Attributes Influencing the Reading and Comprehension of Source Code – Discussing Contradictory Evidence

Talita Vieira Ribeiro and Guilherme Horta Travassos

PESC/COPPE/UFRJ – Systems Engineering and Computer Science Program, Federal University of Rio de Janeiro, Rio de Janeiro, Brazil, 21941-972
{tvribeiro, ght}@cos.ufrj.br

Abstract

Background: Coding guidelines can be contradictory despite their intention of providing a universal perspective on source code quality. For instance, five attributes (code size, semantic complexity, internal documentation, layout style, and identifier length) out of 13 presented contradictions regarding their influence (positive or negative) on the source code readability and comprehensibility. **Aims:** To investigate source code attributes and their influence on readability and comprehensibility. **Method:** A literature review was used to identify source code attributes impacting the source code reading and comprehension, and an empirical study was performed to support the assessment of four attributes that presented empirical contradictions in the technical literature. **Results:** Regardless participants' experience; all participants showed more positive comprehensibility perceptions for Python snippets with more lines of code. However, their readability perceptions regarding code size were contradictory. The less experienced participants preferred more lines of code while the more experienced ones preferred fewer lines of code. Long and complete-word identifiers presented better readability and comprehensibility according to both novices and experts. Comments contribute to better comprehension. Furthermore, four indentation spaces dominated the code reading preference. **Conclusions:** Coding guidelines contradictions still demand further investigation to provide indications on possible confounding factors explaining some of the inconclusive results.

Keywords: Source code quality, Readability, Comprehensibility, Experimental Software Engineering.

1 Introduction

Quality is a complex issue since its perception depends on different people's expectations for a product [1]. Commonly, software quality is assessed through the number of software defects and the effort required with its maintenance. Although this perspective is expected given the high costs involved in the evolution of software [2], many other features can influence the perception of quality regarding software products.

The readability and comprehensibility of source code represent characteristics used to perceive software quality. Such characteristics impact software maintainability since the reading and comprehension of source code are the most time-consuming tasks in software maintenance and evolution, especially in cases in which there is a high turnover of developers and the system's documentation is incomplete or outdated [3] [4]. Besides the amount of effort involved in these tasks, the misunderstandings regarding the software objectives, and, consequently, the source code reconstruction due to a lack of understanding, are usually caused by the low readability and comprehensibility of source code, which represent challenges to requirements conformance and software evolution, leading to rework [3].

Although there is a consensus that readability and comprehensibility are essential characteristics to determine the quality of source code, the features contributing to increasing or decreasing the levels of readability and comprehensibility are still under investigation [4] [5]. Therefore, source code guidelines are usually developed without a proper theoretical and experimental background [6]. For instance, several programming styles aim at providing standards for coding in different programming languages, presenting a list of source code attributes and their referential values (in the form of rules) that should be observed. However, it is common to come across common

attributes, and contradictory thresholds among these standards – the Google Style Guide for Python¹ and Java² present different rules for naming, and the internal style guideline for Python³ developers on Google use presents divergent rules for indentation when compared to the Google Style Guide for Python and the official Python style guide⁴.

The lack of agreement among standards can augment the misalignment regarding the quality of source code among practitioners, leading to its regular reconstructions by different developers over time [3]. The avoidance of source code reconstruction represents a motivation for our work. We observed this issue in one of our collaborations with the Brazilian industry a few years ago. By following an evidence-based approach, we planned and performed a structured literature review to acquire evidence regarding the influence of source code attributes on code reading and comprehension, supporting the assembling and tailoring of coding guidelines aligned with the company's quality goals. However, the identification of contradictory evidence in the technical literature added additional challenges on proposing an evidence-based solution for the company at that moment.

This topic was previously discussed in [7], which presented contradictory evidence concerning the impact of code size, identifier length, and indentation spacing on code readability and comprehensibility. To investigate whether software development experience and domain knowledge would support explaining the influence divergences regarding these quality attributes, we performed an empirical study that turned out to be inconclusive for some cases. Overall, regardless the programming experience, four spaces for indentation dominated the readability preference of participants in the performed empirical study. The readability and comprehensibility preferences towards long and complete-word identifiers were mostly true for both novice/experts developers. Furthermore, while all participants showed more positive comprehensibility perception for Python snippets with more lines of code, their readability perception regarding code size were contradictory. The less experienced participants preferred more lines of code while the more experienced ones preferred fewer. Now, in this extended version, we include further discussions by exploring additional works published in the technical literature in the last years, and by contributing to a broader view of the attributes that influence positively and negatively the readability and comprehensibility of source code, describing additional contextual information on the contradictory evidence identified in the technical literature and in the performed empirical study. We hope they can contribute to future investigations in the field of source code quality.

Therefore, besides this Introduction, this paper is organized into four additional sections. Section 2 presents the literature review extension, as well as an overview of 13 source code attributes⁵ influencing the readability and comprehensibility of source code and their 94 measurement procedures. Still, in this section, we highlight the contradictory results identified through a literature review that led us to plan and execute an empirical study towards observing the role and influence of programming experience and domain knowledge in the revealed contradictions. As we detail in Section 3, the study was planned and executed in the context of a software engineering course with 38 undergraduate students with different levels of expertise and domain knowledge background, which offered us the opportunity of observing whether these factors might support explaining the contradictory results. While programming experience helped us to identify a probable root for the inconsistencies found in the technical literature regarding code size (novices tend to require more verbose code, while experts tend to appreciate compact code while assessing readability) and identifier length (inexperienced programmers had more difficulty in comprehending short identifiers than experts), it did not help us with the explanation for the contradiction for indentation spacing on readability perception. Section 4 discusses the threats to the validity of both studies (literature review and empirical study), Section 5 offers the conclusions, and future works.

2 Contradictory Evidence on Source Code Readability and Comprehensibility

2.1 Readability and Comprehensibility of Source Code

The readability of source code can be defined as a quality characteristic based on the human judgment of how easy it is to read the written code. This feature relates to program elements and their presentation to programmers. Conversely, the comprehensibility of source code is a quality characteristic based on the human judgment of how easy it is to understand the source code. This characteristic relates to the source code interpretation by its reader. This way, a

¹ <https://google.github.io/styleguide/pyguide.html>

² <https://google.github.io/styleguide/javaguide.html>

³ <https://developers.google.com/edu/python/introduction?csw=1>

⁴ <https://www.python.org/dev/peps/pep-0008/#indentation>

⁵ In this paper, attributes refer to different features of a source code entity, such as size of a program unit, meaningfulness of an identifier among, quality of a comment others.

source code can be readable, but not comprehensible and vice-versa, although its readability might affect its comprehensibility. These two concepts are interchangeably used in the technical works due to their close meaning and relation, yet they do not represent the same concept. Different strategies provide ways to support either the source code reading or source code comprehension.

During a scoping review aiming at revealing technical works presenting strategies to enhance or provide support on the source code reading and comprehension, four distinct categories of works were identified. The **first category** regards the structural code analysis and modification. These works (for instance, by Kreimer [8] and Alshayeb [9]) evaluate the program structure as a whole against refactoring techniques and code patterns, assessing the source code comprehensibility and recommending high-level modifications. The **second category** represents the works concerned with providing software visualization support for source code comprehension. In this case, the focus is not the program evaluation/improvement *per se*, but rather its comprehension through its structure visualization [10] [11]. The **third category** organizes the works using colors to highlight the main source code features, especially its control flow, declaration and data input/output structures [12]. Instead of evaluating the source code, these works aim at calling attention to program elements simplifying the reading of source code.

The **fourth category** consists of works evaluating source code in the light of their attributes, identifying their influence on its readability and comprehensibility. Works like [13] [14] [15] [4] [16] assess, through empirical studies, the code size, complexity, commands separation, line length, indentation among others, to identify their influence on the readability and comprehensibility of source code.

Even though the first three categories represent essential works to support the source code reading and comprehension, they do not present alternatives to overcome the reconstruction issue we observed during one of our academia-industry collaborations. Our primary goal was to prevent the rewriting of source code (rework) by the practitioners in the company. Therefore, we needed to identify those source code attributes that could be modified to improve the source code quality and to align their different quality perspectives, focusing on the main types of code commonly reconstructed by the developers: code snippets.

Upon this set of works on readability and comprehensibility of source code, we selected the fourth category as appropriate to support our primary objective of preventing source code reconstruction in the target company. To identify other works in this category and enlarge our dataset, we planned and executed a structured literature review using the fourth category's works as control articles. The literature review was performed in January of 2014 to capture papers up to 2013 and re-executed in June of 2017 to capture works up to 2016. The following results represent the overall outcome of the literature review.

2.2 Literature Review Planning

A structured literature review based on systematic literature review (SLR) practices [17] was undertaken. The decision for performing a less in-depth literature search was based on the time available for executing the task in the context of the academia-industry collaboration. This fact made us simplify some of the SLR steps such as the search strategy and data synthesis – explained in the following subsections. Table 1 presents the **objective** that was set for the performed structured literature review.

Table 1: Structured literature review objective

Analyze for the purpose of with respect to from the point of view of in the context of	source code attributes characterization their influence on readability and comprehensibility of source code developers industrial settings regarding software development and maintenance
---	---

During the literature review, we wanted to answer the following **research questions** that were also used to guide the rest of the planning:

- Which source code attributes are used to evaluate its readability and comprehensibility?
- What are the measurement procedures for these attributes?
- What are the existing relations between the identified attributes and the quality characteristics (readability/comprehensibility)?

2.2.1 Primary Study Search and Selection Strategies

As our first simplification for the structured literature review, we decided to select a single search engine for gathering the primary studies. Based on our previous experiences conducting systematic literature reviews, we chose Scopus as the search engine, since it has presented high coverage of works from the leading conferences and journals in the software engineering field.

To support answering the research questions, we assembled a search string to retrieve specific attributes/measures for source code that would influence the quality characteristics under judgment. Additionally, we used five control articles (Table 2) to gauge the following search string: TITLE-ABS-KEY((metric OR measure OR attribute OR predictor OR evaluation OR assessment OR improvement OR style OR standard OR pattern) AND (readability OR comprehensibility OR understandability OR identifier OR naming OR comment) AND ("software quality" OR "software readability" OR "software comprehension" OR "software understanding" OR "program quality" OR "program readability" OR "program comprehension" OR "program understanding" OR "code quality" OR "code readability" OR "code comprehension" OR "code understanding"))).

Table 2: Control articles identified in the scoping review

Authors	Year	Title
Lawrie, D.; Morrell, C.; Feild, H. & Binkley, D.	2006	<i>What's in a name? A study of identifiers</i>
Lawrie, D.; Morrell, C.; Feild, H. & Binkley, D.	2007	<i>Effective identifier names for comprehension and memory</i>
Buse, R. & Weimer, W.	2008	<i>A metric for software readability</i>
Buse, R. & Weimer, W.	2010	<i>Learning a metric for code readability</i>
Posnett, D.; Hindle, A. & Devanbu, P.	2011	<i>A simpler model of software readability</i>

We did not intend to embrace all the technical literature in the field, but rather to gather a set of attributes and their measurement procedures that would support us assembling some initial evidence-based coding guidelines for the company, preventing its developers from reconstructing source code over time. To suit their development context, we used the categories previously described (Section 2.1) to support the decision for including and excluding papers. We aimed at gathering only papers fitting the **fourth category**. Thus we created the following inclusion and exclusion criteria for the primary studies selection:

- Inclusion Criteria
 - (I1) The paper quality focus includes the readability and/or comprehensibility of source code snippets; AND
 - (I2) The paper presents source code attributes used to assess the quality of source code snippets about its readability and/or comprehensibility; AND
 - (I3) The paper reports evidence of source code attributes influencing the readability and/or comprehensibility of source code snippets.
- Exclusion Criteria
 - (E1) The paper is not related to computer science; OR
 - (E2) The paper focus is not software engineering; OR
 - (E3) The paper quality focus is not the readability and/or comprehensibility of source code snippets; OR
 - (E4) The paper does not present source code attributes used to assess the quality of source code about its readability and/or comprehensibility; OR
 - (E5) The paper is not available; OR
 - (E6) The paper is not written in one of the following languages: English, Spanish or Portuguese; OR
 - (E7) The paper does not report any evidence of source code attributes influencing the readability and/or comprehensibility of source code snippets.

To select a paper for having its data extracted, a researcher should go through all the papers returned and assess their titles and abstracts according to the inclusion and exclusion criteria. In case of doubt on the selection of any paper, a second researcher should re-evaluate it, and if any doubt concerning the paper acceptance would remain, the paper should be included for a thorough reading. Whenever necessary, the papers' introduction and conclusions should be used during the study selection phase to make sure the papers were fulfilling the conditions for their acceptance. In addition, in case any paper was included, but after its reading, the researchers noticed it did not satisfy the conditions for its inclusion in the review, then it should be excluded.

2.2.2 *Data Extraction and Synthesis*

The papers included should have the following information extracted: i) quality characteristics definition; ii) source code quality attributes and their measurement procedures; iii) empirical evaluation type; iv) evaluation description; v) characteristics of the source codes used for the assessment (programming language, application, among others); vi) subjects' characteristics (practitioners or academics) in case applicable; and vii) results from the evaluation.

Based on the control articles identified during the scoping review, we knew it would be hard to aggregate the different included studies results, mainly because while some studies report the same source code attributes, their measurement procedures do not match, making it impossible to combine evidence from different sources. Our idea for the synthesis should then follow a qualitative perspective, in the sense that we should create clusters of similar attributes and then appraise their overall influence on the readability and comprehensibility of source code.

2.2.3 *Quality Assessment*

When following an evidence-based approach in which the results from the technical literature will affect the selection of a software technology in real development scenarios, the quality assessment of the included papers is mandatory. In this work, we identified seven criteria that should be used to assess the included papers. In the end, each paper would receive a score from zero to ten that would be used to determine whether their results would be taken to be discussed with the company's practitioners or not. It is essential to observe that while following an evidence-based approach we had to consider the strength of the evidence as one of the leading items to assess the quality of a study, and for this reason, criterion D is the one that accounts the majority of the assessment score.

- **Quality Appraisal Criteria**
 - (A) Does the paper define the identified quality attributes? (number of attributes with definition/total number of attributes identified in the paper)
 - (B) Does the paper describe the attributes measurement procedures? (number of attributes with measurement procedures/total number of attributes identified in the paper)
 - (C) Are the measurement procedures reproducible? (0 – none of them are; 0.25 – most of them are not; 0.5 – half of them are; 0.75 – most of them are; or 1 – yes, all of them are)
 - (D) What type of empirical study for assessing the influence of the quality attributes on the source code readability and comprehensibility is reported? (1 – proof of concept; 2 – survey or observational study; 3 – case study; 4 – controlled experiment)
 - (E) Are the source codes used as instruments in the empirical studies toy examples or real codes? (0 – toy; 1 – real)
 - (F) Are the participants of the empirical study practitioners or students? (0 – students or not applicable; 1 – practitioners)
 - (G) Are the study results reported in detail? (0 – no; 0.5 – partially yes; 1 – yes)

2.3 **Source Code Attributes Impacting Readability and Comprehensibility**

Overall, 404 papers were returned up to 2016 after re-executing the search string in Scopus. Each paper had its title, and abstract checked according to the inclusion and exclusion criteria (including those that already were included/excluded in the previous literature review presented in [18]), and 24 papers were included for data extraction.

Two papers ([19] and [20]) could not be found. Thus the data we are going to present is concerned with the 22 papers we had access (see Table 3).

Table 3: Papers included for data extraction

ID	REF	Authors	Year	Title
A	[21]	DEYOUNG and KAMPEN	1979	Program factors as predictors of program readability
B	[22]	JøRGENSEN	1980	A methodology for measuring the readability and modifiability of computer programs
C	[23]	WOODFIELD, DUNSMORE, and SHEN	1981	The effect of modularization and comments on program comprehension
D	[24]	MIARA <i>et al.</i>	1983	Program indentation and comprehensibility
E	[25]	TENNY	1988	Program readability: procedures versus comments
F	[26]	TEASLEY	1994	The effects of naming style and expertise on program comprehension
G	[13]	LAWRIE <i>et al.</i>	2006	What's in a name? A study of identifiers
H	[14]	LAWRIE <i>et al.</i>	2007	Effective identifier names for comprehension and memory
I	[15]	BUSE and WEIMER	2008	A metric for software readability
J	[27]	BINKLEY <i>et al.</i>	2009	To camelcase or under_score
K	[4]	BUSE and WEIMER	2010	Learning a metric for code readability
L	[28]	SHARIF and MALETIC	2010	An eye-tracking study on camelCase and under_score identifier styles
M	[29]	BUTLER <i>et al.</i>	2011	Exploring the influence of identifier names on code quality - an empirical study
N	[30]	FEIGENSPAN <i>et al.</i>	2011	Exploring software measures to assess program comprehension
O	[16]	POSNETT, HINDLE, and DEVANBU	2011	A simpler model of software readability
P	[31]	SHARAFI <i>et al.</i>	2012	Women and men - different but equal: on the impact of identifier style on source code reading
Q	[32]	BINKLEY <i>et al.</i>	2013	The impact of identifier style on effort and comprehension
R	[33]	LEE, LEE and IN	2013	A study of different coding styles affecting code readability
S	[34]	SASAKI, HIGO, and KUSUMOTO	2013	Reordering program statements for improving readability
T	[35]	TASHTOUSH <i>et al.</i>	2013	Impact of programming features on code readability
U	[36]	LEE, LEE and IN	2015	Effect analysis of coding convention violations on readability of post-delivered code
V	[5]	BÖRSTLER and PAECH	2016	The role of method chains and comments in software readability and comprehension - An experiment

The works were organized in clusters according to their proximity regarding attributes under evaluation and the primary purpose of the work. Figure 1 depicts the dependency graph among these 22 included papers, and the four main clusters of works returned by our search. As a general comment, most of the works neither provide a precise definition of quality characteristics under investigation (readability and comprehensibility) nor differentiate them,

which hampered the presentation of source code attributes for a specific quality characteristic and led us to indicate their influence on both the readability and comprehensibility.

K [4] is the most cited paper in the group, and the nature of its work can explain it. K intends to identify a way of measuring the readability/comprehensibility of source code using different combinations of source code attributes. The paper also has a high-quality appraisal score, which, among other reasons, indicates it describes in detail the attributes, their measurement procedures and the evidence of their influence on the source code readability/comprehensibility. Likewise, B [22], I [15] and O [16] also propose a metric for source code readability/comprehensibility by combining different source code attribute. That is why they together compose one of the clusters (1) identified in our structured literature review.

A [21], R [33], T [35] and U [36] share some similarities with the first cluster of works. Although they do not intend to propose a metric for measuring the source code readability/comprehensibility, they try to identify, from a big list of source code measures, those that have some influence on the reading and comprehension of source code. For doing so, they measured several source code features in either real or toy projects previously accessed concerning their readability/comprehensibility. After that, they applied either mining or a regression technique to identify those measures that had mostly affected the assessment of readability/comprehensibility. Among all the included papers, this set contains the ones with the lowest quality appraisal scores regarding our evaluation criteria. Although this assessment does not have any external significance, we would not be able to use them to support any final decision, since we could not gather from them the necessary amount of information to answer our research questions.

The cluster containing C [23], E [25] and V [5] (3), differently from the other two clusters, present a very restricted list of source code attributes under evaluation, all of them concerned with comments. Similarly, the cluster containing F [26], G [13], H [14], J [27], L [28], M [29], P [31], Q [32] and S [34] (4) has works that restrict their studies on the influence analysis of different identifier attributes on source code reading/comprehension. In some sense, some papers dealing with specific source code entities (identifier and comment) reflect the importance of these two entities to the reading and comprehension of source code.

Figure 1: Dependency Graph of Included Papers – The Darker the Circle, the More Centralized the Paper is

D [24] and N [30] were not grouped in any cluster given their distance in purpose from the other works. D deals specifically with indentation spacing and N deals with four different measures related to code size, structural and

semantic complexity. Although it looks similar to works in the Cluster 2, it differs regarding how it evaluated the influence of the attributes on source code reading/comprehension – the authors used an experiment instead of mining or regression analysis.

Briefly, all the papers provided some information concerning our research questions, even though not all of them presented detailed information regarding the measurement procedures of the evaluated source code attributes. From the 22 included papers, 13 source code attributes from three source code entities (program unit, identifier, and comment) presented evidence of their influence on the readability and comprehensibility of source code. At least one measurement procedure was found for each attribute, which led to 94 measurement procedures (either qualitative or quantitative). Appendix A presents the complete list of entities, attributes, measurement procedures and impacts on source code readability/comprehensibility gathered and aggregated from the included papers. The list lacks some attributes and measurement procedures since some of the primary studies either did not list all the observed attributes or described some of those they listed without enough information to be understood.

We followed a bottom-up method for identifying the source code attributes and their respective entities. For each of the papers, we first identified the cause-effect model that was being assessed, for instance, we gathered what was being measured in a source code (cause) and what was being observed concerning its impact in the source code quality – readability or comprehensibility – (effect). In addition, we identified how the causes and effects were measured. Not all of the papers made explicit they were evaluating source code attributes, most of them measured different program elements and tried to make a causal relationship between such program elements and the readability/comprehensibility of source code. From a list of 94 measurement procedures (regarding amount/ average/ maximum of lines/ comments/ statements/ operators/ identifiers/ spaces/ characters etc.), we identified 13 source code attributes: six concerning a program unit, six concerning particular features of identifier and one concerning comments (see Figure 2).

One may wonder if it is acceptable to compare/group works from more than 30 years ago with recent ones since the programming languages evolved so much during this time. We should highlight that most of the measurement procedures we identified in the oldest works can also be used in modern programming languages and for this reason, we did not discard them from our included papers set. Moreover, the qualitative aggregation we performed with the results were on the level of attribute, which also allowed us to make a comparison of works that present very different measurement procedures.

Figure 2: The Three Source Code Entities and their Attributes

The measurement procedures regarding program units are concerned with measuring program elements in the whole piece of a program. For instance, counting the number of lines of code in a program unit, counting the number of branches in a program unit, identifying the presence of comments in a program unit, counting the number of spaces for indentation used in a program unit among others. Each of these measurement procedures, although used to measure program elements, serves as surrogates to a program unit attribute. We identified six program unit attributes (details in Appendix A):

- size/length: groups measurement procedures intending to measure spatial dimensions of a program unit;
- structural complexity: groups measurement procedures intending to measure the way the information is organized and connected in the program unit;
- semantic complexity: groups measurement procedures intending to measure the conceptual information embedded in the program unit;
- implementation level: groups measurement procedures intending to measure the level of abstraction of an implementation;
- internal documentation: groups measurement procedures intending to measure the amount of documentation support the program unit has;
- layout style: groups measurement procedures intending to measure the spatial location of program elements in the program unit.

Measurement procedures are related to identifiers concern measuring specific properties of an identifier disregarding their relationship with other program elements in the program unit. We identified six identifier attributes, differently from program unit attributes, most of them are measured qualitatively (details in Appendix A):

- consistency: groups the measurement procedures aiming at measuring the identifier name consistency regarding upper/lower case and underscore use;
- meaningfulness: groups the measurement procedures that intend to measure the identifier name retention in memory;
- length: groups the measurement procedures aiming at measuring the spatial dimensions of an identifier;
- scope: groups measurement procedures intending to measure the identifier boundaries of use;
- multi-word separator style: groups measurement procedures aiming at measuring the way multi-word identifiers have their words separated;
- type encoding style: groups measurement procedures intending to measure the way the identifiers show their type information to the reader.

Similarly to the case of the identifier entity, measurement procedures related to comments regards measuring specific properties of a comment. Just one attribute was identified for comment: quality. In this case, the quality is measured regarding whether the comment presents information that goes beyond what the code provides. It is important to note that the entity program unit also presents measurement procedures related to comments, but they intend to measure the presence, amount, and length of comments in the program unit, meaning no feature of a single comment is assessed concerning its influence on the source code reading/comprehension.

With a variety of measurement procedures, it is not a surprise to encounter contradictory evidence regarding the influence of some source code attributes on its reading/comprehension (rows of positive and negative symbols for each attribute in Figure 2). Still, it is possible to draw some insights from the bigger picture presented in Figure 2, especially from those works presenting confirmatory results. According to our findings, the bigger the structural complexity of a program unit, the worse the source code readability/comprehensibility (negative symbols in Figure 2), also a higher abstraction level of implementation leads to a better source code reading/comprehension. Concerning identifiers; consistency, meaningfulness and multi-word separator style use (especially the use of camel case) presented positive influence on source code reading/comprehension. High-quality comments also influence the quality of source code positively, especially regarding comprehensibility.

Such confirmatory evidence supported their use to align the quality perspectives and, at the same time, to improve the source code quality in the target company, motivating this evidence-based work. On the other hand, the existence of contradictions prevented us from using some of the results, which led us to investigate the contradictions further.

2.4 Contradictory Results Regarding Source Code Attributes

The encountered contradictions are concerned with five attributes: four related to program unit – size/length, semantic complexity, internal documentation, layout style –, and one to the identifier – length. The following subsections present a discussion on them.

2.4.1 Contradictory Behaviors for Program Unit Size/Length

Table 4 presents three different works with divergent results on the impact of source code size on the readability/comprehensibility of source code. Even though the divergent results (positive and negative) are for different measurement procedures, both deal with lines of code which allows us to analyze their results together. DeYoung and Kampen (A) [21] assessed the two ways of measuring the source code. While the total amount of lines did not seem to impact the source code reading/comprehension, the number of lines containing statements showed to have a negative relation to the readability and comprehensibility of source code, meaning the more the lines containing statements, the worse the source code reading/comprehension. Posnett, Hindle, and Devanbu (O) [16] and Feigenspan et al. (N) [30] measured the size using only the total number of lines. While the former identified it as being directly proportional to the source code readability/comprehensibility, the latter reached no conclusive influence on these characteristics. None of the works presented conclusive justifications concerning their findings, which makes unclear the reasons for such divergences. Still, we can conjecture that the number of blank lines, presented in other works (B [22], I [15] and K [4]) as having a positive influence on source code reading/comprehension might have been a confounding factor in these studies. Also, the fact that up to today different procedures and tools measure lines of code differently compromises the actual comparison between the identified works, since we cannot assure how they measured this attribute, leading to another likely confounding factor.

Table 4: Measurement procedures and contradictory behaviors for program unit size

Entity	Attribute	Measurement Procedure	Impact on Source Code Readability/Comprehensibility
Program Unit	Size/Length	Count the number of lines	⊗ A [21] → no impact on source code reading/comprehension was observed concerning the number of lines
			⊗ N [30] → no impact on source code reading/comprehension was observed concerning the number of lines
		Count the number of lines containing statements	⊕ O [16] → the larger the number of lines, the better the source code reading/comprehension
			⊖ A [21] → the larger the number of lines containing statements, the worse the source code reading/comprehension

2.4.2 Contradictory Behaviors for Program Unit Semantic Complexity and Identifier Length

Similarly, the studies in this group of contradictions did not measure program unit semantic complexity nor the identifier length using the same measurement procedures. However, in some sense, their results can be compared. Table 5 presents the works and the real contradictions for these attributes. We can identify three central contradictions in Table 5: i) concerning arithmetic operators; ii) concerning commas; iii) concerning the length of identifiers.

Three different works investigated the impact of arithmetic operators on source code reading/comprehension. While it sounds reasonable that arithmetic expressions can be detrimental to comprehension, since they require extra attention from readers, I [15] and K [4] presented a positive relation between arithmetic operators and source code readability. All the works that investigated this program element applied mining/regression techniques to identify the impact of various other measures of source code reading/comprehension (cluster 2 in Figure 1), and they did not provide rich result details particularly for this case.

The other contradiction concerns the use of commas, which can be controversial. None of the papers provided any explanation on why they decided to measure commas nor why they assumed that commas might have some

influence on source code reading/comprehension. We can infer that the authors used commas as a surrogate for parameters in method calls. Another possible reason for measuring commas is to capture information concerning the declaration of multiple variables for the same type. In both cases, it might be sensible to say that the number of parameters in method calls/multiple variables being declared for the same type can increase the line length, which was shown in these same works (I [15], K [4] and T [35]) to be detrimental to source code reading/comprehension. Especially regarding the declaration of multiple variables in the same line, this might also limit the existence of comments for variable explanations, which can be detrimental to source code comprehension. The lack of further information on this prevents us from making more conjectures about the results.

Regarding identifier length, it is possible to observe that some works dealt with it in a broader scenario, measuring the average length of identifiers in a whole program unit, while others had their focus on the length of a single identifier, disregarding their presence in a program unit. Although most of them assessed the identifier length through the number of characters as presented in DeYoung and Kampen (A) [21], Jørgensen (B) [22], Buse et al. (I and K) [15] [4], and Butler et al. (M) [29]; Lawrie et al. in 2006 (G) [13] and 2007 (H) [14] used identifiers in the form of complete words, abbreviation and single character to assess the influence of these configurations on the source code comprehension. The last study concluded that the identifiers with complete words influence more positively the source code comprehension when compared to abbreviations and single character formats. Lawrie et al. (H) [14] could also draw some interesting conclusions regarding the gender/experience versus identifiers comprehension. Their results revealed that informative identifier names are more important for women than for men, even though women comprehend more from abbreviations than men do. Inexperienced programmers have more difficulty comprehending uninformative identifiers than experienced ones; on the other hand, neither work experience nor schooling seems to influence the writing of correct source code descriptions.

Table 5: Measurement procedures and contradictory behaviors for program unit semantic complexity and identifier length

Entity	Attribute	Measurement Procedure	Impact on Source Code Readability/Comprehensibility
Program Unit	Semantic Complexity	Divide the number of arithmetic operators by the number of lines	<div>⊖ B [22] → the larger the number of arithmetic operators per line, the worse the source code comprehension</div> <div>⊕ I [15] & K [4] → the larger the number of arithmetic operators per line, the better the source code reading/comprehension</div>
		$e^{-\left(\frac{NF}{LoC}\right)}$, where NF is the number of arithmetic formulas and LoC is the number of lines of code	<div>⊖ T [35] → the bigger the formula value, the worse the source code reading/comprehension</div>
		Divide the number of commas (parameters or variables in declaration) by the number of lines	<div>⊖ I [15] & K [4] → the larger the number of commas per line, the worse the source code reading/comprehension</div> <div>⊕ T [35] → the larger the number of commas per line, the better the source code reading/comprehension</div>
		Count the number of characters in the longest identifier (<u>maximum length of identifier</u>)	<div>⊖ I [15] & K [4] → the bigger the maximum length of identifier, the worse the source code reading/comprehension</div>
		Sum the length (in character) of each variable/identifier and divide it by the number of	<div>⊗ A [21] → no impact on source code reading/comprehension was observed concerning the average length of variables</div>

		<i>variables/identifiers (average length of variables/identifiers)</i>	<p>⊕ B [22] → the bigger the average length of variables, the better the source code comprehension</p> <p>⊙ I [15] & K [4] → no impact on source code reading/comprehension was observed concerning the average length of identifiers</p>
		<i>Identify the length variant among three options: <u>meaningful full-word</u>; <u>meaningful single-letter</u> and <u>meaningful abbreviation</u></i>	⊕ G [13] & H [14] → full-word identifiers contribute more to source code comprehension than the other two length variants, and abbreviations contribute more to source code comprehension than single-letter identifiers.
Identifier	Length	Count the number of words in the identifier	⊖ M [29] → the less the number of words (more than two and less than five), the better the source code reading/comprehension
		Count the number of characters in the identifier	⊖ M [29] → the less the number of characters (more than eight and less than 26), the better the source code reading/comprehension

Even though the results for identifier length were not conclusive, DeYoung and Kampen (A) [21] observed that short identifiers names are more suited to use in limited program scope, while long identifiers names in broader program scopes. Jørgensen (B) [22] obtained equivalent results to Lawrie et al. (G and H) [13] [14], showing that the longer the identifier, the better is the comprehension of source code. In the opposite direction are the results of Buse et al. [6, 14], which says the average length of identifiers names has no influence on readability/comprehensibility and the length of the most extended identifiers has negative impact on the readability/comprehensibility of source code, meaning the longer the identifier, the worse to read/comprehend the source code. Butler et al. (M) [29] also presented negative results. However, the authors argued that there is both an upper and a lower limit to the length of the identifier, meaning it cannot be too long nor too short.

2.4.3 Contradictory Behaviors for Program Unit Internal Documentation

The contradiction concerning the presence of comments can also be controversial since it is almost universal sense that commenting the source code is an excellent practice for improving its comprehension. Among the works listed in Table 6, the one by Lee, Lee and In (U) [36] was the only one suggesting the contrary idea. It can be justified by the fact that the authors were not considering comments as part of the source code, this way they noticed that whenever a source code was assessed as not being readable/comprehensible, there was a comment attached to it to provide explanations to clarify probable doubts from readers. It is important to note that the quality appraisal score for this work was significant low, meaning it would not be used to make a definitive decision in our evidence-based work setting.

Table 6: Measurement procedures and contradictory behaviors for program unit internal documentation

Entity	Attribute	Measurement Procedure	Impact on Source Code Readability/Comprehensibility
Program Unit	Internal Documentation	Identify whether the program unit has comments	⊕ C [23] → the presence of comments contributes to better source code comprehension
			⊕ E [25] → the presence of comments contributes to better source code comprehension

	⊕ V [5] → the presence of comments contributes to better source code comprehension
	⊖ U [36] → the presence of comments worsens the source code reading/comprehension
Identify whether the program unit has comments for methods	⊖ U [36] → the presence of comments for methods worsens the source code reading/comprehension
Identify whether the program unit has comments for variables	⊖ U [36] → the presence of comments for variables worsens the source code reading/comprehension

2.4.4 Contradictory Behaviors for Layout Style

The last group of contradictions (Table 7) regards the number of spaces used for indentation. Although not all the works used the same attribute for indentation, the idea of analysing the influence of indentation spacing on source code readability was presented in the empirical studies by Jørgensen (B) in 1980 (B) [22], Miara et al. in 1983 (D) [24], Buse et al. in 2008 [15] and in 2010 [4], and Lee, Lee and In (U) in 2015 [36]. According to these works, indentation does aid program readability, but the number of spaces used for indentation differs among the studies. The works described by Jørgensen (B) [22] and by Lee, Lee and In (U) [36] presented a positive relation between the indentation spacing and the readability of source code, meaning the more spaces used for indentation, the better. Conversely, all other works described the negative relationship between the number of spaces used for indentation and readability, meaning although significant, the indentation should not contain too many spaces to prevent the hampering of readability/comprehensibility.

Miara et al. (D) [24] showed that the best results for source code readability and comprehensibility (characteristic also evaluated in their study) were concerned with using indentation from two to four spaces, as 6-space indentation made either the length of lines become longer or divided the statements into different lines. According to their work, the increasing of the indentation spaces and use of nested commands reduce the readability of source code.

Table 7: Measurement procedures and contradictory behaviors for program unit layout style

Entity	Attribute	Measurement Procedure	Impact on Source Code Readability/Comprehensibility
Program Unit	Layout Style	Identify the indentation variant among the four options: <u>zero space</u> , <u>two spaces</u> , <u>four spaces</u> and <u>six spaces</u>	⊖ D [24] → two and four spaces for indentation contribute to source code reading/comprehension ⊕ U [36] → the larger the number of spaces for indentation (more than four), the better the source code reading/comprehension
		Divide the number of blank characters in the left margin by the number of characters in the program unit	⊕ B [22] → the larger the number of blank characters in the left margin per program unit characters, the better the source code comprehension
		Divide the number of blank characters in the left margin by the number of lines in the program unit	⊖ I [15] & K [4] → the larger the number of blank characters in the left margin per line, the worse the source code reading/comprehension
		Count the number of blank characters in the left margin of the deepest code block (<u>maximum indentation</u>)	⊖ I [15] & K [4] → the bigger the maximum indentation, the worse the source code reading/comprehension

3 Empirical Study on Contradictory Attributes for Readability and Comprehensibility of Source Code

The contradictory results challenged our decision-making on using or not such source code attributes to tailor coding guidelines for the company. Thus, we still had to observe these divergent results, and therefore decided to evaluate some of them in an external context. We should stress, though, that not all of the attributes would be discussed with the company's practitioners, only those with a high-quality assessment and that was important to the company were considered at that moment.

3.1 Observing Contradictions: the Empirical Study Planning

An empirical study was planned with the object presented in Table 8 such as follows.

Table 8: Empirical study objective

Analyze for the purpose of with respect to from the point of view of in the context of	source code attributes concerning code size, length of identifiers, comments presence and indentation spacing characterization their influence on readability and comprehensibility of source code computer science undergraduate students with different backgrounds a software engineering course at the Federal University of Rio de Janeiro
---	---

During the empirical study, we wanted to respond the following **research questions (RQ)** that were also used to guide the rest of the planning:

- What is the influence of the size of code on the comprehensibility of source code? (RQ1)
 - What features of the source code, participants or environmental characteristics support this influence?
- What is the influence of the length of identifiers on the readability and comprehensibility of source code? (RQ2)
 - Do gender, programming experience, and program scope features support this influence?
- What is the influence of the presence of comments on the comprehensibility of source code? (RQ3)
 - Do programming experience, and program scope features support this influence?
- What is the influence of code indentation spacing on the readability of source code? (RQ4)
 - Do the length of lines, nested commands, and programming experience features support this influence?

For each RQ, an H_0 (null hypotheses) representing no influence of the attribute on the quality characteristic was also formulated.

To answer these research questions, we decided to observe the participants' perceptions regarding the readability and comprehensibility of code snippets. We selected and modified, according to the mentioned attributes, source code snippets from real open source projects. Python was the programming language used in the snippets due to the participants' familiarity according to a previous survey done during the software engineering course. Two different well-known open source projects (according to GitHub⁶) written in Python language provided the snippets: BitTorrent and YoutubeDL. For each attribute, two separate versions of the same source code have been created: i): small versus large (in terms of number of lines) source code size; ii) complete-word versus abbreviation relating to the length of identifiers; iii) with versus without comments; and iv) four versus eight spaces regarding indentation spacing. The snippets should be short and self-contained to be assessed during the time (one hour) available for the task.

The empirical study was designed to be conducted in person (in vitro) during a software engineering class regarding the quality of source code. The participants received a lecture on software quality and a general explanation on readability and comprehensibility of source code – no details about source code attributes were provided before the study. Next, they received: i) a consent form; ii) a characterization form; and iii) eight different source code snippets (randomly assigned to the students, one for each version of the attributes under evaluation). Since we did not want to have different source code attributes interacting among themselves, we decided for using different source

⁶ <https://github.com/>

code snippets for the source code attributes evaluation, controlling to the possible extension the factors affecting the source code readability and comprehensibility. This way we had eight different source code snippets, each one having two versions according to each attribute under evaluation. We wanted to guarantee that all the participants would receive the same code snippets, with the only difference being the versions of them (see Figure 3). The study was anonymous, and it did not influence the performance of participants in the module.

Following some pieces of advice on how to measure program comprehension [37], we assembled the study so that the participants should provide their perceptions on the readability (whether applicable) and comprehensibility (whether applicable) of snippets by using a Likert scale (five points) along with reasons (can be more than one) for the perception (in the case of readability) and an answer regarding a comprehension question (in the case of comprehensibility). Since they received printed snippets, some strategies were used to simulate better the way source code is typically viewed on a computer screen, for instance, printing the snippets in landscape orientation and bolding the native commands. In addition, smileys were used as a replacement for written Likert levels codes. Figure 3 depicts an example of two versions of the same code snippet used in the study (a participant assessed only one of the versions).

Snippet 4	Projeto: YoutubeDL
<pre> 1 def calc_eta(start, now, total, current): 2 if total is None: 3 return None 4 if now is None: 5 now = time.time() 6 dif = now - start 7 if current == 0 or dif < 0.001: # One millisecond 8 return None 9 rate = float(current) / dif 10 return int((float(total) - float(current)) / rate) </pre>	
<pre> 1 def calculate_estimated_time_of_arrival(start_time, now, total, current_position): 2 if total is None: 3 return None 4 if now is None: 5 now = time.time() 6 elapsed_time = now - start_time 7 if current_position == 0 or elapsed_time < 0.001: # One millisecond 8 return None 9 rate = float(current_position) / elapsed_time 10 return int((float(total) - float(current_position)) / rate) </pre>	
<p>READABILITY:</p> <p> </p>	<p>Explain your choice for readability:</p> <p>() parentheses, because _____</p> <p>() command separator, because _____</p> <p>() identifiers, because _____</p> <p>() other: _____</p>
<p>COMPREHENSIBILITY:</p> <p> </p>	<p>Explain the "rate" calculation (line 9):</p>

Figure 3: Two Different Versions of a Code Snippet Concerning Length of Identifiers

A senior programmer performed a pilot to assess the code snippets difficulty level and the time to complete all the evaluations. The senior programmer completed the study in 29 minutes (all possible options) and provided some feedback regarding the possible difficulty levels for undergraduate students, which supported the replacement of some of the previously selected snippets.

3.2 Contradictory Results

The study had 38 participants. A third of them finished the evaluation in less than 40 minutes. Their characterization revealed a broad range of different profiles. According to their self-reported expertise (using a five-point *Likert* scale ranging from "I have no experience" to "I practice it in several projects in the industry") on writing and debugging source code, software development, and evolution activities, and Python language knowledge, half of them was characterized in novices and half, in experts; according to their self-reported knowledge (using a three-point *Likert* scale ranging from "I do not know the domain" to "I am familiar with the domain") of the software from where the

snippets were extracted (BitTorrent and YoutubeDL), half of them was marked as low level of knowledge and half, as high level of knowledge. We decided for using self-reported expertise and knowledge since a recent study showed that it could be an accurate way of measuring expertise [38].

Whereas looking the overall results for each attribute may not give us any conclusive answer as to the reasons for their different influence on the readability and comprehensibility of source code, the data stratification considering the main findings in the literature review can give us some hints about the previously observed contradictory results. Thus, the participants' characteristics and source code properties were used to support the data analysis, as can be seen in the following subsections.

3.2.1 Results Regarding Source Code Size (RQ1)

Neither readability nor comprehensibility perceptions presented significant differences in the analyzed cases (see Table 9). Concerning the readability characteristic, almost all explanations against the size of code were for the bigger code snippet versions, independent of the participants' programming experience, which makes us conjecture whether the code size has a negative influence on its readability. It is important to note that more lines tended to have worse readability perceptions for experts than for novices.

Table 9: Readability and comprehensibility perception concerned with code size

Experience	Snippet ID	Style	Mann-Whitney Test – <i>p-value</i>	Code Size – Visual Information of Median and Range for Perception					#Participants	#Mentioning Code Size OR	#for OR #Wrong	#Against OR #IDK/ Blank
Novices	1	Less	1						11	2	2	0
		More							8	4	3	1
	2	Less	0.097						8	3	3	0
		More							11	6	6	0
Experts	1	Less	0.591						9	3	2	1
		More							10	3	0	3
	2	Less	0.54						10	7	7	0
		More							9	5	4	1
Readability Perceptions →				Readability Explanations								
Novices	1	Less	0.6						11	10	0	1
		More							8	7	1	0
	2	Less	0.175						8	6	1	1
		More							11	8	1	2
Experts	1	Less	0.765						9	8	1	0
		More							10	8	2	0
	2	Less	0.484						10	7	2	1
		More							9	8	1	0
Comprehensibility Perceptions →				Comprehensibility Answers								

On the other hand, the results for comprehensibility reached a bit more agreement among the participants and snippets versions. The results, in this case, showed some trend on the more substantial snippet versions, especially for novices, which presented more positive comprehension perceptions than the experts. While analyzing other source code attributes of the two snippets under evaluation, we observed some hints for this slight difference. The smaller version of snippet 1 has more complicated arithmetic expressions than its larger version, which took advantage of

more IF-conditions to solve the problem of formatting bytes with their suffix. According to what was pointed in the previous sections arithmetic expressions can hamper the comprehension of source code, but what we noticed is that their complexity is more detrimental than their quantity. In the case of snippet 2, the smaller version avoided the use of an additional IF-condition while the more extensive version made explicit the return of a function through this condition. At least concerning the perception, novices tended to like more the versions with explicit source code in comparison to experts. No differences in results could be seen to different participants' domain knowledge.

3.2.2 Results Regarding Identifier Length (RQ2)

Gender, programming experiences, and program scope were mentioned as factors that could contribute to the influence of the identifiers length on the readability and comprehensibility of source code. Five out of 38 participants were women, which makes unfeasible to draw any conclusion towards the gender influence on the length of identifier and readability/comprehensibility analysis. The fact that we were more interested in assessing 'low-level' details of readability and comprehensibility (preference of the company) made us select self-contained snippets, which means none of the snippets in the study presented identifiers from outside their scopes, also making unfeasible the program scope analysis. Still, programming experience and domain knowledge were used to support data stratification.

Table 10: Comprehensibility perception concerned with identifier length

Exper./Know				Identifier Length – Visual Information of Median and Range for Perception					
Snippet ID		Style	Mann-Whitney Test – <i>p-value</i>			#Participants	#Right	#Wrong	#IDK/ Blank
Novices	3	Abrev.	0.719			8	8	0	0
		Compl.				11	10	1	0
	4	Abrev.	0.087			11	5	4	2
		Compl.				8	6	0	2
Experts	3	Abrev.	0.003			10	9	1	0
		Compl.				9	9	0	0
	4	Abrev.	0.799			9	6	2	1
		Compl.				10	6	3	1
Comprehensibility Perceptions →				Comprehensibility Answers					
Low	3	Abrev.	0.014			9	8	1	0
		Compl.				10	10	0	0
	4	Abrev.	0.797			10	6	2	2
		Compl.				9	6	2	1
High	3	Abrev.	0.367			9	9	0	0
		Compl.				10	9	1	0
	4	Abrev.	0.138			10	5	4	1
		Compl.				9	6	1	2
Comprehensibility Perceptions →				Comprehensibility Answers					

Although the participants wrote their perceptions regarding the readability of identifiers, all the given explanations for and against the style concerned with comprehension issues instead of reading ones, which made us conjecture whether the participants could correctly assess the influence of this attribute on this characteristic. The fact that both readability and comprehensibility perceptions presented similar results no matter what the used data stratification (experience or domain knowledge), also lower our confidence in making conclusions about the length of identifiers and readability. The results regarding the length of identifiers and comprehensibility, on the other hand, highlighted interesting insights. The comprehensibility perceptions for long identifiers were higher in most of the

cases. Participants with higher programming experiences and low level of domain knowledge were the ones to give lower comprehensibility perception for long identifiers, and it was regarding the snippet 4 (see Figure 3 Table 10). Concerning the statistical tests, significant differences were found in experts and low-level domain concerning snippet 3.

Even though the perceptions were different, we could not conclude much from the scores for the comprehension questions. We could observe, though, that inexperienced programmers did have more difficulty in comprehending short identifiers than experts in the case of snippet 4, which it is by Lawrie et al. findings [14]. In fact, their perception was almost opposite for this case. It is important to point out that the snippet 4 is more extensive, has long lines of code, and presents more information than the snippet 3, supporting an explanation for difficulties presented by novices when compared with experts. We believe identifiers names embody the connection between the source code and the problem domain [32], and for this reason, we expected that domain knowledge could show some influence on the relation between identifier length and source code comprehension. Surprisingly, high domain knowledge did not seem to lead to better scores on comprehension questions. As an interesting observation, many participants in the expert group were also in the low-level domain knowledge group. This fact might help to explain the similarities obtained in these two groups for snippet 3, meaning; we cannot be sure whether the experience or the knowledge influenced the results the most. Probably these other factors confounded this result.

3.2.3 Results Regarding Comments Presence - Study Design Control (RQ3)

The presence of comments was used as a design control to enhance our confidence with the participants' responses for the other attributes. We decided to use the presence of comments as a control because it is a consensus in many studies that the presence of comments influences the comprehensibility of source code positively. To avoid the negative influence presented by Lee, Lee and In (U) [36], we warned the participants that everything that was inside the code box should count in their assessment (see Table 11).

Table 11: Comprehensibility perception concerned with presence of comments

Exper./Know		Snippet ID	Style	Mann-Whitney Test – <i>p</i> -value	Comments – Visual Information of Median and Range for Perception					#Participants	#Right	#Wrong	#IDK/ Blank					
Novices	5	With	Without	0.002						11	10	0	1					
		8								0	2	6						
	6	With	Without	0.015						8	8	0	0					
		11								6	1	4						
Experts	5	With	Without	0.019						9	7	1	1					
		10								2	4	4						
	6	With	Without	0.035						10	10	0	0					
		9								7	0	2						
Comprehensibility Perceptions →										Comprehensibility Answers								
Low	5	With	Without	0.008											12	10	1	1
		7													1	3	3	
	6	With	Without	0.008	7	7	0	0										
		12			9	1	2											
High	5	With	Without	0.006	8	7	0	1										
		11			1	3	7											
	6	With	Without	0.033	11	11	0	0										
		8			4	0	4											
Comprehensibility Perceptions →					Comprehensibility Answers													

The results showed that while answering comprehension questions, students achieved better scores for snippets presenting comments describing their function, as expected. Their perceptions concerning the comprehensibility of snippets were more favourable for those presenting comments as well, which was also confirmed by Mann-Whitney statistic tests for ordinal data ($p\text{-value} < 0.05$ for novices, experts, low level and high-level knowledge – $\alpha = 5\%$, as shown in Table 11), allowing us indicating that there is a difference in the perception of source code with and without comments. This initial analysis allowed us to strengthen confidence in the answers for the other attributes.

3.2.4 Results Regarding Indentation Spacing (RQ4)

Even though Python enforces a specific indentation scheme, the number of spaces used for indentation was under evaluation, rather than the existence of indentation, since the principal contradiction encountered in the technical literature was regarding the former. We clustered the results according to the participants' experience as it can be seen in Table 12.

Table 12: Readability perception concerned with indentation spacing

Experience	Snippet ID	Style	Mann-Whitney Test – <i>p-value</i>	Indentation Spacing – Visual Information of Median and Range for Perception					#Participants	#Mentioning Indentation	#for the Style	#Against the Style
Novices	7	4 spaces	0.669		10	5	5	0				
		8 spaces			9	8	4	4				
	8	4 spaces	0.733		9	5	5	0				
		8 spaces			10	4	2	2				
Experts	7	4 spaces	0.965		10	4	4	0				
		8 spaces			9	7	5	2				
	8	4 spaces	0.3435		9	2	2	0				
		8 spaces			10	8	3	5				
Readability Perceptions →				Readability Explanations								

No significant difference in perception was observed regarding indentation spacing – neither for novices nor experts. Some interesting observations can be made with the perception dispersion in the two groups though. The readability perceptions did not alter much in the expert group as happened in the novice one. As an interesting observation, all the explanations against the used indentation style were towards snippets with eight spaces, indicating an agreement between the results obtained in our empirical study and the ones presented by Miara et al. [24] and Buse et al. [15] [4].

In both groups, the median for readability perception was lower for snippet 8 in its 8-space indentation version in comparison to the other snippets configurations. As additional information, snippet 8 had shorter lines of source code as compared to snippet 7; another difference is the existence of a loop in the snippet 7 that is not present in snippet 8. Other than those, no further explanations could be gathered to support the justification for the difference in perception. Further investigation is required to check whether line length and loop can be moderator factors of indentation spacing and readability of source code. No findings related to nested commands were observed, probably due to the simplified source code structures used in our study.

4 Threats to Validity

Concerning the literature review, we are aware that the coverage of articles might not be complete, mainly because we had to simplify our search to provide information feedback to the company in a reasonable time span. We must

acknowledge that the list of included works presented in this paper can be used as *quasi-golden standard* [39] in future investigations in the area.

Concerning the study, as the human judgment of quality is somewhat subjective, the alternative used to measure it might not have been the most appropriate one, threatening the **construct validity**. Also, since it is almost impossible to isolate the source code attributes to evaluate their influence on quality separately from the rest of the source code, some other source code attributes might have affected the participants' responses, threatening the **internal validity**. Besides, there is a chance the participants' perceived readability and comprehensibility as a single concept, even though their differences were explained beforehand. Concerning the **external validity**, the results from this empirical study cannot be generalized neither can be applied in general cases, since the selected code snippets and participants may not be representative for a broader industrial scenario. Statistical tests for ordinal data were used to support the analysis of readability and comprehensibility perceptions to mitigate the threats regarding **conclusion validity**.

5 Conclusions

The results presented in this work supported a more thorough understanding of the interaction among programmers' characteristics, program elements, and source code readability and comprehensibility, by describing previous study observations and conducting an empirical study regarding source code attributes and their impact on source code quality. We presented 13 different attributes and 94 measurement procedures that have evidence of their influence on source code reading and comprehension. Some contradictions were identified during the literature review, which guided us to an empirical study investigation. For the code size, while it has a negative influence on readability – in particular for experts –, our results showed a positive impact on comprehensibility, supporting the explanation of the contradictory results found in the technical literature. The comprehensibility perceptions for long identifiers was higher in most of the cases – showing some significant differences for snippet 3 –, especially for less experienced participants, showing a positive relation between identifier length and comprehensibility. Regarding the presence of comments (our study design control), we could observe with significant results that the presence of comments is favorable to source code comprehension. Moreover, even though no significant difference regarding readability perceptions was observed, participants showed preferences for indentation using four over eight spaces, indicating a negative relation between the indentation spacing and readability. Most likely, a misunderstanding between the concepts of readability and comprehensibility caused the contradictory results presented in the technical publications. Still, the presented results indicate that coding guidelines contradictions still demand further investigation to provide indications on possible confounding factors explaining some of the inconclusive results and to support the clarification of these findings.

Acknowledgements

The authors thank CAPES and CNPq for supporting this research and the students at EEL873/UFRJ for taking part in the empirical study. Prof. Travassos is a CNPq researcher.

References

- [1] B. Kitchenham and S. L. Pfleeger, "Software quality: the elusive target," *IEEE Software*, vol. 1, no. 13, pp. 12-21, January 1996.
- [2] B. Boehm and V. R. Basili, "Software defect reduction top 10 list," *Computer*, vol. 34, no. 1, pp. 135-137, January 2001.
- [3] T. V. Ribeiro and G. H. Travassos, "On the alignment of source code quality perspectives through experimentation: an industrial case," in *Proceedings of the 3rd International Workshop on Conducting Empirical Studies in Industry (CESI/ICSE 2015)*, Florence, Italy, 2015.
- [4] R. P. L. Buse and W. R. Weimer, "Learning a metric for code readability," *IEEE Transactions on Software Engineering*, vol. 36, no. 4, pp. 546-558, 2010.
- [5] J. Börstler and B. Paech, "The role of method chains and comments in software readability and comprehension - An experiment," *IEEE Transactions on Software Engineering*, vol. 9, pp. 886-898, 2016.

- [6] P. Oman and C. Cook, "A paradigm for programming style research," *ACM SIGPLAN Notices*, vol. 23, no. 12, pp. 69-78, 1988.
- [7] T. V. Ribeiro and G. H. Travassos, "Who is right? Evaluating empirical contradictions in the readability and comprehensibility of source code," in *Proceedings of the XX Ibero-American Conference on Software Engineering*, Buenos Aires, 2017.
- [8] J. Kreimer, "Adaptive detection of design flaws," *Electronic Notes in Theoretical Computer Science*, vol. 141, no. 4, p. 117-136, 2005.
- [9] M. Alshayeb, "Empirical investigation of refactoring effect on software quality," *Information and Software Technology*, vol. 51, no. 9, pp. 1319-1326, 2009.
- [10] J. Rilling and S. P. Mudur, "3D visualization techniques to support slicing-based program comprehension," *Computers and Graphics*, vol. 29, no. 3, pp. 311-329, 2005.
- [11] R. Novais, C. Nunes, C. Lima, E. Cirilo, F. Dantas, A. Garcia and M. Mendonca, "On the proactive and interactive visualization for feature evolution comprehension: an industrial investigation," in *Proceedings of the 34th International Conference on Software Engineering*, Zurich, 2012.
- [12] G. K. Rambally, "The influence of color on program readability and comprehensibility," in *Proceedings of the Seventeenth SIGCSE Technical Symposium on Computer Science Education*, New York, 1986.
- [13] D. Lawrie, C. Morrell, H. Feild and D. Binkley, "What's in a name? A study of identifiers," in *Proceedings of the 14th IEEE International Conference on Program Comprehension*, Athens, 2006.
- [14] D. Lawrie, C. Morrell, H. Feild and D. Binkley, "Effective identifier names for comprehension and memory," *Innovations in Systems and Software Engineering*, vol. 3, no. 4, pp. 303-318, 2007.
- [15] R. P. L. Buse and W. R. Weimer, "A metric for software readability," in *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, New York, 2008.
- [16] D. Posnett, A. Hindle, and P. Devanbu, "A simpler model of software readability," in *Proceedings of the 8th Working Conference on Mining Software Repositories*, New York, 2011.
- [17] J. Biolchini, P. G. Mian, A. C. C. Natali and G. H. Travassos, "Systematic review in software engineering," Rio de Janeiro, 2005.
- [18] T. Ribeiro, Alinhando perspectivas de qualidade em código fonte a partir de estudos experimentais – um caso na indústria, Master Dissertation: COPPE / Universidade Federal do Rio de Janeiro, 2014, p. 161.
- [19] A. A. Takang, P. A. Grubb, and R. D. Macredie, "The effects of comments and identifier names on program comprehensibility: an experimental investigation," *Journal of Programming Languages*, vol. 4, pp. 143-167, 1996.
- [20] D. Malbaski, A. Kupusinac, and S. Popov, "The impact of coding style on the readability of C programs," *Technics Technologies Education Management*, vol. 6, no. 4, pp. 1073-1082, 2011.
- [21] G. E. DeYoung and G. R. Kampen, "Program factors as predictors of program readability," in *Proceedings of the Third IEEE Computer Society's International Computer Software and Applications Conference*, 1979.
- [22] A. H. Jørgensen, "A methodology for measuring the readability and modifiability of computer programs," *BIT Numerical Mathematics*, vol. 20, pp. 393-405, 1980.
- [23] S. Woodfield, H. Dunsmore and V. Shen, "Effect of modularization and comments on program comprehension," in *Proceedings of the 5th International Conference on Software Engineering*, Piscataway, 1981.
- [24] R. J. Miara, J. A. Musselman, J. A. Navarro and B. Shneiderman, "Program indentation and comprehensibility," *Communications of the ACM*, vol. 26, no. 11, pp. 861-867, 1983.
- [25] T. Tenny, "Program readability: procedures versus comments," *IEEE Transactions on Software Engineering*, vol. 14, no. 9, pp. 1271-1279, 1988.
- [26] B. Teasley, "The effects of naming style and expertise on program comprehension," *International Journal of Human-Computer Studies*, vol. 40, no. 5, pp. 757-770, 1994.
- [27] D. Binkley, M. B. Davis, D. A. Lawrie and C. A. Morrell, "To camelcase or under_score," in *Proceedings of the 17th IEEE International Conference on Program Comprehension*, Vancouver, 2009.

- [28] B. Sharif and J. Maletic, "An eye-tracking study on camelCase and under_score identifier styles," in *Proceedings of the IEEE 18th International Conference on Program Comprehension*, Braga, 2010.
- [29] S. Butler, M. Wermelinger, Y. Yu and H. Sharp, "Exploring the influence of identifier names on code quality: An empirical study," in *Proceedings of 14th European Conference on Software Maintenance and Reengineering*, Madrid, 2011.
- [30] J. Feigenspan, S. Apel, J. Liebig and C. Kästner, "Exploring software measures to assess program comprehension," in *Proceedings of the International Symposium on Empirical Software Engineering and Measurement*, Banff, 2011.
- [31] Z. Sharafi, Z. Soh, Y.-g. Guéhéneuc and G. Antoniol, "Women and men - different but equal: on the impact of identifier style on source code reading," in *Proceedings of the IEEE 20th International Conference on Program Comprehension*, Passau, 2012.
- [32] D. A. Binkley, M. C. Davis, D. A. Lawrie, J. I. D. Maletic, C. B. Morrell and B. E. Sharif, "The impact of identifier style on effort and comprehension," *Empirical Software Engineering*, vol. 18, no. 2, pp. 219-276, 2013.
- [33] T. Lee, J. Lee and H. In, "A study of different coding styles affecting code readability," *International Journal of Software Engineering and Its Applications*, vol. 7, no. 5, pp. 413-422, 2013.
- [34] Y. Sasaki, Y. Higo and S. Kusumoto, "Reordering program statements for improving readability," in *Proceedings of the 17th European Conference on Software Maintenance and Reengineering*, Genova, 2013.
- [35] Y. Tashtoush, Z. Odat, I. Alsmadi, and M. Yatim, "Impact of Programming Features on Code Readability," *International Journal of Software Engineering and Its Applications*, vol. 7, no. 6, pp. 441-458, 2013.
- [36] T. Lee, J. Lee and H. In, "Effect analysis of coding convention violations on readability of post-delivered code," *IEICE Transactions on Information and Systems*, Vols. E98-D, no. 7, pp. 1286-1296, 2015.
- [37] A. Dunsmore and M. Roper, "A comparative evaluation of program comprehension measures," *The Journal of Systems and Software*, vol. 52, no. 3, pp. 121-129, June 2000.
- [38] J. Siegmund, C. Kästner, S. A. Jörg Liebig and S. Hanenberg, "Measuring and modeling programming experience," *Empirical Software Engineering*, vol. 19, no. 5, pp. 1299-1334, 2014.
- [39] H. Zhang, M. A. Babar and P. Tell, "Identifying relevant studies in software engineering," *Journal Information and Software Technology*, vol. 53, no. 6, pp. 625-637, 2011.

Appendix A – List of Source Code Attributes, Measurement Procedures and Impacts on Source Code Readability/Comprehensibility

Table 13: Source code entities, attributes, measurement procedures and impacts on source code readability/comprehensibility

Entity	Attribute	Measurement Procedure	Unit	Scale	Impact on Source Code Readability/Comprehensibility
Program Unit	Size or Length	Count the number of lines	Lines of code	Ratio	<p>⊛ A [21] → no impact on source code reading/comprehension was observed concerning the number of lines</p> <p>⊛ N [30] → no impact on source code reading/comprehension was observed concerning the number of lines</p> <p>⊕ O [16] → the larger the number of lines, the better the source code reading/comprehension</p>
		Count the number of lines containing statements	Lines of code	Ratio	<p>⊖ A [21] → the larger the number of lines containing statements, the worse the source code reading/comprehension</p>
		Count the number of words	Words	Ratio	<p>⊖ O [16] → the larger the number of words, the worse the source code reading/comprehension</p>
		Count the number of characters	Characters	Ratio	<p>⊖ O [16] → the larger the number of characters, the worse the source code reading/comprehension</p>
		$NU1 \log_2 NU1 + NU2 \log_2 NU2$, where $NU1$ is the number of unique operators, and $NU2$ is the number of unique operands (<u>estimated program length</u>)	-	Ratio	<p>⊛ A [21] → no impact on source code reading/comprehension was observed concerning the estimated program length</p>
		Sum the number of operators and the number of operands (<u>program length</u>)	-	Ratio	<p>⊖ A [21] → the bigger the program length, the worse the source code reading/comprehension</p> <p>⊖ O [16] → the bigger the program length, the worse the source code reading/comprehension</p>
		$N \log_2 n$ where N is the sum of the number of operators and number of operands, and n is the sum of the number of	Bits	Ratio	<p>⊛ A [21] → no impact on source code reading/comprehension was observed concerning the program volume</p> <p>⊖ O [16] → the bigger the program volume, the worse the source code reading/comprehension</p>

	unique operators and number of unique operands (program volume)			
Structural Complexity	Count the number of code blocks	Code blocks	Ratio	⊛ A [21] → no impact on source code reading/comprehension was observed concerning the number of code blocks
	Sum the depths of the code blocks	-	Ratio	⊛ A [21] → no impact on source code reading/comprehension was observed concerning the number of depths of code blocks
	Divide the sum of depths of the code blocks by the number of code blocks (average depth of nesting)	-	Ratio	⊛ A [21] → no impact on source code reading/comprehension was observed concerning the average depth of nesting
	Identify the depth of the deepest code block (maximum depth of nesting)	-	Ratio	⊛ A [21] → no impact on source code reading/comprehension was observed concerning the maximum depth of nesting
	Divide the number of assignments by the number of lines	Assignments/ Line of code	Ratio	⊖ I [15] & K [4] → the bigger the amount number of assignments per line, the worse the source code reading/comprehension
	Count the number of branches and add one (cyclomatic complexity)	Branches	Ratio	⊖ A [21] → the larger the number of branches, the worse the source code reading/comprehension ⊛ N [30] → no impact on source code reading/comprehension was observed concerning the cyclomatic complexity
	Divide the number of branches by the number of lines	Branches/ Line of code	Ratio	⊖ I [15] & K [4] → the larger the number of branches per line, the worse the source code reading/comprehension ⊛ T [35] → no impact on source code reading/comprehension was observed concerning the number of branches per line
	$e^{-MaxNiD}$, where <i>MaxNiD</i> is the maximum nested if depth (nested if value)	-	Ratio	⊛ T [35] → no impact on source code reading/comprehension was observed concerning the nested if value
	Divide the number of loops by the number of lines	Loops/ Line of code	Ratio	⊖ I [15] & K [4] → the larger the number of loops per line, the worse the source code reading/comprehension

					☼ T [35] → no impact on source code reading/comprehension was observed concerning the number of loops per line
		$e^{-MaxNLD}$, where <i>MaxNLD</i> is the maximum nested loop depth (<u>nested loop value</u>)	-	Ratio	⊖ T [35] → the bigger the nested loop value, the worse the source code reading/comprehension
		Divide the number of goto by the number of labels in the program unit	Gotos/Label	Ratio	⊖ B [22] → the larger the amount of go to per label, the worse the source code comprehension
		e^{-R} , where <i>R</i> is the number of recursions (<u>recursive function value</u>)	-	Ratio	⊖ T [35] → the bigger the recursive function value, the worse the source code reading/comprehension
		Divide the number of inherited classes by the number of classes (<u>inheritance value</u>)	-	Ratio	☼ T [35] → no impact on source code reading/comprehension was observed concerning the inheritance value
		Divide the number of functions inherited from pure virtual functions declarations by the number of all function declarations (<u>overriding value</u>)	-	Ratio	☼ T [35] → no impact on source code reading/comprehension was observed concerning the overriding value
		Identify whether the program unit present method chains	-	Nominal	☼ V [5] → no impact on source code reading/comprehension was observed concerning the presence of method chains
		Divide the number of periods (method chains) by the number of lines	Periods/ Line of code	Ratio	⊖ I [15] & K [4] → the larger the number of periods per line, the worse the source code reading/comprehension
	Semantic Complexity	Count the number of keywords in the line of code with the biggest amount of keywords (<u>maximum number of keywords in a line</u>)	Keywords	Ratio	⊖ I [15] & K [4] → the bigger the maximum number of keywords in a line, the worse the source code reading/comprehension
		Divide the number of keyword by the number of lines	Keywords/Li ne of code	Ratio	⊖ I [15] & K [4] → the larger the number of keywords per line, the worse the source code reading/comprehension
		Identify whether the program unit has illegal token	-	Nominal	⊖ U [36] → the presence of illegal token worsens the source code reading/comprehension

	Count the number of operations	Operations	Ratio	⊛ N [30] → no impact on source code reading/comprehension was observed concerning the number of operations
	Count the number of operators	Operators	Ratio	⊛ A [21] → no impact on source code reading/comprehension was observed concerning the number of operators
	Divide the number of arithmetic operators by the number of lines	Operators/ Line of code	Ratio	⊖ B [22] → the larger the number of arithmetic operators per line, the worse the source code comprehension ⊕ I [15] & K [4] → the larger the number of arithmetic operators per line, the better the source code reading/comprehension
	$e^{-\left(\frac{NF}{LoC}\right)}$, where <i>NF</i> is the number of arithmetic formulas and <i>LoC</i> is the number of lines of code	Arithmetic formulas/ Line of code	Ratio	⊖ T [35] → the bigger formula value, the worse the source code reading/comprehension
	Divide the number of comparison operators by the number of lines	Operators/ Line of code	Ratio	⊖ I [15] & K [4] → the larger the average amount of comparison operators per line, the worse the source code reading/comprehension
	Count the number of unique operators	Operators	Ratio	⊛ A [21] → no impact on source code reading/comprehension was observed concerning the number of unique operators
	Divide the number of commas (parameters or variables in declarations) by the number of lines	Commas/ Line of code	Ratio	⊖ I [15] & K [4] → the larger the number of commas per line, the worse the source code reading/comprehension ⊕ T [35] → the larger the number of commas per line, the better the source code reading/comprehension
	Count the number of operands	Operands	Ratio	⊖ A [21] → the bigger the amount operands, the worse the source code reading/comprehension
	Count the number of unique operands	Operands	Ratio	⊛ A [21] → no impact on source code reading/comprehension was observed concerning the number of unique operands
	Sum the number of unique operators and the number of unique operands (program vocabulary)	-	Ratio	⊛ A [21] → no impact on source code reading/comprehension was observed concerning the programming vocabulary

					⊖ O [16] → the bigger the programming vocabulary, the worse the source code reading/comprehension
		Count the number of numbers in the line of code with the biggest amount of numbers (<u>maximum amount of numbers in a line</u>)	Numbers	Ratio	⊖ I [15] & K [4] → the larger the maximum amount of numbers in a line, the worse the source code reading/comprehension
		Divide the number of numbers by the number of lines	Number/ Line of code	Ratio	⊖ I [15] & K [4] → the larger the number of numbers per line, the worse the source code reading/comprehension
		Identify whether numbers are replaced by constants	-	Nominal	⊖ R [33] → the presence of numbers instead of symbolic constants worsens the source code reading/comprehension
		Divide the number of arrays by the number of identifiers (arrays value)	Arrays/ Identifier	Ratio	⊕ T [35] → no impact on source code reading/comprehension was observed concerning the arrays' values
		Count the number of unique variables/attributes	Variables or Attributes	Ratio	⊕ A [21] → no impact on source code reading/comprehension was observed concerning the number of unique variables ⊕ N [30] → no impact on source code reading/comprehension was observed concerning the number of unique attributes
		Count the number of identifiers in the line of code with the biggest amount of identifiers (<u>maximum number of identifiers in a line</u>)	Identifiers	Ratio	⊖ I [15] & K [4] → the bigger the maximum number of identifiers in a line, the worse the source code reading/comprehension
		Divide the number of identifiers by the number of lines	Identifiers/ Line of code	Ratio	⊖ I [15] & K [4] → the larger the number of identifiers per line, the worse the source code reading/comprehension
		Count the number of characters in the longest identifier (<u>maximum length of identifier</u>)	Characters	Ratio	⊖ I [15] & K [4] → the bigger the maximum length of identifier, the worse the source code reading/comprehension
		Sum the length (in character) of each variable/identifier and divide it by the number of variables/identifiers	Characters/ Variable or Identifier	Ratio	⊕ A [21] → no impact on source code reading/comprehension was observed concerning the average length of variables

	<u>(average length of variables/identifiers)</u>			<p>⊕ B [22] → the bigger the average length of variables, the better the source code comprehension</p> <p>✱ I [15] & K [4] → no impact on source code reading/comprehension was observed concerning the average length of identifiers</p>
	Sum the numbers of lines between the first and the last use of each variable and divide it by the number of variables (<u>width of variables</u>)	Lines of code/ Variable	Ratio	✱ A [21] → no impact on source code reading/comprehension was observed concerning the width of variables
	Divide the length (in characters) of each variable by its width (in lines between first and last use) and take the average of all such numbers (<u>normalized length of variables</u>)	Characters/ Line of code	Ratio	⊕ A [21] → the bigger the normalized length of variables, the better the source code reading/comprehension
	Count the number of occurrences of the single character that was the most present in the program unit	Occurrences	Ratio	⊖ I [15] & K [4] → the larger the number of occurrences of the most present single character, the worse the source code reading/comprehension
	Count the number of occurrences of the single identifier that was the most present in the program unit	Occurrences	Ratio	⊖ I [15] & K [4] → the larger the number of occurrences of the most present single identifier, the worse the source code reading/comprehension
	Identify whether the program unit has a local variable or a parameter that shadows another variable/parameter from a different scope	-	Nominal	⊕ U [36] → the presence of variables/parameters that shadow others contributes to better source code reading/comprehension
	Identify whether the program unit has unused modifiers	-	Nominal	⊖ R [33] → the presence of unused modifiers worsens the source code reading/comprehension
	Identify whether the program unit has redundant modifiers	-	Nominal	<p>⊖ R [33] → the presence of redundant modifiers worsens the source code reading/comprehension</p> <p>⊖ U [36] → the presence of redundant modifiers worsens the source code reading/comprehension</p>
	Divide the number of variables with meaningful names by the number of variables	-	Ratio	⊕ T [35] → the larger the number of variables with meaningful names per variable, the better the source code reading/comprehension

		$\sum_{i=1}^n p(x_i) \log_2 p(x_i), \text{ where } p(x_i) = \frac{\text{count}(x_i)}{\sum_{j=1}^n \text{count}(x_j)}$ <p>and x_i is a term (token or byte) in the program unit (entropy)</p>	-	Ratio	⊖ O [16] → the bigger the entropy, the worse the source code reading/comprehension
Implementation Level		$2 * \frac{NU2}{NU1} * N2,$ <p>where NU1 is the number of unique operators; NU2 is the number of unique operands, and N2 is the number of operands (estimated implementation level)</p>	-	Ratio	⊕ A [21] → the higher the estimated implementation level, the better the source code reading/comprehension
		$\left(2 * \frac{NU2}{NU1} * N2\right)^2 * ((N1 + N2) \log_2 (NU1 + NU2)),$ <p>where NU1 is the number of unique operators; NU2 is the number of unique operands; N1 is the number of operators and N2 is the number of operands (language level)</p>	-	Ratio	⊕ A [21] → the higher the language level, the better the source code reading/comprehension
	Internal Documentation	<p>Identify whether the program unit has comments</p> <p>Identify whether the program unit has comments for methods</p> <p>Identify whether the program unit has comments for variables</p>	- - -	Nominal Nominal Nominal	<p>⊕ C [23] → the presence of comments contributes to better source code comprehension</p> <p>⊕ E [25] → the presence of comments contributes to better source code comprehension</p> <p>⊕ V [5] → the presence of comments contributes to better source code comprehension</p> <p>⊖ U [36] → the presence of comments worsens the source code reading/comprehension</p> <p>⊖ U [36] → the presence of comments for methods worsens the source code reading/comprehension</p> <p>⊖ U [36] → the presence of comments for variables worsens the source code reading/comprehension</p>

		Identify whether the comments and statements are on the same lines	-	Nominal	<div> <div></div> <div>U [36] → the presence of comments in the same line of statements worsens the source code reading/comprehension</div> </div>
		Count the number of comments	Comments	Ratio	<div> <div></div> <div>A [21] → no impact on source code reading/comprehension was observed concerning the number of comments</div> </div>
		Count the number of characters in comments	Characters	Ratio	<div> <div></div> <div>A [21] → the larger the number of characters in comments, the better the source code reading/comprehension</div> </div>
		Divide the number of characters in comments by the number of characters in the program unit	-	Ratio	<div> <div></div> <div>B [22] → the larger the number of characters in comments per program unit characters, the better the source code comprehension</div> </div>
		Divide the number of lines of comments by the number of lines	Lines of comments/ Line of code	Ratio	<div> <div></div> <div>I [15] & K [4] → the larger the number of comments per line, the better the source code reading/comprehension</div> <div> <div></div> <div>T [35] → the larger the number of comments per line, the better the source code reading/comprehension</div> </div> </div>
		Count the number of blocks of contiguous comments	Blocks	Ratio	<div> <div></div> <div>A [21] → no impact on source code reading/comprehension was observed concerning the number of blocks of contiguous comments</div> </div>
	Layout Style	Divide the numbers of blank lines by the number of lines	Blank lines/ Line of code	Ratio	<div> <div></div> <div>B [22] → the larger the number of blank lines per line, the better the source code comprehension</div> <div> <div></div> <div>I [15] & K [4] → the larger the number of blank lines per line, the better the source code reading/comprehension</div> </div> </div>
		Divide the number of characters by the number of lines	Characters/ Line of code	Ratio	<div> <div></div> <div>I [15] & K [4] → the larger the number of characters per line, the worse the source code reading/comprehension</div> </div>
		Count the number of characters in the longest line	Characters	Ratio	<div> <div></div> <div>I [15] & K [4] → the larger the number of characters in the longest line, the worse the source code reading/comprehension</div> </div>
		Divide the median line length by the maximum line length (<u>line length</u>)	-	Ratio	<div> <div></div> <div>T [35] → no impact on source code reading/comprehension was observed concerning the line length</div> </div>

	Divide the numbers of spaces by the number of lines	Spaces/ Line of code	Ratio	⊖ I [15] & K [4] → the larger the number of spaces per line, the worse the source code reading/comprehension
	Identify the indentation variant among the four options: <u>zero space</u> , <u>two spaces</u> , <u>four spaces</u> and <u>six spaces</u>	-	Interval	⊖ D [24] → two and four spaces for indentation contribute to source code reading/comprehension ⊕ U [36] → the larger the number of spaces for indentation (more than four), the better the source code reading/comprehension
	Identify whether the program unit uses tab instead of spaces for indentation	-	Nominal	⊕ R [33] → the use of tabs for indentation contributes to better source code comprehension when compared to spaces
	Divide the number of places with indentation by the number of places where indentation is required	-	Ratio	⊕ T [35] → the larger the number of places with indentation, the better source code reading/comprehension
	Divide the number of blank characters in the left margin by the number of characters in the program unit	-	Ratio	⊕ B [22] → the larger the number of blank characters in the left margin per program unit characters, the better the source code comprehension
	Divide the number of blank characters in the left margin by the number of lines in the program unit	Characters/ Line of code	Ratio	⊖ I [15] & K [4] → the larger the number of blank characters in the left margin per line, the worse the source code reading/comprehension
	Count the number of blank characters in the left margin of the deepest code block (<u>maximum indentation</u>)	Characters	Ratio	⊖ I [15] & K [4] → the bigger the maximum indentation, the worse the source code reading/comprehension
	Divide the number of parenthesis by the number of lines	Parenthesis/ Line of code	Ratio	⊖ I [15] & K [4] → the larger the number of parenthesis per line, the worse the source code reading/comprehension
	Identify whether the program unit has unnecessary use of parenthesis	-	Nominal	⊖ [36] → the presence of unnecessary parenthesis worsens the source code reading/comprehension
	Identify whether the program unit has unnecessary use of curly braces	-	Nominal	⊕ R [33] → the presence of unnecessary braces contributes to better source code reading/comprehension
	Identify whether the program unit has left curly for beginning code blocks along with the heading declaration statements	-	Nominal	⊕ R [33] → the presence of left curly along with heading declaration statements contributes to better source code reading/comprehension

		Identify whether there is a space after a comma	-	Nominal	⊕ R [33] → the presence of spaces after comma contributes to better source code reading/comprehension
Identifier	Consistency	Identify whether the identifier letter case is consistent	-	Nominal	⊛ M [29] → no impact on source code reading/comprehension was observed concerning the mixed use of upper and lower-case characters
		Identify whether constants are capitalized	-	Nominal	⊕ R [33] → the presence of capitalized constants contributes to better source code reading/comprehension
		Identify whether the identifier has a consistent multi-word separator style	-	Nominal	⊕ M [29] → the use of capitalization correctly contributes to better source code reading/comprehension
					⊛ M [29] → no impact on source code reading/comprehension was observed concerning the use of external underscores
	Meaningfulness	Identify whether the identifier relates to the source code or not: <u>meaningful</u> and <u>non-meaningful</u>	-	Nominal	⊕ F [26] → the use of meaningful identifiers contributes to better source code comprehension
		Identify whether the identifier has its words in a shared dictionary and its abbreviations and acronyms are more commonly used than the unabbreviated forms or not: <u>well-known</u> identifiers and <u>unknown</u> identifiers	-	Nominal	⊕ M [29] → the use of words in the dictionary and well-known abbreviations contributes to better source code comprehension
		Identify whether the identifier is composed entirely of numeric words and/or numbers	-	Nominal	⊛ M [29] → no impact was observed on source code reading/comprehension of identifiers composed entirely of numeric words and/or numbers
	Length	Identify the length variant among three options: <u>meaningful full-word</u> ; <u>meaningful single-letter</u> and <u>meaningful abbreviation</u>	-	Ordinal	⊕ G [13] & H [14] → full-word identifiers contribute more to source code comprehension than the other two length variants, and abbreviations contribute more to source code comprehension than single-letter identifiers.
		Count the number of words in the identifier	Words	Ratio	⊖ M [29] → the less the number of words (more than two and less than five), the better the source code reading/comprehension

	Count the number of characters in the identifier	Characters	Ratio	⊖ M [29] → the less the number of characters (more than eight and less than 26), the better the source code reading/comprehension
Scope	Identify whether the variable definition and its first reference is: <u>in different blocks; in the same block but not contiguous; in the same block and contiguous</u>	-	Ordinal	⊖ S [34] → the bigger the distance between the variable definition and its first reference, the worse the source code reading
Multi-word Separator Style	Identify the identifier multi-word separator style among two variants: <u>camelCase</u> (the first letter of each word, but the first word, is capitalized) and <u>under_score</u> (words are separated using underscore)	-	Nominal	⊕ J [27] → the use of camelCase contributes to better source code comprehension when compared to under_score ⊗ L [28] → there is no difference in comprehension between the use of camelCase or under_score for composing multi-word identifiers ⊗ P [31] → there is no difference in comprehension between the use of camelCase or under_score for composing multi-word identifiers ⊕ Q [32] → the use of camelCase contributes to better source code comprehension when compared to under_score
Type Encoding Style	Identify whether type information is encoded in the identifier using <u>Hungarian notation</u> (or similar) or not	-	Nominal	⊗ M [29] → no impact on source code reading/comprehension was observed concerning identifiers encoding type information
Comment	Identify whether the comment provide information beyond the actual code it explains	-	Nominal	⊕ V [5] → the better the quality of the comments, the better the source code comprehension