An Empirical Study Assessing Source Code Readability in Comprehension

Conference Paper · September 2019

DOI: 10.1109/ICSME.2019.00085

5 authors, including:

John Johnson
University of Nebraska at Lincoln
1 PUBLICATION **2** CITATIONS
SEE PROFILE

Jairo Aponte
National University of Colombia
23 PUBLICATIONS **751** CITATIONS
SEE PROFILE

Bonita Sharif
University of Nebraska at Lincoln
74 PUBLICATIONS **1,478** CITATIONS
SEE PROFILE

An Empirical Study Assessing Source Code Readability in Comprehension

John C. Johnson*, Sergio Lubo†, Nishitha Yedla‡, Jairo Aponte† and Bonita Sharif*
*Department of Computer Science and Engineering, University of Nebraska-Lincoln, Lincoln, Nebraska, USA 68588
†Dept. de Ingenieria de Sistemas e Industrial, Universidad Nacional de Colombia, Bogota, Colombia
‡Department of CSIS, Youngstown State University, Youngstown, Ohio, USA 44555
joh19267@umn.edu, slluboa@unal.edu.co, nyedla@student.ysu.edu, jhapontem@unal.edu.co, bsharif@unl.edu

***Abstract*—Software developers spend a significant amount of time reading source code. If code is not written with readability in mind, it impacts the time required to maintain it. In order to alleviate the time taken to read and understand code, it is important to consider how readable the code is. The general consensus is that source code should be written to minimize the time it takes for others to read and understand it. In this paper, we conduct a controlled experiment to assess two code readability rules: nesting and looping. We test 32 Java methods in four categories: ones that follow/do not follow the readability rule and that are correct/incorrect. The study was conducted online with 275 participants. The results indicate that minimizing nesting decreases the time a developer spends reading and understanding source code, increases confidence about the developer's understanding of the code, and also suggests that it improves their ability to find bugs. The results also show that avoiding the do-while statement had no significant impact on level of understanding, time spent reading and understanding, confidence in understanding, or ease of finding bugs. It was also found that the better knowledge of English a participant had, the more their readability and comprehension confidence ratings were affected by the minimize nesting rule. We discuss the implications of these findings for code readability and comprehension.**

***Index Terms*—program comprehension, source code readability rules, correctness, controlled experiment**

I. Introduction

Program comprehension is a precursor to many software engineering activities such as code review, bug fixing, refactoring, or implementing new features [1]–[3]. There are many aspects to how source code is written, such as the adoption of good design, formatting, coding and naming conventions (among others) that could affect its comprehension. The code developers write is read not only by them but also by others in the team and beyond. Hence, it is crucial that code not only satisfy functional requirements but that it is written in a way that facilitates easy readability. The effects of readable code go far beyond just aesthetics. It can have significant effects on maintenance time as developers build mental models [1], [3], [4] of the software during maintenance tasks. Natural language prose such as English has certain metrics like the Flesch's Reading Ease Score [5] by which readability is measured. For source code, such metrics do not work as well because source code is inherently quite different from natural language [6]. Source code is rich in syntax and has an added layer of semantics involved.

In recent work, Scalabrino et al. looked at more than a hundred features in code snippets but none really correlated with any useful operationalization of code understanding [7], [8]. The above methods stem from Borstle et al.'s work [9] and take into account the program's identifiers, keywords, statements and word length in syllables. Trockman et al. [10] reanalyzed the dataset by Scalabrino et al. with the goal of producing a metric for understandability, however, found only a small significant correlation with understandability based on combining multiple features that describe the code snippets.

The question still remains: How does one operationalize code readability? In order to move towards this goal and to study the problem in greater detail from a developer perspective, we report on results from a large empirical study conducted with 275 participants to determine the readability of source code snippets. A previous study by Scalabrino focused heavily on the source code artifact itself, in that metrics were inferred based on the code without asking developers comprehension questions on correct and incorrect methods. In this paper, we present a first study that looks at readability in both correct and incorrect code where the level of understanding is measured with multiple questions (multiple-choice comprehension and logic correctness questions).

McCabe's cyclomatic complexity (CC) has been widely used as a metric for the complexity of a program. One of its original usages was to track and restrict the complexity of routines during program development by using a threshold that should never be exceeded. Empirical studies, however, have shown that CC does not match a developers perceived readability [11]–[13]. One problem of CC is that it gives the same weight to all control structures [14]. To overcome this issue, there are proposed metrics for estimating the comprehension effort for typical control structures of imperative programming languages [15], [16]. The aim of these proposals is to assign a cognitive weight to each control structure. However, there is little empirical evidence to support the different weights assigned. An exception is a recent study in which Ajami et al. empirically assessed the difficulty of reading loops, ifs, predicates, and the effects of idioms on comprehension [14]. The contributions of this paper are as follows.

- Two readability rules, minimize nesting and avoid do-while, are investigated with respect to a set of comprehension measures.

- Both correct and incorrect versions of the code are investigated with respect to whether or not they follow the two readability rules.
- A comparative analysis of rule following is done for correct solutions in both rules.

In this study, we provide empirical evidence related to two widely recommended programming practices aimed at improving code readability by simplifying loops and logic. Empirical results provide evidence to improve our current programming practices and, eventually, identify structural features of the code that help to define better readability metrics. The minimize nesting and avoid do-while rules [17] were chosen because they are old and popular coding practices without any empirical evidence of their actual effect on perceived readability and comprehension. This paper seeks to provide that evidence via a controlled experiment. The results show support for the minimize nesting rule both in terms of efficient reading time and ability to find a bug. However, no support was found for avoiding the do-while statement for code that was correct or incorrect. We consider this study as a first attempt to systematically investigate both correct and incorrect code for readability rules.

II. Related Work

A variety of source code features have been studied to identify the extent to which they affect software understandability. In this section we report on experiments aimed at providing empirical evidence of the impact of source code characteristics on program understandability, within the context of developers reading and understanding short code snippets.

A. Surface-level Features

There are surface-level features that, when taken together, can greatly affect the understandability of an entire software system. In this category, aspects such as names of identifiers, comments, formatting, and syntax highlighting, among others, have been empirically studied. As identifiers represent a major part of the program text and their semantic content is high, they have been the focus of many studies. For instance, evidence has shown that full-word identifiers increase readability more than identifiers of a single letter or that use abbreviations [18], camel-case notation turns out to be better than underscore style to reduce effort and increase code understanding, especially for beginning programmers [19], and longer but more informative identifier names improve code comprehension when an in-depth understanding of the code is required [20]. By the same token, individual terms in source code were successfully related to measures of developers' cognitive load [21]. Moreover, linguistic antipatterns in source code significantly increases the cognitive load of developers during program comprehension tasks.

Syntax highlighting allows the developers to display source code in different colors and fonts according to the category of terms. Empirical studies have shown, for instance, that although developers feel that the color code fragments are easier to read and more aesthetically pleasing, the visual effort is not significantly higher for black-and-white code than for color code [22]. Similarly, within the more specific context of preprocessor directives used in software-product-line engineering, a family of controlled experiments suggest that programmers prefer background colors and these have the potential to enhance program understanding [23].

Other studies have chosen to analyze the possible effects of groups of lexical and syntactical aspects on the ease of understanding. Regarding coding style, the models of readability proposed by Buse and Weimer [24], Dorn [25], Posnett et al. [26], and Scalabrino et al. [8] have been used as the basis for investigating the impact of Java coding practices on the readability perceived by software developers [27]. The data collected show evidence that most practices improve readability, a few hinder it, and some do not seem to have any significant effect. Scalabrino et al. has gone on to be cited in a number of studies, including Daka et al. [28], Srinivasulu et al. [29], and Pantiuchina et al. [30]. A study on the same subject shows that code formatting aspects (e.g., spaces, variables initialization, code alignment, order of instructions, operator overloading, function call order), often considered trivial, can deeply impact programmers' comprehension [31]. A similar eye tracking experiment assessed the importance of structural and textual features of source code for its readability and comprehensibility [32]. The results reveal that the absence of textual features increases the average fixation duration, which may indicate that the textual features are important for the comprehension of the code.

B. Structural Features

Structural features of source code such as control flow statements and logical conditions introduce jumps and branches that can make it extremely hard to understand and maintain. The effects of these features on the program reading performance of developers have been a concern that is reflected in many coding practices recommended in countless programming texts and standards. However, there is little empirical evidence to confirm or refute the effectiveness of such coding practices. Empirical evidence suggests that the read/process loop would be easier to understand than the process/read loop, at least for students [33]. Furthermore, for loops are significantly harder than ifs, and loops counting down are slightly harder than loops counting up [14]. Empirical findings also show that neither perceived readability nor code comprehension are affected by method chain variants [34]. From an educational viewpoint, the evidence has allowed identification of several frequent fixes to unreadable code that includes improvements to variable and method names, the creation of new methods to reduce code duplication, simplifying if conditions and structures, and simplifying loop conditions [35].

III. Experimental Design

The *goal* of this study is to *analyze* two readability rules, i.e., good programming practices, for the *purpose* of providing empirical evidence of their impact on the understandability of

TABLE I
EXPERIMENT OVERVIEW

Goal	Analyze two readability rules for the purpose of providing empirical evidence of their impact on the understandability of source code.
Experimental factors	Logical correctness, Rule following
Dependent variables	Comprehension Time, Comprehension Confidence, Level of Understanding, Readability Rating

source code. The *quality focus* is the effectiveness (level of understanding reached by a reader), the efficiency (time spent by a reader), and the level of confidence a developer has about their own understanding of source code. The *perspective* is of researchers who want to evaluate to what extent readability rules influence source code understandability, in the *context* of developers reading and understanding short Java methods that follow/do not follow the two readability rules and are logically correct/incorrect. We focus on the programming practices aimed at making the control flow in the source code easy to read. Specifically, we consider two coding practices that we present as readability rules - R1: Minimize nesting and R2: Avoid do-while loops

A. Research Questions and Hypotheses

The empirical evidence we seek might support or refute the idea that R1 (minimize nesting) and R2 (avoid do-while) help a developer when understanding source code. With that in mind, we define and measure three dependent variables: the time in seconds spent by each person reading and understanding the code, the confidence level of the person in her own degree of understanding, and her actual understanding level. Moreover, we aim at addressing the following research questions:

RQ1a: Does the minimize nesting rule reduce the *time* a developer spends in understanding source code?

RQ1b: Does the minimize nesting rule increase the *level of confidence* a developer has about their own understanding of source code?

RQ1c: Does the minimize nesting rule improve the developer's *level of understanding* when reading source code?

RQ2a: Does the avoid do-while rule reduce the *time* a developer spends in understanding source code?

RQ2b: Does the avoid do-while rule increase the *level of confidence* a developer has about their own understanding of source code?

RQ2c: Does the avoid do-while rule improve the developer's *level of understanding* when reading source code?

We control for two independent variables: *Correctness* and *Rule Following*. Correctness indicates whether the Java method satisfies the problem specification, while Rule Following refers to whether the Java code follows a readability rule. Table I shows an experiment overview, while Table II presents the null and alternative hypotheses for R1. There are similar hypotheses for R2.

TABLE II
NULL AND ALTERNATIVE HYPOTHESES (R1)

Null Hypotheses	Alternative Hypotheses
H10: The use of the minimize nesting rule does not produce a significant reduction on the *time* a developer spends understanding source code.	H1: The use of the minimize nesting rule produces a significant reduction on the *time* a developer spends understanding source code.
H20: The use of the minimize nesting rule does not produce a significant increase in the *level of confidence* a developer has about their own understanding of source code.	H2: The use of the minimize nesting rule produces a significant increase in the *level of confidence* a developer has about their own understanding of source code.
H30: The use of the minimize nesting rule does not produce a significant improvement on the *level of understanding* a developer reaches when reading source code.	H3: The use of the minimize nesting rule produces a significant improvement on the *level of understanding* a developer reaches when reading source code

B. Objects and Tasks

For each readability rule, we chose four basic algorithmic problems, which are presented in Tables III and IV. For each programming problem, we designed four solutions and each one of them is a Java method. Thus, the j^{th} solution (or method) for the P_k problem is denoted as P_kS_j, where k $\in$ 1..4, and j $\in$ 1..4. With respect to the two experimental factors, the four solutions to P_k represent the four different treatments namely, T1, T2, T3, and T4, because they have the following characteristics:

- P_kS_1 is a correct solution to P_k that follows the readability rule we are interested in. (Treatment T1)
- P_kS_2 is another correct solution to P_k that does not follow the readability rule. (Treatment T2)
- P_kS_3 is an incorrect solution to P_k that follows the readability rule. (Treatment T3)
- P_kS_4 is an incorrect solution to P_k that does not follow the readability rule. (Treatment T4)

We use 16 methods to test one single readability rule. The methods do not call other methods or use objects of other classes. Thus, the participant does not need to study additional code to understand these methods. Figure 1 shows P_1S_1 and P_1S_4 for R1. In addition to rule following and logical correctness, the methods were measured in terms of several other features, detailed in Table VI. The main tasks created for the purpose of the experiment fall into two categories: *method analysis* or *method comparison*. Within a method analysis task, the participant reads a problem statement, analyzes one of the four methods proposed as solutions for that problem, rates the readability of the method, answers a multiple-choice comprehension question, rates her confidence in her level of comprehension, and finally, says whether the method is correct. On the other hand, in a method comparison task, the participant reads the problem statement P_k and analyzes the two correct solutions for that problem (P_kS_1 and P_kS_2). After that, the participant selects the most readable solution and explains the rationale behind her choice.

TABLE III
PROBLEMS FOR R1 (MINIMIZE NESTING RULE)

Problem	Problem Statement
P1	Given three integer numbers, the method must return the greatest.
P2	Given a mark, which is an integer between 0 and 100, the method must return a letter. Letter A if $mark \geq 90$; B if $mark \in [80, 90)$; C if $mark \in [70, 80)$; D if $mark \in [60, 70)$; and F if $mark < 60$.
P3	Given the body mass index (bmi), the method must return the category in which the index is located. The category is "very severely underweight" if $bmi \leq 15$; "severely underweight" if $bmi \in [15, 16)$; "underweight" if $bmi \in [16, 18.5)$; "healthy weight" if $bmi \in [18.5, 25)$; "overweight" if $bmi \geq 25$.
P4	Given three integers, the method must count how many of them are positive numbers.

TABLE IV
PROBLEMS FOR R2 (AVOID DO-WHILE LOOPS RULE)

Problem	Problem Statement
P1	Ask the user to answer a multiple choice question. Show the question, get the user response, and end when the user chooses the correct answer or when she decides not to try more (typing 'q' or 'e')
P2	Let's assume that we want to force a user to change her password. The user must give a new password that must have 4 different characters. Besides, the given password must be different to the old one. The method receives the old password as a parameter and asks the user to give the new one.
P3	Add the positive numbers in an array. The method receives an integer array as a parameter and must return the sum of the positive numbers in the array.
P4	Count the occurrences of a character. This method receives a string and a character as parameters. It must count and return the number of occurrences of the character in the string.

```
public int findTheBiggest(int numOne, int numTwo, int numThree) {
  int theLargest = numOne;
  if (numTwo > theLargest) {
    theLargest = numTwo;
  }
  if (numThree > theLargest) {
    theLargest = numThree;
  }
  return theLargest;
}
```

```
public int findTheBiggest(int numOne, int numTwo, int numThree) {
  int theLargest = 0;
  if (numTwo < numThree) {
    theLargest = numThree;
  } else {
    if (numTwo > numOne) {
      theLargest = numTwo;
    }
  }
  if ((numOne > numTwo) && (numOne > numThree)) {
    theLargest = numOne;
  }
  return theLargest;
}
```

Fig. 1. Two methods for Problem 1. The one on top is correct and follows R1 (P_1S_1). The bottom one is incorrect and does not follow R1 (P_1S_4)

TABLE V
ONE TRIAL FOR R1 (MINIMIZE NESTING RULE). S1-S4 CORRESPOND TO THE DIFFERENT TREATMENTS FOR EACH PROBLEM. S1 IS BOTH LOGICALLY CORRECT AND FOLLOWS THE READABILITY RULE IN QUESTION, S2 IS LOGICALLY CORRECT AND BREAKS THE RULE, S3 IS LOGICALLY INCORRECT AND FOLLOWS THE RULE, AND S4 IS LOGICALLY INCORRECT AND BREAKS THE RULE.

method analysis tasks				method comparison task	
P1S4	P2S3	P3S2	P4S1	P4S1	P4S2

C. Participants

The study was conducted with 275 participants. Out of which, 208 participants were undergraduates in computer science, 49 were pursuing Masters in computer science and 61 were currently working as professional developers. Participants were from four different countries. While the first language of the majority of participants was Spanish, they also had their primary language as English, Arabic, German, Telugu and French. Nineteen participants had more than 5 years of experience as professional programmers, 15 participant's professional experience ranged between 3 and 5 years, 31 participant's experience ranged between 1 and 3 years, 39 had less than one year experience, and 171 had no professional programming experience. More than 75% of the participants had less than 3 years of experience with Java and the remaining participants has been working with it from 3 years to more than 5 years. Almost all the participants had general programming experience ranging from 1 year to more than 5 years. Eight participants self-reported their knowledge of Java programming language as very good, 43 reported it as good, 121 reported it as satisfactory and the rest reported that they had poor knowledge of the Java programming language.

D. Study Procedure and Instrumentation

All of the data was collected through an online questionnaire designed using the Qualtrics System. The first questions collected demographic data to assess programming experience of the participants, their knowledge of Java, and their reading skills in English. A within-subjects design is used, so that all the subjects serve in all the treatments. Thus, the rest of the questionnaire is divided in two parts. In part A, we ask the participant to perform 8 method analysis tasks, 4 related to the R1 rule and 4 to R2. In part B, the participant carries out one method comparison task associated with R1 and another one with R2. As a closing section (post-questionnaire), we asked them to rank the importance of R1 and R2, for writing readable code.

To avoid "carryover effects", each participant is given the four analysis tasks associated with a rule in a randomly selected order, and besides that, each one of them corresponds to a different problem. For instance, Table V shows the part of a trial associated with R1 in which the participant analyzes the solution S4 for the problem P_1, the solution S_3 for problem P_2, the solution S_2 for problem P_3, and the solution S_1 for problem P_4. In addition, she carries out a method comparison task where she compares the readability of S_1 and S_2 for

TABLE VI

CODE SNIPPET METRICS. FOR CLARIFICATION, AN "IDENTIFIER" IS A VARIABLE NAME AND "MAX PAREN NEST LEVEL" REFERS TO THE MAXIMUM NUMBER OF NESTED PARENTHESIS ON ANY ONE LINE. AVG. IDENTIFIER LENGTH WAS CALCULATED BY TAKING THE AVERAGE CHARACTER LENGTH OF ALL OCCURRENCES OF ALL IDENTIFIERS.

Name	Logically Correct?	Follows Rule?	Lines	Characters Per Line	Max Indentation	Average Indentation	Identifiers	Avg Identifier Length	Max Occurrences Of any Identifier	If Statements	Max Paren Nest Level	Max Conditions Per If Statement
FindTheBiggest1	yes	yes	10	23.4	2	1	4	7.5	6	2	1	1
FindTheBiggest2		no	17	20.06	3	1.71				3		
FindTheBiggest3	no	yes	13	26.92	2	1.08					2	2
FindTheBiggest4		no	14	22.36	3	1.29			5			
FindGrade1	yes	yes	15	14.53	2	1.13	1	5	5	4	1	1
FindGrade2		no	21	15.48	5	2.43	2		7			
FindGrade3	no	yes	15	17.2	2	1.13	1		8			2
FindGrade4		no		21.07		1.2	2					
BodyMassIndex1	yes	yes	16	19.94	2	1.13	1	3	5	4	1	1
BodyMassIndex2		no	21	20.57	5	2.43	2	4.5	7			
BodyMassIndex3	no	yes	16	22.88	2	1.13	1	3	8			2
BodyMassIndex4		no	21	22.81	5	2.43	2	4.5				
CountPosNumbers1	yes	yes	13	16.31	2	1.08	4	6.25	5	3	1	1
CountPosNumbers2		no	25	16.88	4	2.08	3	6.67		7		
CountPosNumbers3	no	yes	13	16.54	2	1.08	4	6.25		3		
CountPosNumbers4		no	25	16.92	4	2.08	3	6.67		7		
WhoIsTheAuthor1	yes	yes	27	31.44	3	1.63	3	5.3	5	2	2	2
WhoIsTheAuthor2		no		31.3		1.59						
WhoIsTheAuthor3	no	yes	25	33.32		1.56						
WhoIsTheAuthor4		no	26	32.04		1.54						
ChangePassword1	yes	yes	23	25.57	6	2.26	6	3..83	9	2	2	2
ChangePassword2		no	24	24		2.21						
ChangePassword3	no	yes	22	25.82	5	1.95						
ChangePassword4		no	23	24.17		1.91						
SumPositiveNums1	yes	yes	12	18.58	3	1.25	4	4.75	5	1	1	1
SumPositiveNums2		no	16	16.63						2		
SumPositiveNums3	no	yes	12	18.67		1.33				1		
SumPositiveNums4		no	16	16.69		1.31				2		
CountLetter1	yes	yes	14	18.93	3	1.43	5	5.4	5	1	2	1
CountLetter2		no	18	17		1.39				2		
CountLetter3	no	yes	13	19.38		1.31			4	1		
CountLetter4		no	17	17.24		1.29				2		

problem P_4. The trial is completed with a similar arrangement of tasks in order to test R2.

E. Verifiability

The entire replication package for the experiment is available at https://bit.ly/2OWNkZG. We provide the tasks, responses, questionnaires, and analysis performed as part of the package.

IV. EXPERIMENTAL RESULTS

After the data was collected, we performed t-tests and found effect sizes for accuracy, completion time, confidence results, and readability assessment with regard to whether the snippet presented followed the rule and if the snippet was logically correct. The p value, or probability value, is the probability of getting a value greater than or equal to the observed value if the null hypothesis is true. Very small p values ($p \leq 0.05$) are statistically significant, meaning that in cases where $p \leq 0.05$, we can reject the null hypothesis (shown in table II). Cohen's d is an effect size (a statistic used to measure the magnitude of a phenomenon) based on the standardized difference between two means. Cohen's d allows us to interpret effect size independent of the scale, where the magnitude of the value indicates the magnitude of the effect, and the sign of the value (positive or negative) indicates the nature of the effect. All t-tests performed for this analysis were going in the same direction, that is to say that all comparisons were comparing logically correct snippets to logically incorrect snippets, and rule following snippets to rule breaking snippets, and not the other way around. This is relevant because switching the order of the comparison will affect the sign of the Cohen's d value, which is positive for positive relationships and negative for inverse relationships.

A. Level of Understanding

The level of understanding was measured by two different questions: one multiple choice question about the nature of the code snippet, which we refer to as the comprehension question, and one question about whether or not the snippet was logically correct. From the data that was collected, shown in Table VII, it appears that following the minimize nesting rule has a very small but significant effect on how accurately a reader can answer multiple choice questions about a given code snippet (p = 0.0495, Cohen's d = 0.1224). For snippets testing the minimize nesting rule, whether or not a snippet was logically correct played a bigger role in question accuracy than readability rule following. For snippets testing the minimize nesting rule, logical correctness had a small effect on the comprehension question (p <.0001, Cohen's d = 0.3901), and a medium effect on the logical correctness question (p <.0001, Cohen's d = 0.6987). For snippets testing the avoid do-while rule, logical correctness had a medium effect on the logical correctness question (p <.0001, Cohen's d = 0.5551), but an insignificant and very small effect on the comprehension question(p = 0.2573, Cohen's d = 0.0706). Following the avoid do-while rule had very small and insignificant effects on both the comprehension question (p = 0.0891, Cohen's d = -0.1059) and the logical correctness question (p = 0.7884, Cohen's d = -0.0167). Finally, for snippets testing the avoid do-while rule, a snippet's logical correctness had a very small, insignificant effect on the comprehension question (p = 0.0891, Cohen's d = 0.1059).

TABLE VII
RESULTS FOR T-TEST FOR R1 AND R2. EACH DEPENDENT VARIABLE IS SHOWN PER RULE FOLLOWING AND CORRECTNESS CRITERIA.

		Readability	Comprehension	MC Accuracy	Logical Accuracy	Read Time	Answer Time
r1 correctness	p	0.79348	0.18317	<0.0001	<0.0001	0.11926	0.00505
	Cohen's d	-0.0163	-0.08292	0.39009	0.69869	-0.09707	-0.17493
r1 rule following	p	<0.0001	<0.0001	0.04953	0.25733	<0.0001	0.06023
	Cohen's d	1.09733	0.45175	0.12242	0.07056	-0.48068	-0.11712
r2 correctness	p	0.54257	0.02688	0.08911	<0.0001	0.23294	0.01953
	Cohen's d	-0.03792	-0.13799	0.10595	0.55515	0.0743	-0.14562
r2 rule following	p	0.23558	0.85384	0.08911	0.78837	0.73196	0.90999
	Cohen's d	0.07389	0.01147	-0.10595	-0.01672	0.02133	-0.00704

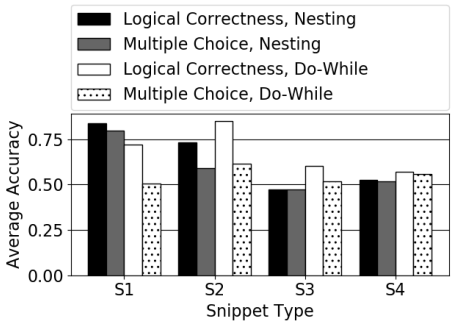

Fig. 2. Average Question Accuracy by Snippet Type. S1 snippets were both logically correct and followed the readability rule, S2 snippets were correct and broke the rule, S3 snippets were incorrect and followed rule, and S4 snippets were incorrect and broke the rule. Note that logically incorrect snippets had much lower accuracy on average than correct snippets, except for the multiple choice questions for snippets testing the avoid do-while rule.

Finding: Whether or not a snippet was logically correct had a much bigger impact on question's level of understanding(accuracy) than following either of the readability rules. Only the minimize nesting rule had a significant effect on the level of understanding (Figure 2).

B. Comprehension Time

Completion time was measured by two metrics: time to read the snippet, and time to answer the questions. As shown in table VII, the minimize nesting rule was once again the only readability rule that had a significant impact on completion time. For reading time, following the minimize nesting rule had a small but significant effect on reading time (p <0.0001, Cohen's d = -0.4807), and a very small, insignificant effect on answer time (p = 0.0602, Cohen's d = -0.1171). Following the avoid do-while rule had insignificant and very small effects on both read time (p = 0.7320, Cohen's d = 0.0213) and answer time (p = 0.9100, Cohen's d = -0.0070). Logical correctness had very small but significant effects on answer time for snippets testing both the minimize nesting rule (p = 0.0051, Cohen's d = -0.1749) and the avoid do-while rule (p = 0.0195, Cohen's d = -0.1456), but very small and insignificant effects on read time for both minimize nesting (p = 0.1193, Cohen's d = -0.0971) and avoid do-while (p = 0.2329, Cohen's d = 0.0743) snippets.

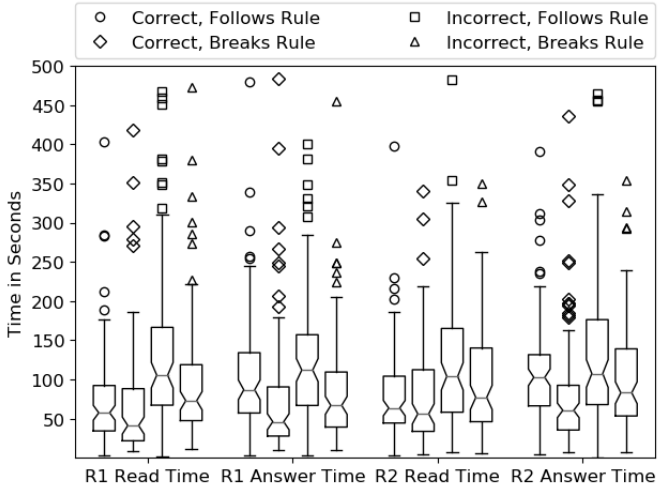

Fig. 3. Average Completion Time by Snippet. Note that the incorrect, rule-following snippets consistently took the longest to both read and answer, while the correct, rule breaking snippets consistently took the least time.

Finding: Logically correct snippets took less time to answer questions about, and that snippets that followed the minimize nesting rule took less time to read, on average (Figure 3).

C. Readability Assessment

Readability was measured by the readability rating given by participants for each code snippet that they saw. According to our results, shown in Table VII, only following the minimize nesting rule had a large, significant effect on the readability rating (p <.0001, Cohen's d = 1.0973), which was the most significant and largest effect measured in this study. Code correctness had very small, insignificant effects on the readability rating for both minimize nesting snippets (p = 0.7935, Cohen's d = -0.0163) and avoid do-while snippets (p = 0.5426, Cohen's d = -0.0379). Following the avoid do-while rule also had a very small, insignificant effect on readability rating (p = 0.2356, Cohen's d = 0.0739).

Finding: Following the minimize nesting rule made readers perceive code as more readable, and that both logical correctness and following the avoid do-while rule had no significant effect on perceived readability (Figure 4).

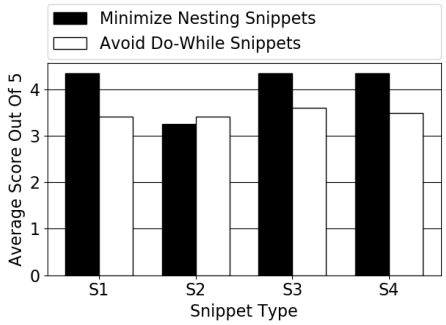

Fig. 4. Average Readability Rating by Snippet. S1 snippets were both logically correct and followed the readability rule, S2 snippets were correct and broke the rule, S3 snippets were incorrect and followed rule, and S4 snippets were incorrect and broke the rule. Note that scores for snippets types are relatively constant except for logically correct snippets that broke the minimize nesting rule, which were rated much lower than the others.

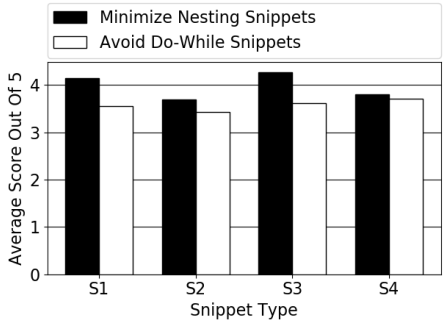

Fig. 5. Average Comprehension Confidence Rating by Snippet. S1 snippets were both logically correct and followed the readability rule, S2 snippets were correct and broke the rule, S3 snippets were incorrect and followed rule, and S4 snippets were incorrect and broke the rule. Note the fluctuation in score for minimize nesting snippets compared to the relatively stagnant scores for avoid do-while snippets.

D. Comprehension Confidence

A self-rated comprehension confidence question measured how confident participants were in the accuracy of their answers. Table VII shows the results of our t-tests for this metric. Once again, the only rule that had a significant impact on comprehension confidence was the minimize nesting rule, with a small effect ($p < .0001$, Cohen's $d = 0.4517$), while following the avoid do-while rule had a very small, insignificant effect ($p = 0.8538$, Cohen's $d = 0.0115$). Whether or not a snippet was logically correct had an insignificant and very small effect on confidence for snippets testing the minimize nesting rule ($p = 0.1832$, Cohen's $d = -0.0829$). On the other hand, for snippets testing the avoid do-while rule, the effect that logical correctness had was very small, but significant ($p = 0.0269$, Cohen's $d = -0.1380$). The negative effect size here suggests that logically correct code made readers less confident in their comprehension of it.

Finding: Following the minimize nesting rule made readers more confident in their understanding of a snippet. We also found that for the snippets testing the avoid do-while rule, logically correct code made participants less sure of their understanding (Figure 5).

E. Readability Rating

For the qualitative ranking, participants were shown two code snippets side by side, one that followed a readability rule and one that did not, and asked to rank them in order of readability. For these qualitative ranking questions, there were some participants that did not complete the question, with 259 out of 275 answering the question regarding the minimize nesting rule and 258 out of 275 answering the question regarding the avoid do-while rule. We decided to use the answers to these incomplete surveys for the rest of the analysis because only these last questions were omitted, and the rest of the survey was still taken. The results suggest that both the minimize nesting rule and the avoid do-while rule are important. 86.82% of respondents ranked the rule-following snippet higher for the minimize nesting rule, and 67.44% did so for the avoid do-while rule. Given the results from the rest of the survey, which showed no significant impact in any of the readability metrics for the avoid do-while rule, it is possible that these results are more related to what readers felt that readable code should look like or what their personal preferences were, rather than how readable a snippet actually was, possibly biasing them against do-while loops, which are generally used less frequently than while loops. An alternative possibility is that the differences in readability were just more pronounced when rule following and rule-breaking snippets were presented side by side. When examining the reasons given by respondents for their answers, differing patterns can be seen in the two rules.

1) Minimize Nesting: For the minimize nesting snippets, of the 35 respondents that ranked the rule-breaking snippet higher, 7 gave explanations implying that they had mis-clicked and meant to rank in the other way (ie. saying that they liked the lack of nesting in the code they ranked first when they actually ranked the code with more nesting higher), 13 made arguments related to how the code would run, including some who thought that there were errors present in the rule following code and some who thought the rule-breaking code would run faster, 8 provided no reasoning at all, and 7 said that they genuinely thought the rule-breaking code was more readable. Of the 224 participants that ranked the rule-following code higher, none gave explanations that implied a mis-click, 8 made arguments solely about how the code would run, 32 provided no answer, and 184 appeared to have genuinely thought the rule-following code was more readable. Most responses that ranked the rule-following snippet higher cited either overly complex if statements or code length as their reason.

2) Avoid do-while: For avoid do-while snippets, of the 84 participants that ranked the rule-breaking snippet higher, 5 gave explanations that implied a mis-click, 13 made arguments about code performance (for example, they thought there was an error in the code of the rule-following snippet), 17 provided

no explanation, and 49 said that they genuinely thought the do-while code was more readable. Of the 174 participants that said the rule-following snippets were better, 3 gave explanations that implied a mis-click, 5 gave performance-related explanations, 3 said that they had no preference, 22 provided no explanation, and 141 said that they thought the rule-following code was more readable. Of all 258 responses collected, 93 of them mentioned the dichotomy between while and do-while loops as one of their reasons for their choice, with 28.57% (24 people) of those who ranked the do-while snippet higher mentioning it and 39.66% (69 people) of those who ranked the while snippet higher. Some respondents said that they felt the difference was down entirely to personal preference, for example one response for comparing *WhoIsTheAuthor1* (R2-P1S1) and *WhoIsTheAuthor2* (R2-P1S2) said: "*Both functions have easy to understand code that is written in a way that anyone can determine what the purpose of it is, but in terms of function 2, more people would prefer that style of writing code because they might not understand the do while loop well enough to understand it*". Other responses were similar, with respondents citing the perceived poor readability of do-while loops as their reason for ranking segments using them lower, even though the snippets presented were equally readable to them.

Finding: Participants rated both minimize nesting and avoid do-while rules as important.

F. Secondary Factors: Native Language, Java Knowledge, and English Knowledge

When subdividing the population by native language, it was discovered that English speakers had significantly shorter read and answer times than any other group, with answer times being 12.29% faster and read times 31.06% faster than native Spanish speakers. In all other areas, Spanish and English speakers performed very similarly, with Spanish speakers usually performing slightly higher than English speakers. Additionally, small, significant correlations between answer time and logical correctness for snippets testing the minimize nesting rule were found for both Spanish speakers (p = .003346, Cohen's d = -0.2182) and English speakers (p = 0.002667, Cohen's d = -0.2198) which was obfuscated in the complete data set by the difference between answer times of the populations. When participants are broken out into separate populations based on their self-assessed knowledge of English, several interesting observations can be made. The group with the highest read and answer times are those with a 'Satisfactory' knowledge of English (this option was the middle of 5 choices), and the farther away from the middle one's knowledge of English was (in either direction), the faster they read and answered the questions. Knowledge of English had an interesting effect on the p values and effect sizes for readability rating and comprehension confidence in relation to following the minimize nesting rule. As knowledge of English increased, p-values decreased and effect sizes increased, from small and insignificant correlations for both readability rating (p = 0.8376, Cohen's d = .1214) and comprehension confidence (p = 0.4475, Cohen's d = -0.4564) among those with very poor knowledge of English to a significant and very large correlation for readability rating (p <.0001, Cohen's d = 1.3252) and a significant and medium correlation for comprehension confidence (p <.0001, Cohen's d = 0.6893). When breaking participants down by Java knowledge, the closest thing to a half and half split was between those with less than 'Satisfactory' knowledge ('Poor' or 'Very Poor') (n = 560) and those with more ('Satisfactory', 'Good', or 'Very Good') (n = 1,456). Observing these groups showed that as Java knowledge increased, readability rating, multiple choice and logical correctness question accuracy, and comprehension confidence increased while read and answer times decreased (Figure 6). This effect was also seen between population groups with less than and more than 1 year of Java experience, and in this case was slightly more pronounced for English speakers.

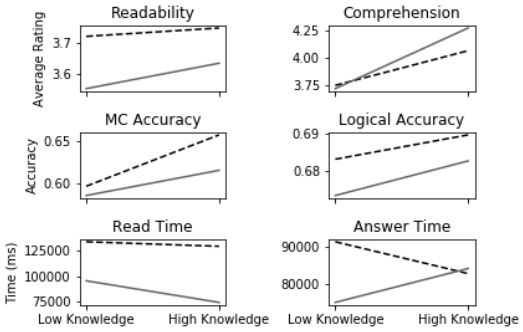

Fig. 6. Comparison of average values for metrics after breaking the population up into 4 groups based on first language and self-reported knowledge of Java. Dotted lines represent Spanish speakers and solid lines represent English speakers. For most metrics, English and Spanish speakers saw similar effects. With the exception of answer time, all of the correlation pairs had the same sign.

Finding: English speakers had significantly shorter read and answer times than any other group but most other metrics had similar effects.

G. Post-Questionnaire Results

Participants were asked to rank the importance of minimize nesting rule and avoid do-while loop rule on a scale of very important, somewhat important, neutral, somewhat not important and not important. For the minimize nesting rule, 133 participants ranked that it is very important, 93 participants ranked it as somewhat important, 31 participants ranked it as neutral and only one participant ranked it as somewhat not important. For the avoid do-while loop rule, 44 participants ranked it as very important, 85 participants ranked it as somewhat important, 88 participants ranked it as neutral, 20 participants ranked it as somewhat not important and 21 participants ranked it as not important.

V. Threats to Validity

We now discuss the various threats to validity and how we mitigate them in this study design.

A. Internal Validity

Threats to internal validity are the exogenous factors that may influence the observed results. Firstly, to mitigate the influence of human factors we collected demographic data to make sure that the participants had a proficient Java programming level, acceptable English reading skills, and no reading disorder. Apart from that, participants were expressly instructed (i) not to use other tools or copy and paste the code to answer the respective questions, (ii) not to assume the survey as an assessment of their programming skills, (iii) not to worry about the required time to complete the tasks, (iv) conduct the survey in a single session, i.e., without pauses or interruptions, and (v) be focused on understanding each method, since it is presented only once before each set of questions. In spite of that, it may have been possible for some of the participants who performed the experiment without our supervision (about 47 percent of the total) to have used additional tools to understand the code, or to have paused during the experimental session. Secondly, to avoid the effect of poorly designed experimental artifacts, before the experiment we conducted a pilot study with two graduate students who were both working for software companies. The results allowed us to detect unclear problems' descriptions, deficiencies in the forms used for collecting responses, and too complex code snippets or questions. Moreover, we assessed the overall feasibility and the time required to complete each of the tasks of the experiment. Finally, with the goal of controlling order effects, we organized all trials of the experiment in such a way that for a particular readability rule, the participant analyzed four different problems and each of the four solutions proposed for these problems is in a different treatment. Apart from that, the four solutions were presented in a random order.

B. External Validity

External validity threats reduce the degree of generalization of our results. First, since we restricted the scope of the experiment to test the impact of rules related to simplifying loops and logic, the results could be valid for languages with similar loop and conditional constructs. However, if the code snippets used are more complex and have dependencies on other methods, the results may be somewhat different. Second, only 22% of the subjects had industrial experience and most of them were graduate and undergraduate students, so that it is plausible that our conclusions cannot be fully applied to the population of professional developers. Therefore, replications involving industrial subjects or other types of source code samples are highly desirable and welcome.

C. Construct Validity

Construct validity deals with the gap, if any, between the theoretical concepts and the actual representation and measurement of them within the experiment. In our experiment there was no mathematical treatment of the concepts of program correctness and formal specification of problems. However, the natural language descriptions of the programming problems used were carefully reviewed to eliminate any ambiguity or lack of clarity that could lead to misunderstandings. In that sense, the pilot study confirmed that the inputs and expected outputs for each of the four programming problem are clear. The readability perception of a code is affected by many syntactic and semantic aspects, such as the type of syntactic structures used, the choice of names, the dependencies, the abstractions represented, etc. The presentation of the code can also influence readability, in aspects such as font size, line length, line height, and colors. To mitigate the influence of all these factors and isolate the effect of the two readability rules, we chose very simple problems, wrote short methods without dependencies and comments, and did not use colors. In addition, in the four solutions associated with a problem, we unified all the syntactic and presentation aspects of the code, in such a way that they only differed in compliance with the readability rule and in the logical correctness attribute. Regarding the accuracy of our time measures, the Qualtrics survey design features allow us to measure exactly the time in seconds spent by each subject when reading the problem statement and its solution. Apart from that, the participant was not allowed to revisit the code once they were done understanding the solution at hand. Assessing a person's level of understanding is a very complicated matter, even when considering a small piece of code. We chose to ask a single multiple-choice question, in which the participant must indicate which of 4 given statements is true about the method execution. It may have been possible for some of the participants to have answered the question correctly without having understood the code. More questions would have been desirable, of course, but there must be a balance between reducing the risk of measuring incorrectly and increasing the complexity of the questionnaire and its grading. Lastly, even within short and simple methods, several types of logical errors may exist (e.g., a wrong logical operator, an index out of bounds, an incorrect boolean expression, a wrong sequencing, etc.). In the design of the incorrect methods, we did not use a systematic strategy to build them as we cannot know how a developer will write an incorrect version. Thus, if the errors in the incorrect methods are of other types, the results could possibly be different.

D. Conclusion Validity

Conclusion validity addresses the proper selection of statistical tests to determine statistical significance of the study results. In all our analyses we use standard statistical measures, i.e. t-test and Cohens d, which are conventional tools in inferential statistics.

VI. Discussion and Implications

The main findings of the paper are that following the minimize nesting rule appears to have a significant impact on code readability, especially perceived readability and reading time. In contrast to that, the avoid do-while rule had no significant impact on any of the readability metrics that we collected.

For readability ratings, it was very clear that following the minimize nesting rule had the largest impact of any of the variables measured in this study. Following the avoid do-while rule had no significant effect on readability ratings.

Looking at the averages of completion times shown in Figure 3, there are several interesting observations to be made. First, the snippets testing the effects of the avoid do-while rule unilaterally took longer to both read and answer questions about than the minimize nesting snippets. This was likely because the average line length of the do-while snippets was about 17% more than that of the minimize nesting snippets. The reason that following the minimize nesting rule had such an effect on read time is likely because snippets that followed the minimize nesting rule were on average 6 lines shorter than those that did not. Logical correctness had a significant effect on completion time for both sets of snippets, meaning that it took participants longer to answer questions about snippets that were logically incorrect than it did for snippets that were correct. This increase in time can most likely be attributed to the fact that participants had to fill out another question for snippets that they thought were wrong, explaining why they believed the snippet to be incorrect. This additional question could have been what caused the increase in average answer time for incorrect snippets.

With regard to comprehension confidence, a measure of how well a readers believes that they understood a snippet, there were some interesting results. The variable that had the biggest effect on this metric was whether or not a snippet followed the minimize nesting rule, which suggests that snippets that minimize the amount of nesting will not only be easier and faster to read, but will also leave readers more confident in their understanding of it. The other significant correlation we found for comprehension confidence was with the logical correctness of snippets testing the avoid do-while rule, but the results that we got were somewhat counter-intuitive. According to the data we collected, logically correct code received slightly but significantly lower comprehension confidence ratings than logically incorrect code. This could be for any number of reasons, but one possibility is that once a bug is found, a snippet is known for sure to be incorrect, but if no bugs or errors are found in a snippet, there are two possibilities: either the snippet was actually correct, or the reader missed something. This uncertainty may have led to the decrease in comprehension confidence for logically correct code snippets.

For level of understanding, we found that logically correct code snippets had consistently higher question accuracy rates for questions about logical correctness, which suggests that readers were more likely to wrongfully assume that incorrect code was correct than vice versa, while following readability rules had no effect on these questions. It was also found that following the minimize nesting rule increased accuracy on the multiple-choice code comprehension question, which suggests that the minimize nesting rule has a real impact on how well readers are able to understand code. Following the avoid do-while rule had no effect on the accuracy of either question.

In summary, these results bring to light the importance of minimizing nesting, which is a worthwhile practice for educators to teach students learning to program. We envision this practice being promoted in all courses related to programming. It effectively reduces reading effort and improves the level of understanding of the reader. Even though avoiding the do-while rule did not have a significant effect on any of our understanding measures, it was still ranked as important in the post questionnaire and in the side by side comparisons of code snippets. It is possible that the questions that were asked did not have much relevance to the avoid do-while rule to show its significance. Such results also provide evidence-based support for coding style guidelines that the industry could adopt.

VII. Conclusions and Future Work

We investigate the impact of two coding practices (R1: Minimize nesting and R2: Avoid do-while loops) on source code readablity in comprehension/understandability. In the first part of our experiment, each of the 275 subjects performed eight method analysis tasks, four associated with R1 and four with R2. In the second part, each subject carried out one method comparison task associated with R1 and another one with respect to R2. The experiment ended asking the participant to rank the importance of both coding practices.

Regarding the minimize nesting rule (R1), we found that, indeed, this practice has the potential to decrease the time a developer spends reading and understanding source code, increases her level of confidence about her own understanding of the code, and also suggests that it improves her ability to find bugs. With respect to R2 (avoid do-while), the results indicate that avoiding the do-while statement saves reading time only for simple and short loops; however, the level of understanding is higher in cases where the code has a do-while, instead of a while statement, and it is natural to use it (i.e. the loop should execute at least once). For both practices, the perceived readability indicates that it is worthwhile to follow them. In the case of R1, 86.5% of the participants judged that, for the same problem, the method that follows the rule is more readable than the one that does not. For R2, 67.4% prefer the method that does not use the do-while structure. Lastly, 51.6% of the participants judged the practice of minimizing nesting as very important in order to write more readable code, while 36% say that is somewhat important. For R2, the majority of participants are neutral with respect to its importance for improving readability.

As part of our ongoing work, we plan to perform this experiment using an eye tracker to study eye gaze behavior while the subjects solve the tasks. The eye-tracking measures representing visual effort may reveal additional differences. Moreover, studying the gaze patterns could also reveal interesting aspects of how the developers read each of the versions of the source code used.

Acknowledgment

The authors would like to thank all the participants in the study. This work has been partly funded by the National

Science Foundation under grant numbers CCF 18-55756 and CCF 15-53573.

REFERENCES

[1] M.-A. Storey, "Theories, methods and tools in program comprehension: past, present and future," in *13th International Workshop on Program Comprehension (IWPC'05)*, 2005, pp. 181–191.

[2] E. Soloway and K. Ehrlich, "Empirical studies of programming knowledge," *IEEE Transactions on software engineering*, vol. SE-10, no. 5, pp. 595–609, 1984.

[3] R. Brooks, "Towards a theory of the comprehension of computer programs," *International journal of man-machine studies*, vol. 18, no. 6, pp. 543–554, 1983.

[4] A. M. Vans and A. von Mayrhauser, "Program comprehension during software maintenance and evolution," *Computer*, vol. 28, no. 8, pp. 44–55, 1995.

[5] R. Flesch, "A new readability yardstick," *Journal of Applied Psychology*, vol. 32, no. 3, pp. 221–233, 1948.

[6] T. Busjahn, R. Bednarik, A. Begel, M. Crosby, J. H. Paterson, C. Schulte, B. Sharif, and S. Tamm, "Eye movements in code reading: Relaxing the linear order," in *Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension*, 2015, pp. 255–265.

[7] S. Scalabrino, G. Bavota, C. Vendome, M. Linares-Vásquez, D. Poshyvanyk, and R. Oliveto, "f," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, 2017, pp. 417–427.

[8] S. Scalabrino, M. Linares-Vásquez, D. Poshyvanyk, and R. Oliveto, "Improving code readability models with textual features," in *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, 2016, pp. 1–10.

[9] J. Börstler, M. E. Caspersen, and M. Nordström, "Beauty and the beast: On the readability of object-oriented example programs," *Software Quality Journal*, vol. 24, no. 2, pp. 231–246, 2016.

[10] A. Trockman, K. Cates, M. Mozina, T. Nguyen, C. Kästner, and B. Vasilescu, ""automatically assessing code understandability" reanalyzed: combined metrics matter," in *Proceedings of the 15th International Conference on Mining Software Repositories*, 2018, pp. 314–318.

[11] K. B. Lakshmanan, S. Jayaprakash, and P. K. Sinha, "Properties of control-flow complexity measures," *IEEE Transactions on Software Engineering*, vol. 17, no. 12, pp. 1289–1295, 1991.

[12] G. K. Gill and C. F. Kemerer, "Cyclomatic complexity density and software maintenance productivity," *IEEE Transactions on Software Engineering*, vol. 17, no. 12, pp. 1284–1288, 1991.

[13] A. Jbara, A. Matan, and D. G. Feitelson, "High-mcc functions in the linux kernel," in *2012 20th IEEE International Conference on Program Comprehension (ICPC)*, 2012, pp. 83–92.

[14] S. Ajami, Y. Woodbridge, and D. G. Feitelson, "Syntax, predicates, idioms - what really affects code complexity?" in *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, 2017, pp. 66–76.

[15] J. Shao and Y. Wang, "A new measure of software complexity based on cognitive weights," *Canadian Journal of Electrical and Computer Engineering*, vol. 28, no. 2, pp. 69–74, 2003.

[16] Y. Wang, "Cognitive complexity of software and its measurement," in *2006 5th IEEE International Conference on Cognitive Informatics*, vol. 1, 2006, pp. 226–235.

[17] D. Boswell and T. Foucher, *The Art of Readable Code*. O'Reilly Media, Inc., 2011.

[18] D. Lawrie, C. Morrell, H. Feild, and D. Binkley, "What's in a name? a study of identifiers," in *14th IEEE International Conference on Program Comprehension (ICPC'06)*, 2006, pp. 3–12.

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

[20] A. Schankin, A. Berger, D. V. Holt, J. C. Hofmeister, T. Riedel, and M. Beigl, "Descriptive compound identifier names improve source code comprehension," in *Proceedings of the 26th Conference on Program Comprehension*, 2018, pp. 31–40.

[21] S. Fakhoury, Y. Ma, V. Arnaoudova, and O. Adesope, "The effect of poor source code lexicon and readability on developers' cognitive load," in *Proceedings of the 26th Conference on Program Comprehension*, 2018, pp. 286–296.

[22] T. R. Beelders and J.-P. L. du Plessis, "Syntax highlighting as an influencing factor when reading and comprehending source code," *Journal of Eye Movement Research*, vol. 9, no. 1, 2015.

[23] J. Feigenspan, C. Kästner, S. Apel, J. Liebig, M. Schulze, R. Dachselt, M. Papendieck, T. Leich, and G. Saake, "Do background colors improve program comprehension in the #ifdef hell?" *Empirical Software Engineering*, vol. 18, no. 4, pp. 699–745, 2013.

[24] 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.

[25] J. A. Dorn, "A general software readability model," 2012.

[26] D. Posnett, A. Hindle, and P. Devanbu, "A simpler model of software readability," in *Proceedings of the 8th Working Conference on Mining Software Repositories*, ser. MSR '11. New York, NY, USA: ACM, 2011, pp. 73–82. [Online]. Available: http://doi.acm.org/10.1145/1985441.1985454

[27] R. M. a. dos Santos and M. A. Gerosa, "Impacts of coding practices on readability," in *Proceedings of the 26th Conference on Program Comprehension*, 2018, pp. 277–285.

[28] E. Daka, J. Campos, G. Fraser, J. Dorn, and W. Weimer, "Modeling readability to improve unit tests," in *ESEC/SIGSOFT FSE*, 2015.

[29] D. Srinivasulu, A. Sridhar, and D. P. Mohapatra, "Evaluation of software understandability using rough sets," in *ICACNI*, 2013.

[30] J. Pantiuchina, M. Lanza, and G. Bavota, "Improving code: The (mis) perception of quality metrics," *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 80–91, 2018.

[31] M. E. Hansen, R. L. Goldstone, and A. Lumsdaine, "What makes code hard to understand?" *CoRR*, vol. abs/1304.5257, 2013.

[32] A. Wulff-Jensen, K. Ruder, E. Triantafyllou, and L. E. Bruni, "Gaze strategies can reveal the impact of source code features on the cognitive load of novice programmers," in *Advances in Neuroergonomics and Cognitive Engineering*, H. Ayaz and L. Mazur, Eds. Springer International Publishing, 2019, pp. 91–100.

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

[34] J. Borstler and B. Paech, "The role of method chains and comments in software readability and comprehension-an experiment," *IEEE Transactions on Software Engineering*, vol. 42, no. 9, pp. 886–898, 2016.

[35] T. Sedano, "Code readability testing, an empirical study," in *2016 IEEE 29th International Conference on Software Engineering Education and Training (CSEET)*, 2016, pp. 111–117.