activities. For example, all the instances of the *"Find and fix bugs in the code"* task involve the Debug activity, and all the tasks that require subjects to provide an opinion are connected to the Opinion activity. In addition, some tasks may be connected to various activities. For instance, *"Modify*

Table 2

Mapping of tasks (rows) to learning activities (columns), adapted from Oliveira et al. (2020). The rows exhibit tasks that subjects performed during the investigated studies and the columns present the activities explained in Table 1. Opinion is market with "*" because it is not directly related to learning.

	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	✓	✓	✓			✓			✓	✓

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.

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 the following research questions in this paper:

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

RQ2 Which levels of formatting elements have been found to make the source code more legible?

To answer our research questions, we conducted a systematic literature review designed following the guidelines proposed by Kitchenham et al. (2015). Fig. 1 presents the roadmap of our review, including all steps we followed. First, we performed the selection of studies. We started with a set of papers (11) about code legibility found in our previous work (Oliveira et al., 2020), which we used 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 3,999 unique 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). After this process, 14 papers were selected. Then, we performed backward and forward snowballing (Section 3.5), where we found 915 new documents, which passed through all the selection steps. We found one additional paper, totaling 15 papers for our review. Finally, the selected 15 papers were analyzed (Section 3.6) for data extraction and synthesis to answer our research questions. We detail these steps in the following sections.

Full disclosure on novelty. Our methodology resembles the one employed in our previous work (Oliveira et al., 2020). However, the search for papers is different, which means the whole process started based on a new set of papers for this work. Moreover, the data analysis and synthesis methods are completely new because we answer different research questions in this paper. A detailed elaboration on the differences between the two works is presented in Section 3.7.

3.1. Search strategy

Our search strategy comprises three parts: the selection of papers to be used as seed studies, the definition of a generic search string, and the automatic search in search engines. First, we selected relevant papers as seeds to define a generic search string with terms that would return relevant papers for answering our research questions. We selected 11 papers from our previous study (Oliveira et al., 2020) to be considered as seeds in this one. These papers contain studies that compare two or more ways of writing code by only changing formatting elements to help code understanding in terms of legibility. This is the kind of study we are searching for in this work and, therefore, these papers form the ideal seed set for systematically searching other papers.

We analyzed the title and keywords of the seed papers and extracted the general terms related to our research questions. We used the resulting terms to build the following search string:
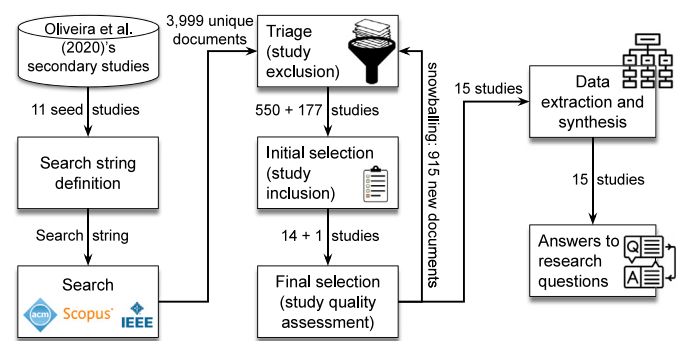

Fig. 1. Systematic literature review roadmap.

$$Title(ANY(terms)) \text{ OR } Keywords(ANY(terms)),$$

where *terms* = { "code comprehension", "code understandability", "code understanding", "code readability", "code complexity", "code misunderstanding", "code patterns", "program comprehension", "program understandability", "program understanding", "program readability", "program idioms", "program style", "program patterns", "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.

Finally, we performed an automatic search for studies using our generic search string adapted for three search engines: ACM Digital Library[5], IEEE Explore[6], and Scopus[7]. We retrieved 1,520, 979, and 3,458 documents, respectively, on October 10, 2022. 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 3,999 unique documents. The 11 seed papers were returned by the automatic search.

3.2. Triage (study exclusion)

The 3,999 documents retrieved with our automatic search passed through a triage process so that we could discard clearly irrelevant documents. For that process, we first defined five exclusion criteria:

EC1. The study is clearly irrelevant to our research questions, i.e., it is not primarily related to source code comprehension, readability, legibility, or hard-to-understand code, or does not involve any comparison of different ways of writing code.

EC2. The study is not a full paper (e.g., PhD theses or 4-page papers), is not written in English, or was not peer-reviewed: not considering these types of documents in systematic reviews is a common practice (Kitchenham et al., 2015).

EC3. The study is about legibility, readability, or understandability metrics.

EC4. The study is about program comprehension aids, such as visualizations or other forms of analysis or sensory aids, e.g., trace-based execution, code summarization, and character encodings supporting colors.

EC5. The study focuses on accessibility, e.g., targets individuals with visual impairments or neurodiverse developers.

Then, each of the 3,999 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 did 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 550 papers.

3.3. Initial selection (study inclusion)

After discarding clearly irrelevant documents in the triage process, we applied the following inclusion criteria to the 550 papers to build our initial set of papers:

IC1 (Scope). The study must be primarily related to *legibility*.

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 process, 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. 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 14 papers.
Deprecated studies. We identified some papers that we refer to as *deprecated*. A paper is deprecated if it is extended by another paper that we selected. For instance, the paper of Binkley et al. (2009) was extended in a subsequent paper (Binkley et al., 2013). In this case, we consider the former to be deprecated and only take the latter into account.

3.4. Study quality assessment

The selected 14 papers passed through a final selection step, aiming 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). Three questions were about study design, e.g., *"are the aims clearly stated?"*, four questions were about analysis, e.g., *"are the data collection methods adequately described?"*, and two questions were about rigor, e.g., *"do the researchers explain the threats to the study validity?"*. Each question was answered with one of the following possible answers: yes (1), partially (0.5), and no (0). The sum of the answers for a given paper is its quality 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 of this paper. 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 papers 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 papers were removed because of low quality.

3.5. Snowballing

To make sure our study is comprehensive, we carried out a snowballing process. To do so, we collected new papers by gathering all the references used in the included papers (backward snowballing) and all the papers that cited the included papers in our review (forward snowballing). We used Google Scholar to extract the citations of included papers. The newly collected papers passed through the triage (study exclusion), initial selection (study inclusion), and final selection (study quality assessment). At this stage, only papers published from the 1990s onwards were included because that period marked a significant advancement in the field of software engineering with the emergence of object-oriented programming. We repeated this process until no new paper was found. We performed one complete iteration, and no new paper was found for a second one. With the initial set of 14 selected papers, we gathered 915 new documents. After analyzing them by considering the exclusion and inclusion criteria, we found one new paper, totaling 15 papers for our review.

3.6. Data analysis

To answer our research questions (which is done in Section 4), we analyzed a total of 15 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. 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 formatting elements (e.g., indentation) and their levels (e.g., two-space indentation) being compared. For RQ2, for each 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, 4) the characteristics of the human subjects, and 5) the programming languages considered.

Synthesis process. To better understand and present the results in an organized way, we performed card sorting (Wood and Wood, 2008) on the extracted formatting elements. With this method, we transformed the formatting elements into cards and then grouped the cards based on their similarities. This process was bottom-up, i.e., formatting elements → groups, as follows.

Formatting elements (factors). We used the formatting elements extracted from the studies as factors in the card-sorting process. Then, we created a card for each factor evaluated by the study (e.g., indentation). For synonymous factors, we created one unique card to represent them. For instance, Santos and Gerosa

(2018) evaluated whether using a beginning block delimiter in the same line of their corresponding statements (e.g., class declaration) is better than in 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 location" for these factors. The levels (e.g., in the statement line and in a separate line) of each factor compared by the studies are described and evaluated separately.

Groups. 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 *"Vertical and horizontal spacing"*. We created the first group, named "Group 1", and included that card in it. The second card we analyzed was *"Formatting layout"*. Such a card is not similar to the card in "Group 1", so we created a second group named "Group 2" for it. The third card we analyzed was *"Indentation"*, which is similar to the card in "Group 1", so we included it in that group as well. After including all cards in some groups, 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 of this paper, and the results were later refined with the collaboration of all authors in multiple live sessions. The identified groups and factors are presented in Table 3 and detailed in Section 4.

3.7. Differences between this study and the previous one

This work is part of a bigger research project where we investigate formatting elements, coding idioms, and programming constructs that affect code legibility and readability, what they are, which ones are the best, and how researchers and developers evaluate what "best" means. Thus, the methodology we used in this work resembles the methodology of a previous study (Oliveira et al., 2020), but the studies are different. 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 and data analysis (explained in Section 3.6) are also different. Moreover, we changed the process of the search for papers to better match the goal of this study. We used papers found in our previous work as seed papers for defining a new search string for this work.

The remainder of the methodology, which deals with study selection (i.e., study exclusion, study inclusion, and study quality assessment), was reused or adapted from our previous work. Because some studies that were selected in our previous work are not adequate to answer the research questions of this study, we changed inclusion criteria, more specifically IC1 and IC3, to select only studies on legibility that directly compare formatting elements or styles that can be clearly identified. Finally, a subset of the 4,914 documents had already been analyzed in the selection phases of our previous work, and therefore we partially reused the results from it.

Table 3
Card sorting results.

Groups	Formatting elements (factors)
Formatting	Formatting style Formatting layout
Spacing	Indentation Appropriate use of indentation with blocks Vertical and horizontal spacing Vertical spacing between related instructions Blank space around operators and parameters
Block Delimiters	Block delimiter location Block delimiter style Block delimiter visibility
Long or Complex Code Line	Line length Statements per line
Word Boundary Styles	Identifier style

3.8. Data availability

Raw data, such as the list of the 3,999 documents returned by our automatic search, 915 documents returned by snowballing, and additional details about our methodology, such as the questionnaires we used for the 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 organized the subsections by the groups of formatting elements (factors) we found in our review, which are named formatting (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). In each section, we address both research questions. The first research question aims to identify the formatting elements that researchers investigated in their studies about code legibility, i.e., the characteristics of the code that make it easier or harder to identify its elements. For a given group, we present the formatting elements with their corresponding levels, separated per the primary studies included in our review. Table 3 presents the mapping of groups to formatting elements. With the second research question, we aim to synthesize the studies' findings related to legibility by comparing the levels of the formatting elements. For that, we consider data such as the studies' answers, statistical tests, activities performed by humans, and dependent variables. Table 4 presents an overview of the papers included in our study and summarizes the factors (from Table 3) investigated in each of them.

4.1. Formatting

This group gathers studies about different ways of formatting code from a global, higher-level perspective. Table 5 presents a summary of the results. For each factor, the table presents the references to the studies, the levels of the factors compared, the programming languages considered, the dependent variables of each study, the corresponding activities (see Section 2.3) performed by the subjects, and the main results found for each comparison of levels.

Formatting style. Oman and Cook (1990) proposed the notion of Book Format Style for structuring source code and compared it to two well-known styles for the Pascal and C languages: the Lightspeed Pascal style (Johnson and Beekman, 1988) and the Kernighan & Ritchie style (Ritchie et al., 1988). The Book Format style, as the name implies, takes inspiration from how books

Table 4
Included papers and the factors analyzed, sorted in descending order by year of publication.

Study	Study title	PL	Formatting elements (factors)
Langhout and Aniche (2021)	"Atoms of Confusion in Java"	Java	– Appropriate use of indentation with blocks – Block delimiter visibility
Bauer et al. (2019)	"Indentation: Simply a Matter of Style or Support for Program Comprehension?"	Java	– Indentation
Medeiros et al. (2019)	"An investigation of misunderstanding code patterns in C open-source software projects"	C	– Block delimiter visibility – Statements per line
Santos and Gerosa (2018)	"Impacts of Coding Practices on Readability"	Java	– Indentation – Vertical spacing between related instructions – Block delimiter location – Line length – Statements per line
Siegmund et al. (2017)	"Measuring Neural Efficiency of Program Comprehension"	Java	– Formatting layout
Gopstein et al. (2017)	"Understanding Misunderstandings in Source Code"	C/C++	– Block delimiter visibility
Sampaio and Barbosa (2016)	"Software readability practices and the importance of their teaching"	Java	– Blank space around operators and parameters – Block delimiter visibility – Statements per line
Binkley et al. (2013)	"The impact of identifier style on effort and comprehension"	C/Java	– Identifier style
Sharif and Maletic (2010)	"An Eye Tracking Study on camelCase and under_score Identifier Styles"	None	– Identifier style
Furman et al. (2002)	"A Look at Programmers Communicating through Program Indentation"	Pascal	– Indentation
Arab (1992)	"Enhancing Program Comprehension: Formatting and Documenting"	Pascal	– Block delimiter location
Oman and Cook (1990)	"Typographic Style is More than Cosmetic"	Pascal/C	– Formatting style
Miara et al. (1983)	"Program Indentation and Comprehensibility"	Pascal	– Indentation – Block delimiter location
Sykes et al. (1983)	"The Effect of Scope Delimiters on Program Comprehension"	Pascal	– Block delimiter visibility – Block delimiter style
Love (1977)	"An Experimental Investigation of the Effect of Program Structure on Program Understanding"	Pascal	– Vertical and horizontal spacing

are structured as an approach to organizing source code. In this format, programs include a preface, table of contents, chapter divisions, pagination, code paragraphs, sentence structures, and intramodule comments, among other elements. For the comparison with the Lightspeed Pascal style, the authors asked 36 students to answer 14 multiple-choice, short-answer questions about the characteristics of a code snippet in a total of 10 minutes (both requiring the Trace activity), 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 style, and both received the same instructions. Then, the authors assessed (i) the number of questions the students answered correctly (score of 1 to 14 points), (ii) the time they spent answering the questions (1 to 10 minutes), (iii) their performance score (number of correct answers per minute), and (iv) their subjective opinions (rating on a five-point scale). The study concluded that the Book Format style is considered a better formatting style compared to the Lightspeed Pascal style based on the significant differences in 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, the authors prepared a very similar experiment with 44 different students. The only difference is that the questionnaire had 10 questions instead of the 14 from the previous experiment. They assessed the same response variables and found out that the Book Format style was statistically better for correctness score (ANOVA, $p < 0.005$) and performance (ANOVA, $p < 0.005$), but there are no significant differences for time and subjective opinion.

Formatting layout. Siegmund et al. (2017) compared a pretty-printed code layout with a disrupted layout. More specifically,

Table 5
Summary of results for the Formatting group.

Factor – Study: *Levels (Programming Language)* – Dependent Variables (Activities): Results

Formatting style

Oman and Cook (1990): *book format and Lightspeed Pascal style (Pascal)*
Correctness (Trace): The book format style is the best.
Time (Trace): No significant difference.
Opinion: The book format style is the best.
Oman and Cook (1990): *book format and Kernighan & Ritchie style (C)*
Correctness (Trace): The book format style is the best.
Time (Trace): No significant difference.
Opinion: No significant difference.

Formatting layout

Siegmund et al. (2017): *pretty-printed and disrupted (Java)*
Brain Metrics (Relate): No significant difference.

they compared the code comprehension of subjects considering code snippets following common coding conventions for layout, e.g., indentation, line breaks, and correctly-placed scope delimiters, 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 such a study is that it analyzed 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 the 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 the 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 in Java: (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.

4.2. Spacing

This group assembles studies about different types of spacing in source code. Table 6 presents a summary of the analyzed results.

Indentation. Miara et al. (1983) conducted an experiment with 54 novice students and 32 professional developers and experienced students (i.e., people with three or more years of programming experience) to evaluate the influence of indentation levels and blocked code on program comprehension. In the 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 (blocked and non-blocked): the latter belongs to block delimiters, so it is presented in Section 4.3. Each subject received a program in Pascal using one of the levels of indentation and one of the methods of block indentation, accompanied by a quiz with 10 questions that required subjects to (i) select the correct answer about code characteristics (Trace and Inspect), (ii) explain what the code does (Present), and (iii) provide an opinion on the difficulty of the task (Giving an opinion). The authors considered two dependent variables: the number of questions correctly answered (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-space indentation resulted in the best results.

Furman et al. (2002) also evaluated the effect of the left (no space), normal (2-4 spaces), and random indentation on code legibility. They conducted an experiment with 24 inexperienced and 18 experienced programmers in Pascal. The subjects had to understand three sorting programs, one at a time, presented in one indentation version. The authors used DISCOVERY (a tool developed by them) to measure visual metrics. This tool printed the programs to mask the code lines with the character 'X' and only allowed subjects to look at one line at a time. After understanding the program, the subjects had to answer eight multiple-choice questions for each of the three sorting programs (Trace, Inspect, and Present). The authors evaluated the correctness of the answers, the average line-look time (i.e., time spent looking to a revealed code line), the average line-search time (i.e., time spent choosing a code line to be revealed), and the total of code lines revealed in a program. Also, the authors asked subjects to answer a subjective questionnaire (Giving an opinion) using the Likert scale to evaluate the difficulty of performing the task, the subjective preference, and the level of fatigue. They found

Table 6
Summary of results for the Spacing group.

Factor – Study: *Levels (Programming Language)* – Dependent Variables (Activities): Results
Indentation
Miara et al. (1983): *0, 2, 4, and 6 (Pascal)*
Correctness (Trace, Inspect, Present): Two-space indentation is the best.
Opinion: Two-space indentation is the best.
Furman et al. (2002): *left [0], normal [2-4], and random (Pascal)*
Correctness (Trace, Inspect, Present): No significant difference.
Visual Metrics (Trace, Inspect, Present): Left and normal are better than random indentation on line-look time, normal is better than the other levels on revealed lines, and there is no difference on line-search time.
Opinion: Normal is considered less difficult than other indentation levels for subjective difficulty and preference. However, for fatigue, normal is considered only less difficult than random indentation.
Santos and Gerosa (2018): *2 and 4 (Java)*
Opinion: No significant difference.
Bauer et al. (2019): *0, 2, 4, and 8 (Java)*
Correctness (Trace): No significant difference.
Time (Trace): No significant difference.
Visual Metrics (Trace): No significant difference.
Opinion: No significant difference.
Appropriate use of indentation with blocks
Langhout and Aniche (2021): *with and without (Java)*
Correctness (Trace): Appropriate use of indentation is the best only when it is not preceded by curly braces.
Opinion: Appropriate use of indentation is the best.
Vertical and horizontal spacing
Love (1977): *paragraphed and unparagraphed (Pascal)*
Correctness (Memorize and Present): No significant difference.
Vertical spacing between related instructions
Santos and Gerosa (2018): *with and without (Java)*
Opinion: No significant difference.
Blank space around operators and parameters
Sampaio and Barbosa (2016): *with and without (Java)*
Opinion: No significant difference.

a statistical difference between different indentation versions in terms of correctness. With the visual metrics, they found a statistical difference in line-look time (F-test, $p < 0.05$) and in the total of revealed lines (F-test, $p < 0.05$). For the line-look time, subjects spent less time in program versions with left and normal indentation than in programs with random indentation, and for the total of revealed lines, they spent less time in programs with normal indentation than in other versions. The authors did not find a statistical difference for line-search time. Finally, the subjects considered the version with normal indentation less difficult than others (F-test, $p < 0.05$), as well as their subjective preference (F-test, $p < 0.05$). However, they considered that version with normal indentation only was less fatigued than the random version (F-test, $p < 0.05$).

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. Among other practices, subjects had to choose (Giving an opinion) between two alternatives for indentation level: two and four spaces. The authors 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.

Bauer et al. (2019) conducted an experiment with 7 developers and 15 students to investigate the influence of levels of indentation (zero, two, four, and eight spaces) on code comprehension. They designed an experiment similar to the one of Miara et al. (1983), with the differences that they asked subjects for the output of code snippets in an open box instead of

providing subjects with multiple choices, and did not ask for a description of code functionality. They assessed code comprehension by (i) the correctness of answers (Trace) and (ii) the rank of perception of task difficulty (Giving an opinion). Differently from the work of Miara et al. (1983), this experiment included eye-tracking to measure visual effort using fixations, which occur when the gaze is resting on a point, and saccades, which are the transition between two fixations. Furthermore, it measured the time subjects spent providing an answer. The code snippets in this experiment were in Java. The authors first applied Mauchly's sphericity test to evaluate which test was more adequate for each dependent variable, i.e., one-way ANOVA with repeated measures or Friedman's test. The results of this experiment were: the correctness of answers (Friedman's test, $p = 0.36$), the log-transformed response time (ANOVA, $p = 0.72$), perceived difficulty (Friedman's 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, the authors could not confirm this difference with a post hoc test. Therefore, unlike in the previous study, there was no statistically significant difference between indentation levels.

Appropriate use of indentation with blocks. In the work by Langhout and Aniche (2021), a partial replication of the study presented by Gopstein et al. (2017) (further explained in Section 4.3) was conducted, focusing on the Java language instead of C/C++. These papers measure the impact of atoms of confusion in code understandability. The authors recruited 132 novice developers for an experiment conducted in two parts, in which participants had to (i) predict the output of code snippets with and without atoms (Trace) and (ii) give their opinions about which ones are confusing among two functionally-equivalent code snippets, one with an atom of confusion and another without (Giving an opinion). For the first part, the authors calculated the odds ratio of subjects correctly predicting the output of the code snippets as a measure of effect size. Two atoms of confusion related to inappropriate use of indentation were investigated: the Remove Indentation atom and the Indentation atom. In both cases, there exist indented statements after the end of a block, which makes it look like the statements are part of the block. In the latter case, the incorrectly indented statement is preceded by curly braces, whereas in the former, it is not. The authors found out that Remove Indentation is associated to misunderstanding (odds ratio $= 56.21$, $p < 0.05$), and for 57.1% of the subjects, this atom is unfavorable for legibility. There was no statistical difference in answers for Indentation, although 72.7% of the subjects considered it confusing.

Vertical and horizontal spacing. Love (1977) evaluated 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. In the paragraphed version, the source code contains blank lines between different instructions and indentation to represent blocks. Unlike, there are no blank lines or indentation in the unparagraphed version. The author experimented with 19 undergraduate and 12 graduate students 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 correctly recalled in the proper order. The descriptions of the program functionality were rated with a 5-point scale, where one point means that a description has nothing right about the program functionality and five points means that a description shows a complete understanding of it. The author did not find

a statistically significant difference between paragraphed and unparagraphed programs, considering the correctness scores in the Memorize activity (ANOVA, $p = 0.79$) and in the Present activity (ANOVA, $p = 0.6$) for both undergraduate and graduate students.

Vertical spacing between related instructions. 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 did not show a statistically significant difference (Two-tailed test for proportion, $p = 1.0$) between subjects who agree and disagree with this practice.

Blank space around operators and parameters. Sampaio and Barbosa (2016) investigated the importance of teaching a set of best practices for code understandability. For this, the authors conducted a survey to ask object-oriented programming teachers to provide a subjective rating on a 5-point Likert scale about the importance of coding practices for code understandability (Giving an opinion). The authors considered that practices with a median rating greater than 3 are relevant to code understandability. Four practices related to legibility were investigated, among others. The practices related to spacing, which are the ones relevant to this section, are about blank space around operators and arguments/parameters: *"The assignment operator include blank space before and after"* and *"Blank space between the arguments/parameters of functions"*. The authors found out that the first (*median = 3, range = 4, interquartile = 2*) and the second (*median = 2, range = 4, interquartile = 2*) practices are not relevant for legibility.

4.3. Block delimiters

This group gathers studies about different types of block/scope delimiters. Table 7 presents a summary of the analyzed results.

Block delimiter location. Miara et al. (1983) compared two styles of block indentation: (i) blocked, where the code within a block is at the same indentation level as the block delimiters (begin-end), and (ii) non-blocked, where the code within the block appears at least one more indentation level to the right of the block delimiters. This comparison is different from what has been previously reported about indentation (Section 4.2) because this one evaluated when the indentation of the block starts (i.e., in the line of the block delimiter itself or in the first statement after the block delimiter), 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 blocked and non-blocked styles.

Arab (1992) compared three combinations of the use of block delimiters in the same line as their associated statements with the use of block delimiters in a separate line. More specifically, 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 study consisted of an opinion survey with 30 subjects (22 novices and 8 experts), who were asked to classify the three schemes in descending order (Giving an opinion). The results showed that the block delimiters in their own lines, 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 their corresponding statements (e.g., class declaration). They found a statistically significant difference (Two-tailed test for proportion, $p = 0.006$) between subjects who preferred the first alternative and subjects who preferred the second alternative, where most participants consider that opening braces in their own lines are easier to read, similar to results of Arab (1992).

Block delimiter style. In the study reported by Sykes et al. (1983), different styles of scope delimiters in Pascal were investigated: (i) ENDIF, which uses `IF-ENDIF` and `WHILE-ENDWHILE` scope delimiters for the bodies of conditional and iterative statements, respectively; (ii) REQ-BE, which always uses `BEGIN-END` scope delimiters for simple and compound statements within a conditional structure; and (iii) BE, which uses `BEGIN-END` scope delimiters only for compound statements. The study employed a questionnaire where most of the questions asked subjects to determine the values of variables by simulating the execution of program segments in Pascal with initial values given in the questions (Trace). 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 two years of programming experience) and intermediate. The authors 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 $\alpha = 0.1$) showed that the subjects performed better using the ENDIF delimiters than using the REQ-BE ($p = 0.074$) and BE ($p = 0.073$) delimiter styles.

Block delimiter visibility. Also, in the work of Sykes et al. (1983), we found an evaluation of block delimiter visibility in Pascal, i.e., whether omitted or present block delimiters impact code legibility. We limited the comparison analysis between REQ-BE and BE to evaluate the block delimiter visibility. The details of the study are explained in the previous paragraph. The authors did not find a difference between the REQ-BE and BE delimiter styles (t-test, $p = 0.955$).

The work of Sampaio and Barbosa (2016) (introduced in Section 4.2) evaluated the relevance of the practice "*use { and } to enclose the statements in a loop*" for code legibility, in the context of Java code. They found that this practice is not relevant (*median = 2, range = 4, interquartile = 2*) for legibility.

Gopstein et al. (2017) compared the use of omitted and present block delimiters in the specific context of small C programs. The authors conducted a broad-scoped experiment[8] with 73 programmers with three 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 (~8 lines), where the control version contained a single atom of confusion candidate, and the treatment version contained a functionally-equivalent version where the atom does not exist. For each program, the participants had to predict its output (Trace). The authors compared the correctness of the answers of the version with the atom of confusion and their counterparts without the atoms. One of 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

---

[8] 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.

Table 7
Summary of results for the Block Delimiters group.

Factor – Study: *Levels (Programming Language)* – Dependent Variables (Activities): Results

Block delimiter location

Miara et al. (1983): *blocked and non-blocked (Pascal)*
  Correctness (Trace, Inspect, Present): No significant difference.
  Opinion: No significant difference.
Arab (1992): *in the statement line and in separate line (Pascal)*
  Opinion: Block delimiters in their own lines is the best.
Santos and Gerosa (2018): *in the statement line and in separate line (Java)*
  Opinion: Block delimiter in its own line is the best.

Block delimiter style

Sykes et al. (1983): *ENDIF/ENDWHILE (without BEGIN) and BEGIN-END in all blocks (Pascal)*
  Correctness (Trace): ENDIF/ENDWHILE is the best.
  Opinion: ENDIF/ENDWHILE is the best.
Sykes et al. (1983): *ENDIF/ENDWHILE (without BEGIN) and BEGIN-END only in compound statement blocks (Pascal)*
  Correctness (Trace): ENDIF/ENDWHILE is the best.
  Opinion: ENDIF/ENDWHILE is the best.

Block delimiter visibility

Sykes et al. (1983): *omitted and present (Pascal)*
  Correctness (Trace): No significant difference.
  Opinion: No significant difference.
Sampaio and Barbosa (2016): *omitted and present (Java)*
  Opinion: No significant difference.
Gopstein et al. (2017): *omitted and present (C/C++)*
  Correctness (Trace): Present block delimiters is the best.
Medeiros et al. (2019): *omitted and present (C)*
  Opinion: Present block delimiters is the best.
Langhout and Aniche (2021): *omitted and present (Java)*
  Correctness (Trace): Present block delimiters is the best.

---

analyzing programs with omitted curly braces made more mistakes. Langhout and Aniche (2021) (introduced in Section 4.2) also evaluated the atom *Omitted Curly Braces*, but in Java code. They found a statistically significant difference (odds ratio = 4.62, $p < 0.05$), indicating that omitting curly braces is associated with misunderstanding. Furthermore, 93.3% of the subjects agreed that a code snippet with this atom is hard to read.

Similarly, Medeiros et al. (2019) investigated 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 the negative or positive impact of using one code snippet version containing 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 remove several misunderstanding patterns. This study focused on the C language. In this section, we analyze the results of the misunderstanding pattern named *Dangling Else*, which happens when an `if` statement without braces contains one `if/else` statement without braces. The reader of the code 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 instances of the *Dangling Else* pattern, where two of them were accepted, one was rejected, three were not answered, and four were ignored by developers (according to the authors, due to the pattern instances being in third party or deprecated code).

4.4. Long or complex code line

This group comprises studies that evaluate elements related to the size and complexity of code lines. Table 8 presents a summary of the analyzed results.

Line length. In the study of Santos and Gerosa (2018) (details about the methodology are in Section 4.2), the subjects were asked if they agree that the practice "*line lengths not exceeding 80 chars*" has a positive impact on code legibility. The authors found out that students and professionals find code snippets containing lines of up to 80 characters more legible than code snippets where there are longer lines (Two-tailed test for proportion, $p < 0.001$).

Statements per line. Three studies tackle the question of whether lines of code including a single statement are more legible than ones with multiple statements. Sampaio and Barbosa (2016) evaluated the relevance of the practice of *"breaking the line after semicolon"* for code legibility (the methodology is detailed in Section 4.2). The authors found that this practice is relevant for code legibility (*median = 4, range = 3, interquartile = 1.5*).

Santos and Gerosa (2018) found out that students and professionals agreed with the practice "*avoid multiple statements on a same line*" (Two-tailed test for proportion, $p < 0.001$), i.e., the subjects prefer code snippets where lines of code do not include multiple statements (details about the methodology are in Section 4.2). As discussed before, the paper of Medeiros et al. (2019) (Section 4.3) introduced and evaluated 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 one pull request to remove this pattern, which was rejected by developers.

4.5. Word boundary styles

The fifth and last group we have identified comprises studies of word boundary styles for identifiers. Table 9 presents a summary of the analyzed results.

Identifier style. Sharif and Maletic (2010) investigated the effect of the camel case and underscore identifier styles on code legibility. They compared the identifier styles using phrases formed by two and three words from code, i.e., phrases that are likely to be in source code (e.g., "start time"), and non-code, i.e., phrases that are not likely to be in source code (e.g., "river bank"). In the experiment, 15 students had to read a phrase and, on the next screen, choose an identifier (from four choices) that exactly matches the phrase they just saw (Memorize), and answer it verbally. Only one of the choices is correct, and the rest are distracters that change the beginning, middle, or end of the identifier. It was repeated for eight phrases. The authors measured the correctness, time, and visual effort for each phrase. They found no difference between identifier styles for correctness. Using a simple linear mixed model, they found a significant difference ($p = 0.0001$) in time where subjects took 13.5% longer for camel-cased identifiers than underscored. For visual effort, they employed two metrics: fixation rate and average fixation duration. They were measured when the subjects looked at the correct answer and the distracters. Using Wilcoxon test, they only found a significant difference in the average fixation duration (correct: $p = 0.015$; distracters: $p = 0.026$), where the distribution showed that camel-cased identifiers required a higher average duration of fixations than underscored.

Binkley et al. (2013) 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

Table 8
Summary of results for the Long or Complex Code Line group.

Factor – Study: *Levels (Programming Language)* – Dependent Variables (Activities): Results
Line length
Santos and Gerosa (2018): *limit of 80 characters and exceed of 80 characters (Java)*
Opinion: Line length within the limit of 80 characters is the best.
Statements per line
Sampaio and Barbosa (2016): *multiple and one (Java)*
Opinion: One statement per line is relevant for code legibility.
Santos and Gerosa (2018): *multiple and one (Java)*
Opinion: One statement per line is the best.
Medeiros et al. (2019): *multiple and one (C)*
Opinion: No significant difference.

Table 9
Summary of results for the Word Boundary Styles group.

Factor – Study: *Levels (Programming Language)* – Dependent Variables (Activities): Results
Identifier style
Sharif and Maletic (2010): *camel case and underscore (None)*
Correctness (Memorize): No significant difference.
Time (Memorize): Underscore is the best.
Visual Metrics (Memorize): Underscore is the best for average fixation duration.
Binkley et al. (2013): *camel case and underscore (C and Java)*
Correctness (Inspect, Memorize, Present): Camel case is the best only in the Inspect activity.
Time (Inspect, Present): Camel case is the best only in the Inspect activity.
Visual Metrics (Memorize): Camel case is the best.

study, and Read Aloud study. In the first study, 135 programmers and non-programmers attempted to find all occurrences of a particular identifier in code fragments (Inspect) 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-cased identifiers, to an estimated 0.24 fewer missed lines than with underscored ones. For the time, it was marginally significant ($p = 0.0692$) and indicated that camel-cased identifiers took, on average, 1.2 fewer seconds to be found than underscored ones.

In the second study, they asked 15 programmers (undergraduate and graduate students) to study two C++ code snippets, reproduce them, and answer questions by selecting the identifiers they remember from the code (Memorize). The authors considered the correctness of the answers. They also used an eye tracker to measure eye fixations and gaze duration, which is the duration of a fixation within a specific area of interest (boxes around the identifiers with some extra padding). 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* required significantly more fixations than *colSum* (1-tailed, $p = 0.007$), and that its average gaze duration was 704ms longer than that of *colSum* (1-tailed, $p = 0.005$). 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, $p = 0.3048$).

4.6. Addressing the two research questions

The two research questions that this work aims to answer are:

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

RQ2 Which levels of formatting elements have been found to make the source code more legible?

Based on the results of our study, for RQ1, we have identified 15 scientific papers in which researchers compared alternative levels of formatting elements with human subjects. We found 13 factors (i.e., formatting elements), which are about code formatting (e.g., formatting layout), code spacing (e.g., indentation), block delimiters (e.g., block delimiter location), long or complex code lines (e.g., line length), and word boundary styles (i.e., identifier style). For the 13 factors, researchers examined 33 different levels (e.g., 2 and 4 levels of indentation) in 27 comparisons (e.g., one comparison considered 0, 2, 4, and 6 indentation levels).

For RQ2, researchers found statistically significant results in 17 comparisons and no significant results in 10 comparisons. Considering the significant differences, it was found that the best alternative levels for four factors were: book format style (cf. Lightspeed Pascal and Kernighan & Ritchie styles), appropriate use of indentation with blocks (cf. inappropriate use), the use of `ENDIF/ENDWHILE` for block delimiters (cf. the use `BEGIN-END` in all blocks and the use `BEGIN-END` for compound statement blocks), and the practice that line lengths should be kept within the 80-character limit (cf. line lengths that exceed such a limit). In other comparisons where significant differences were found, there were divergent results. In one case, more than one comparison of the same factor found statistically significant results but in favor of different levels. This is the case for the factor identifier style, for which one study found that underscore is the best alternative for identifier style while another study found that camel case is better than underscore. In other cases of divergent results, which involved four factors, some comparisons found significant differences in favor of some level, but other comparisons found no significant differences. This is the case for the factors indentation, block delimiter location, block delimiter visibility, and statements per line. Finally, for four factors (i.e., formatting layout, vertical and horizontal spacing, vertical spacing between related instructions, and blank space around operators and parameters), no statistically significant results were found for their levels.

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. Contrasting empirical results with existing coding style guides

In this section, we discuss the results of this study in the light of five existing style guides for Java (Google Java Style Guide[9]),

JavaScript (AirBNB JavaScript Style Guide[10]), Python (Style Guide for Python Code[11]), C (Linux Kernel Coding Style[12]), and Pascal (Free Pascal[13]), and contrast their recommendations against the findings of this study.

Formatting. We found three comparisons of formatting styles, 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, one of these results is from a study that used Pascal, which is not a popular programming language in 2022[14,15], 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 coding style guide supports the use of the Book Format style.

Spacing. Most studies that evaluated spacing did not produce statistically significant results. The results of one study (Love, 1977) 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 5.2). Two studies (Santos and Gerosa, 2018; Bauer et al., 2019) did not find a significant difference between different indentation levels, e.g., two versus four spaces, in Java code. An older study (Miara et al., 1983) comparing zero, two, four, and six spaces in Pascal code snippets found that two spaces exhibited the best result. The coding style provided by Free Pascal recommends two spaces for indentation. Although Miara et al. (1983)'s finding is for Pascal, it is also consistent with some coding guides such as the Google Java Style Guide, the AirBNB JavaScript Style Guide, and the Style Guide for Python Code. Others, such as the Linux Kernel Coding Style, 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 comparisons related to block delimiters (7/10 comparisons) showed statistically significant results. They evaluated the location, visibility, and style of block delimiters. The studies (in Pascal, C, C++, and Java) suggest that block delimiters are relevant to legibility, and should be visible and stay in their own line. The Google Java Style Guide is consistent with this result, suggesting the use of braces even when the block is empty or contains only a single statement (block delimiter visibility), but not about the placement of braces (block delimiter location). The Linux Kernel Coding Style agrees with this for functions but disagrees for statements. In addition, the AirBNB JavaScript Style Guide 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 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

9 https://google.github.io/styleguide/javaguide.html, last access May 22, 2023.

10 https://github.com/airbnb/javascript, last access May 22, 2023.

11 https://peps.python.org/pep-0008/, last access May 22, 2023.

12 https://www.kernel.org/doc/html/v4.10/process/coding-style.html, last access May 22, 2023.

13 https://wiki.freepascal.org/Coding_style, last access May 22, 2023.

14 https://www.tiobe.com/tiobe-index/, last access May 22, 2023.

15 https://redmonk.com/sogrady/2022/03/28/language-rankings-1-22/, last access May 22, 2023.

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. One study investigated and found that keeping line lengths within 80 characters in Java is considered positive for legibility (Santos and Gerosa, 2018). On the other hand, Sampaio and Barbosa (2016), Santos and Gerosa (2018), and Medeiros et al. (2019) investigated the use of one vs. multiple statements per line and obtained different results. Sampaio and Barbosa (2016) and Santos and Gerosa (2018) found out that one statement per line in Java is preferred by the participants, but Medeiros et al. (2019) did not find a significant difference between one or more statements per line in C. In these three studies, only the subjects' opinions were used as the dependent variable. The Style Guide for Python Code and Linux Kernel Coding Style agree with the 80-character limit. The Google Java Style Guide recommends one statement per line but recommends 100-character lines. The AirBNB JavaScript Style Guide also establishes 100 characters as the limit for lines, but it makes no prescription about the number of statements per line.

Word boundary styles. The two studies that investigated word boundary styles for identifier names revealed divergent results. While Binkley et al. (2013) found that camel case is a positive standard for legibility in comparison to snake case (a.k.a. underscore) in Java and C, Sharif and Maletic (2010) found that snake case is the best alternative (no specific programming language was considered). The camel case style 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 Kernel Coding Style explicitly argues against camel case and prescribes the use of snake case.

5.2. 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 a certain level of statistical power is called power analysis. It depends on the significance criterion, the sample size, and the population effect size.

For nine 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 Gopstein et al. (2017) and Bauer et al. (2019) 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. This is 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. It attempted 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 compared 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.

5.3. Limitations of our study and of existing studies

The studies analyzed in this work were conducted between 1977 and 2021 and covered several formatting elements. Nevertheless, there are other aspects beyond what we captured. 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, aspects that fall outside what can be tinkered with at the source code level, in an ASCII text editor, are 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. This study highlights the limitations in our current understanding of the impact of formatting elements on code legibility. We have found only 15 papers containing direct comparisons between alternative levels of formatting elements and styles out of 4,914 examined documents. Replications, even partial ones, are almost non-existent. This suggests that this area of research is **immature** and can greatly benefit from new studies.

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 identified only one paper that reports on the comparison of different types of block delimiters (Sykes et al., 1983), and that paper was published 40 years ago, at a time when software development was very different from what it is today. Another example is indentation. Although many style guides provide directives about how to indent code, most of the studies on the topic found inconclusive results (Section 4.2). Notwithstanding, the importance of the topic for software developers in practice emphasizes that more investigation is required. 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 (Stefik and Siebert, 2013; Gopstein et al., 2017, 2018; Medeiros et al., 2019) suggests that they do, though, even in very mature, high-complexity projects (Gopstein et al., 2018).

Some of the studies we analyzed are **outdated**. These 5 papers were published 30 or more years ago. Software development was very different back then. For example, object-oriented programming was a relative novelty, the world wide web had been public for just one year, and mobile devices did not exist for the general public. In addition, the main programming language examined by these old studies, Pascal, is rarely used in practice nowadays. These studies investigated aspects that are relevant to contemporary software development, such as indentation and the use of block delimiters. However, they do that in a scenario that

is disconnected from current practice and software development methods and tools.

The learning activities involved in the analyzed studies (Section 2.3) required participants to Memorize, Inspect, Trace, Present, and Relate, plus the task of Giving an opinion. The learning activities were modeled in two semi-independent dimensions by Fuller et al. (2007) and extended by Oliveira et al. (2020). Each dimension defines hierarchical linear levels where a deeper level requires the competencies from the previous ones. This means that some activities require more competencies than others. The majority of the comparisons of formatting element levels we found in the literature were based on the Giving an opinion task, which is not modeled because it is not a learning activity. Trace is the most used learning activity, but it does not even require an intermediate level of competencies. The activity used in the analyzed studies that requires more competencies is Relate, which was used for only one formatting element evaluation. More generally, the analyzed studies focused on "lower" cognitive skills according to the model of Fuller et al. (2007), emphasizing their **narrow scope**. Program comprehension aims to support other activities such as performing maintenance, fixing bugs, and writing tests. The absence of studies requiring participants to conduct these activities points to a possible direction for future work.

Since our ultimate goal is to create a coding style guide based on empirical evidence, an analysis of the programming languages used in the experiments of the primary studies included in our review is necessary. Five studies used the Java language (Sampaio and Barbosa, 2016; Siegmund et al., 2017; Santos and Gerosa, 2018; Bauer et al., 2019; Langhout and Aniche, 2021). Another five studies used the Pascal language (Love, 1977; Miara et al., 1983; Sykes et al., 1983; Arab, 1992; Furman et al., 2002). Two more targeted the C or C++ languages (Gopstein et al., 2017; Medeiros et al., 2019). The paper by Oman and Cook (1990) used the languages C and Pascal, and the paper by Binkley et al. (2013) used the languages C and Java. Sharif and Maletic (2010) did not specify a programming language. This characteristic of the studies makes it impossible to generalize the results of our study and emphasizes that very few languages have been targeted by previous work. For example, none of these studies examined a scripting language, such as Python or JavaScript, a functional programming language, e.g., Haskell or Elixir, or even more contemporary multiparadigm languages that receive strong influence from functional languages, e.g., Kotlin or Swift.

6. Threats to validity

In this section, we discuss 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. 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. 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. Finally, comparing different studies poses some threats due to the many differences in their setups, such as materials, programming languages, number of subjects, the subjects' expertise, and tasks to be performed by the subjects. It is not feasible to consider all differences in the studies' setups to compare the studies; some of them do not even provide fine-grained information, such as the size of source code snippets used in the experiments. In our study, however, we mitigated the threat of directly comparing studies by considering the activities performed by subjects in the experiments (which require different levels of cognitive skills), the dependent variables of the studies, the programming languages, the number of subjects, and statistical power.

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

7. Conclusion

In this paper, through a systematic review of the literature, we investigated what formatting elements have been studied and which levels of formatting elements were found to have a positive impact on code legibility when compared to functionally equivalent ones in human-centric studies. We present a comprehensive categorization of the elements and levels found. In addition, we analyzed the results considering the subjects, activities, dependent variables, and programming languages involved in the studies.

We identified 13 factors of formatting elements, which were categorized into five groups: formatting, spacing, block delimiters, long or complex code lines, and word boundary styles. For four factors, the results showed that the levels book format style, appropriate use of indentation with blocks, the use of `ENDIF/ENDWHILE` for block delimiters, and the practice that line lengths should be kept within the limit of 80 characters exhibited positive impact on code legibility. Also, for other four factors, some studies found significant results while others did not find. Furthermore, there were contradictory results in the best alternative level for one factor. Finally, no statistically significant results were found for other four factors.

Although we found some positive results, there are only a few suggestions that can be made to developers based on our findings. The limitations of the literature (i.e., few studies, narrow-in-scope studies, outdated studies, and inconclusive results) and, consequently, of our work, limit us toward our bigger goals: to provide developers with guidelines so that they can choose the best ways to format their code, and to help other researchers in developing automated support, e.g., linters and recommendation systems, to aid developers during programming activities and make their code more legible. Our systematic literature review allowed us to identify and discuss those limitations in this paper. Finally, these limitations call for more research and replication studies because there is much to be understood about how formatting elements influence code legibility before the creation of guidelines and automated aids to help developers.

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

Acknowledgments

CRediT authorship contribution statement

Delano Oliveira: Conceptualization, Methodology, Investigation, Writing – original draft, Writing – review & editing. **Reydne Santos:** Conceptualization, Methodology, Investigation, Writing – original draft, Writing – review & editing. **Fernanda Madeiral:** Conceptualization, Methodology, Investigation, Writing – original draft, Writing – review & editing, Supervision. **Hidehiko Masuhara:** Writing – review & editing. **Fernando Castor:** Conceptualization, Methodology, Investigation, Writing – original draft, Writing – review & editing, Supervision.

Data availability

The data used in our study is available on GitHub.

A Systematic Literature Review on the Impact of Formatting Elements on Program Legibility (Original data) (GitHub)

References

Ajami, S., Woodbridge, Y., Feitelson, D.G., 2019. Syntax, predicates, idioms – what really affects code complexity? Empir. Softw. Eng. 24 (1), 287–328. http://dx.doi.org/10.1007/s10664-018-9628-3.

Almeida, J.R.d., Camargo, J.B., Basseto, B.A., Paz, S.M., 2003. Best Practices in Code Inspection for Safety-Critical Software. IEEE Softw. 20 (3), 56–63. http://dx.doi.org/10.1109/MS.2003.1196322.

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

Arab, M., 1992. Enhancing program comprehension: Formatting and documenting. ACM SIGPLAN Notices 27 (2), 37–46. http://dx.doi.org/10.1145/130973.130975.

Bauer, J., Siegmund, J., Peitek, N., Hofmeister, J.C., Apel, S., 2019. 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, pp. 154–164. http://dx.doi.org/10.1109/ICPC.2019.00033.

Benander, A.C., Benander, B.A., Pu, H., 1996. Recursion vs. Iteration: An empirical study of comprehension. J. Syst. Softw. 32 (1), 73–82. http://dx.doi.org/10.1016/0164-1212(95)00043-7.

Binkley, D., Davis, M., Lawrie, D., Maletic, J.I., Morrell, C., Sharif, B., 2013. The impact of identifier style on effort and comprehension. Empir. Softw. Eng. 18 (2), 219–276. http://dx.doi.org/10.1007/s10664-012-9201-4.

Binkley, D.W., Davis, M., Lawrie, D.J., Morrell, C., 2009. To CamelCase or Under_score. In: Proceedings of the 17th International Conference on Program Comprehension (ICPC '09). IEEE Press, Piscataway, NJ, USA, pp. 158–167. http://dx.doi.org/10.1109/ICPC.2009.5090039.

Blinman, S., Cockburn, A., 2005. Program comprehension: Investigating the effects of naming style and documentation. In: Proceedings of the 6th Australasian User Interface Conference (AUIC '05). ACS, Newcastle, Australia, pp. 73–78.

Bloom, B., Engelhart, M., Furst, E., Hill, W.H., Krathwohl, D.R., 1956. Taxonomy of Educational Objectives: The Classification of Educational Goals. Handbook I: Cognitive Domain. David McKay Company, New York.

Brooks, R., 1978. Using a Behavioral Theory of Program Comprehension in Software Engineering. In: Proceedings of the 3rd International Conference on Software Engineering (ICSE '78). pp. 196–201.

Buse, R.P.L., Weimer, W.R., 2010. Learning a Metric for Code Readability. IEEE Trans. Softw. Eng. 36 (4), 546–558. http://dx.doi.org/10.1109/TSE.2009.70.

Chance, B., Zhuang, Z., UnAh, C., Alter, C., Lipton, L., 1993. Cognition-activated low-frequency modulation of light absorption in human brain.. Proc. Natl. Acad. Sci. 90 (8), 3770–3774. http://dx.doi.org/10.1073/pnas.90.8.3770.

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

Cohen, J., 1992. Statistical Power Analysis. Curr. Direct. Psychol. Sci. 1 (3), 98–101. http://dx.doi.org/10.1111/1467-8721.ep10768783.

Crider, J.E., 1978. Structured Formatting of Pascal Programs. ACM Sigplan Notices 13 (11), 15–22. http://dx.doi.org/10.1145/953777.953779.

Daka, E., Campos, J., Fraser, G., Dorn, J., Weimer, W., 2015. 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, pp. 107–118. http://dx.doi.org/10.1145/2786805.2786838.

Dolado, J.J., Harman, M., Otero, M.C., Hu, L., 2003. An Empirical Investigation of the Influence of a Type of Side Effects on Program Comprehension. IEEE Trans. Softw. Eng. 29 (7), 665–670. http://dx.doi.org/10.1109/TSE.2003.1214329.

DuBay, W.H., 2004. The Principles of Readability. Online Submiss..

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

Feitelson, D.G., 2022. Considerations and pitfalls for reducing threats to the validity of controlled experiments on code comprehension. Empir. Softw. Eng. 27 (6), http://dx.doi.org/10.1007/s10664-022-10160-3.

Fuller, U., Johnson, C.G., Ahoniemi, T., Cukierman, D., Hernán-Losada, I., Jackova, J., Lahtinen, E., Lewis, T.L., Thompson, D.M., Riedesel, C., Thompson, E., 2007. Developing a Computer Science-specific Learning Taxonomy. ACM SIGCSE Bull. 39 (4), 152–170. http://dx.doi.org/10.1145/1345375.1345438.

Furman, S., Boehm-Davis, D.A., Holt, R.W., 2002. A Look at Programmers Communicating through Program Indentation. J. Wash. Acad. Sci. 88 (2), 73–88.

Gopstein, D., Iannacone, J., Yan, Y., DeLong, L., Zhuang, Y., Yeh, M.K.-C., Cappos, J., 2017. 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, pp. 129–139. http://dx.doi.org/10.1145/3106237.3106264.

Gopstein, D., Zhou, H.H., Frankl, P., Cappos, J., 2018. 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, pp. 281–291. http://dx.doi.org/10.1145/3196398.3196432.

Gough, P.B., Tunmer, W.E., 1986. Decoding, Reading, and Reading Disability. Remedial Special Edu. 7 (1), 6–10. http://dx.doi.org/10.1177/074193258600700104.

Gustafson, G., 1979. Some Practical Experiences Formatting Pascal Programs. ACM Sigplan Notices 14 (9), 42–49. http://dx.doi.org/10.1145/988113.988118.

Hofmeister, J.C., Siegmund, J., Holt, D.V., 2019. Shorter identifier names take longer to comprehend. Empir. Softw. Eng. 24 (1), 417–443. http://dx.doi.org/10.1007/s10664-018-9621-x.

Hoover, W.A., Gough, P.B., 1990. The simple view of reading. Read. Writ. 2 (2), 127–160. http://dx.doi.org/10.1007/BF00401799.

Jbara, A., Feitelson, D.G., 2014. 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, pp. 189–200. http://dx.doi.org/10.1145/2597008.2597140.

Johnson, M., Beekman, G., 1988. Oh! Thinks Lightspeed Pascal!. WW Norton.

Keele, S., et al., 2007. Guidelines for Performing Systematic Literature Reviews in Software Engineering. Technical Report, Version 2.3 EBSE.

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

Langhout, C., Aniche, M., 2021. Atoms of Confusion in Java. In: Proceedings of the 29th IEEE/ACM International Conference on Program Comprehension (ICPC '21). IEEE, pp. 25–35. http://dx.doi.org/10.1109/ICPC52881.2021.00012.

Love, T., 1977. An Experimental Investigation of the Effect of Program Structure on Program Understanding. In: ACM SIGOPS Operating Systems Review – Proceedings of An ACM Conference on Language Design for Reliable Software, Vol. 11. ACM, New York, NY, USA, pp. 105–113. http://dx.doi.org/10.1145/390018.808317.

Medeiros, F., Lima, G., Amaral, G., Apel, S., Kästner, C., Ribeiro, M., Gheyi, R., 2019. An investigation of misunderstanding code patterns in C open-source software projects. Empir. Softw. Eng. 24 (4), 1693–1726. http://dx.doi.org/10.1007/s10664-018-9666-x.

Miara, R.J., Musselman, J.A., Navarro, J.A., Shneiderman, B., 1983. Program Indentation and Comprehensibility. Commun. ACM 26 (11), 861–867. http://dx.doi.org/10.1145/182.358437.

Oliveira, D., Bruno, R., Madeiral, F., Castor, F., 2020. Evaluating Code Readability and Legibility: An Examination of Human-centric Studies. In: Proceedings of the 36th IEEE International Conference on Software Maintenance and Evolution (ICSME '20). IEEE, pp. 348–359. http://dx.doi.org/10.1109/ICSME46990.2020.00041.

Oman, P.W., Cook, C.R., 1990. Typographic Style is More than Cosmetic. Commun. ACM 33 (5), 506–520. http://dx.doi.org/10.1145/78607.78611.

O'Neal, M.B., Edwards, W.R., 1994. Complexity Measures for Rule-Based Programs. IEEE Trans. Knowl. Data Eng. 6 (5), 669–680. http://dx.doi.org/10.1109/69.317699.

Peterson, J.L., 1977. On the Formatting of Pascal Programs. ACM Sigplan Notices 12 (12), 83–86. http://dx.doi.org/10.1145/954618.954624.

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

Sampaio, I.B., Barbosa, L., 2016. Software readability practices and the importance of their teaching. In: Proceedings of the 7th International Conference on Information and Communication Systems (ICICS '16). IEEE, pp. 304–309. http://dx.doi.org/10.1109/IACS.2016.7476069.

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

Scanniello, G., Risi, M., 2013. 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, pp. 190–199. http://dx.doi.org/10.1109/ICSM.2013.30.

Schulze, S., Liebig, J., Siegmund, J., Apel, S., 2013. 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, pp. 65–74. http://dx.doi.org/10.1145/2517208.2517215.

Sharif, B., Maletic, J.I., 2010. An Eye Tracking Study on camelCase and under_score Identifier Styles. In: Proceedings of the 18th IEEE International Conference on Program Comprehension (ICPC '10). IEEE Computer Society, Washington, DC, USA, pp. 196–205. http://dx.doi.org/10.1109/ICPC.2010.41.

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

Smit, M., Gergel, B., Hoover, H.J., 2011. Code Convention Adherence in Evolving Software. In: Proceedings of the 27th IEEE International Conference on Software Maintenance (ICSM '11). IEEE, pp. 504–507. http://dx.doi.org/10.1109/ICSM.2011.6080819.

Stefik, A., Siebert, S., 2013. An Empirical Investigation into Programming Language Syntax. ACM Trans. Comput. Edu. 13 (4), 19:1–19:40. http://dx.doi.org/10.1145/2534973.

Strizver, I., 2013. Type Rules: The Designer's Guide To Professional Typography. John Wiley & Sons.

Sykes, F., Tillman, R.T., Shneiderman, B., 1983. The Effect of Scope Delimiters on Program Comprehension. Softw. - Pract. Exp. 13 (9), 817–824. http://dx.doi.org/10.1002/spe.4380130908.

Tekfi, C., 1987. Readability formulas: An overview. J. Doc. 43, 261–273. http://dx.doi.org/10.1108/eb026811.

Tenny, T., 1988. Program readability: Procedures versus comments. IEEE Trans. Softw. Eng. 14 (9), 1271–1279. http://dx.doi.org/10.1109/32.6171.

Wood, J.R., Wood, L.E., 2008. Card Sorting: Current Practices and Beyond. J. Usability Stud. 4 (1), 1–6.

Zuffi, S., Brambilla, C., Beretta, G., Scala, P., 2007. Human Computer Interaction: Legibility and Contrast. In: Proceedings of the 14th International Conference on Image Analysis and Processing (ICIAP '07). IEEE, pp. 241–246. http://dx.doi.org/10.1109/ICIAP.2007.4362786.

Delano Oliveira is a Ph.D. candidate at the Federal University of Pernambuco, Brazil, and a professor at the Federal Institute of Pernambuco, Brazil. His main research goal is to understand the best coding alternatives for code legibility and readability to help developers write understandable code.

Reydne Santos is a Ph.D. candidate at the Federal University of Pernambuco, Brazil. His main research interest is productivity in software engineering and source code legibility and readability.

Fernanda Madeiral is an Assistant Professor at the Vrije Universiteit Amsterdam, The Netherlands. Her broad research goal is to help developers produce and maintain high-quality software systems. She is interested in software bugs and their fixes, linter violation, and source code legibility and readability.

Hidehiko Masuhara is the dean of the School of Computing and a Professor of Mathematical and Computing Science, Tokyo Institute of Technology, Japan. His research interest is programming languages, especially on aspect- and context-oriented programming, partial evaluation, computational reflection, meta-level architectures, parallel/concurrent computing, and programming environments.

Fernando Castor is an Assistant Professor at the Department of Information and Computing Sciences, Utrecht University, The Netherlands, and an Associate Professor (on leave) at the Informatics Center of the Federal University of Pernambuco, Brazil. His broad research goal is to help developers build more efficient software systems more efficiently. More specifically, he conducts research in the areas of Software Maintenance, Software Energy Efficiency, and Code Understandability.