UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Delano Hélio Oliveira

Understanding Code Understandability

Recife

2023

Delano Hélio Oliveira

Understanding Code Understandability

Trabalho apresentado ao Programa de Pós-graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco, como requisito parcial para obtenção do grau de Doutor em Ciência da Computação.

Área de Concentração: Engenharia de Software

Orientador (a): Fernando Jose Castor De Lima Filho

Coorientador (a): Fernanda Madeiral Delfim

Recife

2023

Catalogação na fonte
Bibliotecária: Mônica Uchôa, CRB4-1010

O48u Oliveira, Delano Hélio Oliveira.
Understanding Code Understandability / Delano Hélio Oliveira.– 2023.
193 f.: il., fig., tab.

Orientador: Fernando José Castor de Lima Filho.
Tese (Doutorado) – Universidade Federal de Pernambuco. Centro de Informática. Programa de Pós-graduação em Ciência da Computação, Recife, 2023.
Inclui referências.

1. Legibilidade de forma de código. 2. Legibilidade de conteúdo de código. 3. Compreensão de código. 4. Revisão de código. I. Lima Filho, Fernando José Castor de. II. Título.

310 CDD (23. ed.) UFPE - CCEN 2024 – 54

Delano Hélio Oliveira

“Understanding Code Understandability”

Tese de Doutorado apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Pernambuco, como requisito parcial para a obtenção do título de Doutor em Ciência da Computação. Área de Concentração: Engenharia de Software e Linguagens de Programação.

Aprovada em: 01/11/2023.

_____

Orientador: Prof. Dr. Fernando José Castor de Lima Filho

BANCA EXAMINADORA

_____

Prof. Dr. Sérgio Castelo Branco Soares
Centro de Informática/ UFPE

_____

Prof. Dr. Leopoldo Motta Teixeira
Centro de Informática/ UFPE

_____

Prof. Dr. Henrique Emanuel Mostaert Rebêlo
Centro de Informática/ UFPE

_____

Prof. Dr. Rodrigo Bonifacio de Almeida
Departamento de Ciência da Computação/UnB

_____

Prof. Dr. Marcelo de Almeida Maia
Faculdade de Computação / UFU

Dedico essa tese aos meus avós: José Pereira da Silva, Joana Rosendo Pereira, Enoque Leite de Oliveira, Iraci Simão de Oliveira. Não tiveram direito à educação, mas proporcionaram esse direito as suas futuras gerações.

ACKNOWLEDGEMENTS

Um trabalho como esse realizado durante bastante tempo requer o apoio e a contribuição de muita gente. Eu tentarei apresentar minha gratidão nesse pedaço de texto.

Eu sou muito grato ao Eterno que me permitiu estar ao redor de pessoas muito cuidadosas e não me deixou sentir sozinho. Dentre essas pessoas, a que mais serei eternamente grato é a senhora minha esposa, Mayara Gabriel. Ela passou por todas as dores e alegrias que esse doutorado me propiciou, lutou comigo contra o desânimo, a síndrome do impostor e tantas outros inimigos do doutorando. No final de todas essas lutas, eu, sozinho, recebo o mérito do trabalho. Ela merece todas as honras que eu puder dá pelo companheirismo sem hesitação e sem pedir nada em troca. Em falar em lutas, eu sou grato pela vida de Reydne Santos por persistir comigo nesse longo trabalho em todas as etapas. É uma honra muito grande ter Reydne como amigo e companheiro de pesquisa pois ele nunca deixou esse processo ser um lugar solitário. Fora que muito da qualidade das revisões sistemáticas desse trabalho foi por causa do zelo dele. Quando penso em qualidade de trabalho, eu preciso mencionar meus orientadores, Fernanda Madeiral e Fernando Castor. Eu tenho certeza que foi o Eterno que direcionou esse encontro, pois eu tive a melhor dupla de orientadores do mundo. Sou grato a ambos por me ensinar a ser competente em meu objeto de pesquisa, ético com a comunidade, zeloso com a metodologia e caprichoso com o documento final (Fernanda ia até as últimas comigo para entregar o melhor documento possível). Mas vocês foram além, cuidaram do orientando como ser humano, abraçaram minha família e me fizeram família em muitos momentos. Mais especificamente, obrigado Fernanda por sonhar comigo cada conquista que eu pude alcançar nesse processo (publicações, participações em eventos e doutorado sanduíche). Obrigado Castor por sempre trazer conforto nos momentos mais díficies mesmo sem perceber. Nossas reuniões eram sempre mais ricas em (des)informação contigo.

Eu também sou grato as pessoas que estiveram presente em algum momento desse processo, direto ou indiretamente. Eu agradeço ao professor Martin Monperrus pelas contribuições no meu trabalho e, claro, pelo acolhimento sem distinção em sua equipe durante o doutorado sanduíche. Nesse mesmo contexto, eu sou grato ao meu colega de sala durante o estágio, Thomas Durieux, pelo cuidado comigo e minha família. Ainda, sobre colegas de trabalho, agradeço a Fernando Alves e Fernando Oliveira pela oportunidade de trabalhar com vocês. Durante meu estágio pude fazer amigos que eu sou grato pelo conforto que trouxeram nos

momentos de frio e sem luz. Obrigado Gabriel e Susana Loubake, Carlos e Maria.

Ainda, sou muito grato pelos meus irmãos, Danilo, Samella, Letícia e Severino por me apoiarem nas dificuldades e celebrarem nas vitórias. Eu agradeço aos meus pais, Andrea e Cícero, pelo incentivo à educação desde sempre. Também agradeço aos meus sogros, Roberto e Kelly, meus (co)cunhados e relativos Kécia, Felipe, Renata, Mayra, Wanderson e Nilza, além de muitos primos, por cuidarem de mim e da minha família, principalmente nos momentos de minha ausência.

Eu também recebi muito apoio de amigos. Eu sou grato aos meus amigos da igreja Mangue que sempre me encheram de muita fé e esperança em momentos que era impossível encontrar uma saída. Obrigado Phablo, Ariel, Wesley, Victor, Dani, André, Maelyson, Bruna, Leo Marinho, Eduarda, Hemilly, Pedro, Tatiana, Daniel, Poli, Gabriel, Gabriela, Débora, Gabi, Thaís, Joana e amigos que não cabem nesse texto por motivos de limite. Agradeço aos amigos da vida, que indiretamente me incentivaram a realizar meus sonhos. Obrigado Thales, Renali, Júnior, Natan, Édipo, Natã, Izabela, Savyo, Demontiê e mais alguns que sabem que meu coração é grato.

Finalmente, agradecer ao IFPE Campus Palmares por apoioraem meu afastamento, que foi primordial para a qualidade desse trabalho. Sem sombras de dúvida, agradecer a Rafael, meu filho, que chegou para me lembrar o quanto eu sou amado, mesmo quando eu não consigo fazer isso por mim. Além disso, eu sou grato pela vida dos meus sobrinhos, Liz, Antônio, Ravi, Miguel e Manoel por alegrar meus dias.

“Eu não estou interessado em nenhuma teoria [...] amar e mudar as coisas me interessa mais” BELCHIOR (1).

ABSTRACT

Understanding source code is vital in software development, and developers spent between 58% and 70% of their time to code comprehension. This comprehension relies on code legibility and readability, influenced by factors like formatting, code constructs, and naming conventions. Formatting elements, such as spacing, are factors that impact the *legibility* of the source code and, consequently, may affect the ability of developers to identify the elements of the code while reading it. Structural and semantic characteristics, such as programming constructs, impact the *readability* of the source code and may affect the ability of developers to understand it while reading the code. This thesis explores code alternatives for improving code legibility and readability through empirical studies and practical evidence. We conducted four distinct studies. In the first one, we performed a systematic literature review to identify how code legibility and readability are evaluated, revealing categories of tasks and response variables. In the second one, we conducted another review, but focused on formatting elements, discovering 13 factors categorized into five groups that impact code legibility. A third review examined code elements, categorizing 20 factors into five groups affecting code readability. Finally, in the fourth study, we performed a survey of 2,401 code review comments where we found that over 42% aimed to improve code understandability. Also, we identified eight categories of code understandability smells where the 84.3% of improvement suggestions were accepted by developers. Based on these studies, we found limitations in the literature, which include a lack of replications, outdated studies, and insufficient power analyses. Despite these challenges, developers often adopt code alternatives suggested by research. Additionally, practical evidence highlights limitations of linters in code understandability where only 30% of the code understandability smells identified are flagged by linters. This thesis contributes a comprehensive catalog of code alternatives, aiding the creation of evidence-based coding guidelines and automated tools. We also presented a learning taxonomy adapted to program comprehension that simplifies research design in program comprehension. Moreover, we have available a dataset of code review comments and analysis of popular linters that serves as a valuable resource for researchers and developers, enabling the creation of automated tools to detect and repair code understandability smells.

Keywords: code legibility; code readability; program comprehension; code review

RESUMO

Compreender o código-fonte é vital no desenvolvimento de software, com os desenvolvedores gastando entre 58% e 70% de seu tempo na compreensão do código. Essa compreensão depende da legibilidade de forma e legibilidade de conteúdo do código, influenciada por fatores como formatação, construções de código e convenções de nomenclatura. Elementos de formatação, como espaçamento, são fatores que impactam a *legibilidade de forma* do código-fonte e, consequentemente, podem afetar a capacidade dos desenvolvedores de identificar os elementos do código durante a leitura. Características estruturais e semânticas, como construções de programação, impactam a *legibilidade de conteúdo* do código-fonte e podem afetar a capacidade dos desenvolvedores de entendê-lo durante a leitura do código. Esta tese explora alternativas de código para melhorar a legibilidade de forma e legibilidade de conteúdo do código através de estudos empíricos e evidências práticas. Nós realizamos quatro estudos distintos. No primeiro estudo, nós desenvolvemos uma revisão sistemática da literatura para identificar como são avaliadas a legibilidade de forma e a legibilidade de conteúdo do código, revelando categorias de tarefas e variáveis de resposta. No segundo estudo, nós executamos outra revisão, mas com foco nos elementos de formatação, descobrindo 13 fatores categorizados em cinco grupos que impactam a legibilidade de forma do código. Uma terceira revisão examinou os elementos do código, categorizando 20 fatores em cinco grupos que afetam a legibilidade de conteúdo do código. Por fim, no quarto estudo, realizamos um *survey* com 2.401 comentários de revisão de código, onde descobrimos que mais de 42% visavam melhorar a compreensão do código. Além disso, identificamos oito categorias de *understandability smells* onde 84,3% das sugestões de melhoria foram aceitas pelos desenvolvedores. Com base nesses estudos, encontramos limitações na literatura, que incluem falta de replicações, estudos desatualizados e potência estatística insuficientes. Apesar desses desafios, os desenvolvedores frequentemente adotam alternativas de código sugeridas pela pesquisa. Além disso, evidências práticas destacam as limitações dos *linters* na compreensibilidade do código, onde apenas 30% dos *code understandability smells* identificados são sinalizados por *linters*. Esta tese contribui com um catálogo abrangente de alternativas de código, auxiliando na criação de guia de estilos de codificação baseadas em evidências e ferramentas automatizadas. Também apresentamos uma taxonomia de aprendizagem adaptada à compreensão do programa que simplifica o projeto de pesquisas em compreensão de programas. Além disso, nós disponibilizamos um conjunto de dados de comentários de revisão de código e análise de *linters* populares que

serve como um recurso valioso para pesquisadores e desenvolvedores, permitindo a criação de ferramentas automatizadas para detectar e reparar *code understandability smells*.

Palavras-chave: legibilidade de forma de código; legibilidade de conteúdo de código; compreensão de código; revisão de código.

LIST OF FIGURES

LIST OF TABLES

CONTENTS

1 INTRODUCTION

Understanding source code is an important task in software development (11). For any kind of task that requires changing the code, such as fixing bugs, developing new features, and optimizing code, developers need to understand code. Previous studies (12, 13) found that developers spend between 58% and 70% of their time understanding code during software development. Source code that is difficult to read negatively affects programming activities (14). Factors such as documentation, developer experience, and source code structure are essential for understanding (14). The latter is associated with the legibility and readability of the source code. Formatting elements, such as spacing, are factors that impact the *legibility* of the source code and, consequently, may affect the ability of developers to identify the elements of the code while reading it. Structural and semantic characteristics, such as programming constructs, impact the *readability* of the source code and may affect the ability of developers to understand it while reading the code. We use the term *understandability* to refer to the ease with which developers are able to extract information from a program that is useful for software development- or maintenance-related tasks just by reading its source code. Understandability is impacted by legibility and readability.

Recent advancements in Large Language Models (LLMs) (15) have introduced a novel approach to software development. In this paradigm, developers employ LLMs to automatically generate code. However, their primary task shifts from traditional coding to code comprehension. Developers now need to read and evaluate the generated code to ensure it aligns with the intended functionality. This approach signifies a transformation in the developer's role, emphasizing code understanding and validation as a crucial aspect of the software development process.

Considering the importance of code understandability in the development process, we find efforts in the literature and the practice to establish the best code alternatives, i.e., different code snippets that are functionally equivalent, for code comprehension. Empirical studies, e.g., (16, 17, 18, 19, 20), have compared several different ways of writing code to find which factors impact code legibility and readability, e.g., levels of indentation (zero vs. two vs. four vs. six) (21, 22, 16, 23) and pre/pos-increment vs. separated operations (24, 25, 18) respectively.

In the practice of software development, some software organizations (e.g., Google[1] and

[1] <https://google.github.io/styleguide/>, last access Sept 28, 2023.

Sun Microsystems[2]) have adopted code conventions to enforce the usage of software development best practices by their developers. Smit et al.(26) explain that *"code conventions are a body of advice on lexical and syntactic aspects of code, aiming to standardize low-level code design under the assumption that such a systematic approach will make code easier to read, understand, and maintain"*. Furthermore, practitioners have adopted code review, which is a software development best practice, consisting of developers reading and commenting on code written by fellow developers in the same projects (27, 28, 29). In a typical code review process, code changes are submitted through pull-requests[3], and then read by developers who suggest improvements. Once the concerns are addressed, and the improvements are performed, the pull-request is approved for integration into the codebase. These suggestions might concern several code quality aspects, including understandability.

1.1 LEGIBILITY AND READABILITY

In software engineering, the terms readability, legibility, understandability, and comprehensibility have overlapping meanings. For example, Buse and Weimer (30) define "***readability*** *as a human judgment of how easy a text is to understand*". In a similar vein, Almeida et al. (31) affirm that "***legibility*** *is fundamental to code maintenance; if source code is written in a complex way, understanding it will require much more effort*". In addition, Lin and Wu (32) state that "*Software* ***understandability****" determines whether a system can be understood by other individuals easily, or whether artifacts of one system can be easily understood by other individuals*". Xia et al. (12) treat comprehension and understanding as synonyms, expressing that "*Program* ***comprehension*** *(aka., program understanding, or source code comprehension) is a process where developers actively acquire knowledge about a software system by exploring and searching software artifacts, and reading relevant source code and/or documentation*".

In linguistics, the concept of text comprehension is similar to program comprehension in software engineering. Gough and Tunmer (33) state that "*comprehension (not reading comprehension, but rather linguistic comprehension) is the process by which given lexical (i.e., word) information, sentences and discourses are interpreted*". However, Hoover and Gough (34) further elaborate on that definition and claim that "*decoding and linguistic comprehension are separate components of reading skill*". This claim highlights the existence of two separate

[2] <https://www.oracle.com/java/technologies/javase/codeconventions-contents.html>, last access Sept 28, 2023.

[3] In projects hosted on Github.

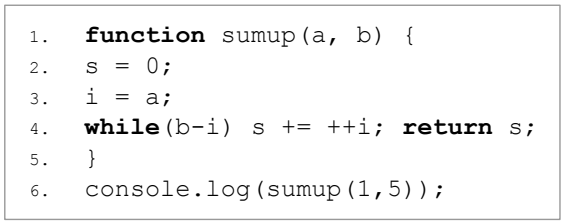
processes during text comprehension: (i) decoding the words/symbols and (ii) interpreting them and sentences formed by them. DuBay (35) separates these two processes and defines them as **legibility**, which concerns typeface, layout, and other aspects related to the identification of elements in text, and **readability**, that is, what makes some texts easier to read than others. In a similar vein, for Tekif (36), legibility studies are mainly concerned with typographic and layout factors while readability studies concentrate on the linguistic factors.

These two perspectives also apply to programs. We can find both the visual characteristics and linguistic factors in source code, although with inconsistent terminology. For example, Daka et al. (37) affirm that "*the visual appearance of code in general is referred to as its readability*". The authors clearly refer to legibility (in the design/linguistics sense) but employ the term "readability" possibly because it is more often used in the software engineering literature.

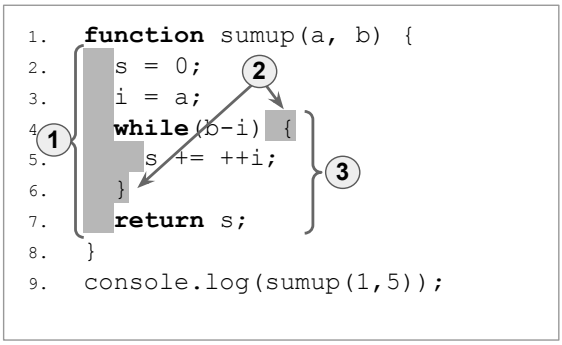
Based on the differences between the terms "readability" and "legibility" that are well-established in other areas such as linguistics (35), design (38), human-computer interaction (39), and education (36), we believe that the two terms should have clear, distinct, albeit related, meanings also in the area of software engineering. On the one hand, the structural and semantic characteristics of the source code of a program that affect the ability of developers to understand it while reading the code, e.g., programming constructs, coding idioms, and meaningful identifiers, impact its **readability**. On the other hand, the visual characteristics of the source code of a program, which affect the ability of developers to identify the elements of the code while reading it, such as line breaks, spacing, alignment, indentation, blank lines, identifier capitalization, impact its **legibility**.

To make these two terms easier to grasp, we illustrate the difference with an example. Consider the JavaScript code snippet in Figure 1a. It implements a function that, given two numbers, sums up all the intermediate numbers, the last one excluded. In the source code, when the function `sumup` is called with the arguments 1 and 5, it should return 10 (i.e., [1 to 5[$= 1 + 2 + 3 + 4 = 10$). However, the function returns 14, which means the source code has a bug. To make it easier to detect the bug, we can apply some improvements discussed in the literature.

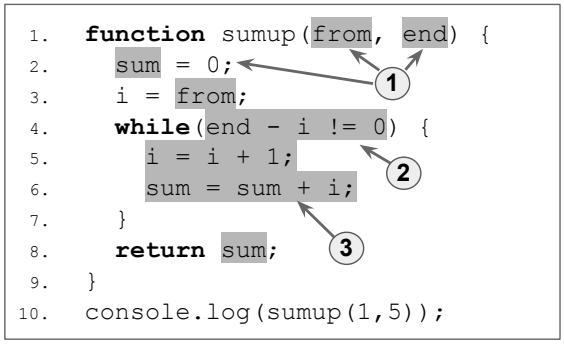
Several studies investigate formatting styles and perform changes in the source code as presented in Figure 1b. Miara et al. (21) found out that indentation has a statistically significant effect on program comprehension, which is why we indent the source code (Figure 1b–1). Gopstein et al. (25) and Medeiros et al. (18) found out that the presence of block delimiters



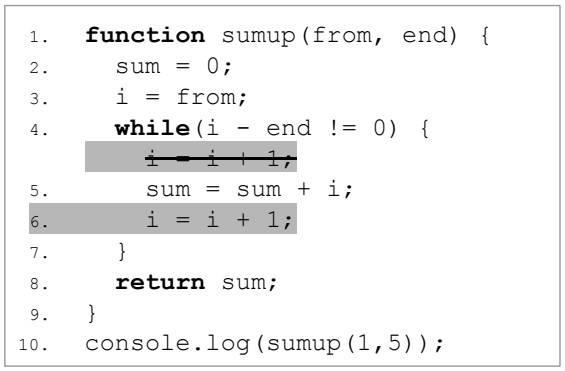
(a) Original source code.



(b) Changes on legibility-related code elements.



(c) Changes on readability-related code elements.



(d) Bug-free source code.

Figure 1 – Illustration of code element changes that influence legibility and readability.

makes the code better for program comprehension and, therefore, we added curly braces to the while loop in the lines 4 and six of the source code (Figure 1b–2). Medeiros et al. (18), Santos and Gerosa (16), and Sampaio and Barbosa (40) investigated one vs. multiple statements per line. Medeiros et al. and Sampaio and Barbosa found out that developers prefer one statement per line, which motivates us to introduce line breaks between statements (Figure 1b–3). Overall, Figure 1b illustrates that these formatting elements are what influences the ease of identifying elements of a program, which is what we define as **legibility**.

Still, existing literature contains studies that found code alternatives that better convey the meaning, that is, the semantics of the code elements. As shown in Figure 1c, the best alternative for code readability based on these studies has been used to present an improved version of Figure 1b. Hofmeister et al. (41) found out that abbreviations and letters reduce a program's comprehensibility and, therefore, we use full words to improve the code readability (Figure 1c–1). Gopstein et al. (25) found out that implicit predicate, i.e., the inference of numerical values as boolean values in our example, causes confusion in code comprehension. Consequently, we replace the implicit expression with an explicit boolean expression (Figure 1c–2). Dolado et al. and Gopstein et al. (25) found out that expressions with side-effects are harmful to code readability and, hence, we replace the pre-increment with its alternative

without side-effects (Figure 1c–3).

We improved the legibility and readability code of the example shown in Figure 1a based on the findings of empirical studies. In Figure 1d, we could identify the line of code that was causing the bug in the method, and then we fixed the order of statements within the block of the `while` construct. In this thesis, we use the term *"understandability"* to refer to both legibility and readability.

1.2 PROBLEM STATEMENT

Understanding a program usually requires reading code. A program might be easier or harder to read depending on its *readability* and *legibility*. Different factors can influence code readability and legibility, such as which constructs are employed (25, 42), how the code is formatted with whitespaces (43, 44, 23), and identifier naming conventions (45, 46, 47, 48, 49, 50, 51, 52, 53, 41). It is worth to mention that human factors also affect the code readability and legibility, such as programmer experience. Researchers have conducted empirical studies to investigate several of those factors, where different but functionally equivalent ways of writing code are compared, e.g., recursive vs. iterative code (54) and abbreviated vs. word identifier names (41). These studies involve asking subjects to perform one or more tasks related to source code and assessing their understanding or the effort involved in the tasks. The different methods used to assess code comprehension consider various cognitive skills and can lead to different results. For example, in a comparison between ternary/conditional expressions (e.g., `a?b:c`) and `if` statements, Gopstein et al. (25) evaluated the accuracy and time it took for subjects to determine the output of code snippets and found that the conditional operator was more confusing than the `if` statement. In a more recent study, Medeiros et al. (18) asked for the subjects' opinions and found that the use of the conditional operator was neither negative nor positive compared to `if` statements.

To the best of our knowledge, however, **no previous work has investigated how code readability and legibility are evaluated in studies comparing different ways of writing code, in particular, what kinds of tasks these studies conduct and what response variables they employ (problem #1)**. Although there are systematic literature reviews about program comprehension studies (55, 11, 14), they have the focus in other aspects, such as confounding parameters (55) or the aspect of software development (14). Analyzing the tasks and variables of the primary studies can help researchers identify limitations in the

evaluations of previous studies, gauge their comprehensiveness, and improve our understanding of what said tasks and variables evaluate.

Moreover, there exist guidelines and coding standards for different programming languages, such as the Google Java Style Guide[4] for the Java language, which describe good practices and well-accepted conventions on how to write code. However, these guides are built based on the intuition and experience of the developers that elaborate them. In fact, **what makes the code more legible or readable is an open question (problem #2)**. Researchers have conducted empirical studies to compare different but functionally equivalent ways of writing code in terms of their legibility and readability. For instance, Miara et al.(21) explored different levels of indentation (ranging from zero to six spaces), while Dolado et al.(24) delved into the usage of pre/post-increment versus separated operations. However, coding style guidelines seem to be inconsistent with the findings of these studies. For instance, the results obtained by Gopstein et al.(25) suggest that omitting braces may be bug-prone. In contrast, The Linux Kernel Coding Style[5] for C language, argues in favor of block delimiters in some cases, such as when the block consists of only one line, to make the code less verbose. These studies provide insight into the influence of different code alternatives on code legibility and readability, but they are parts of a bigger whole that is still unclear.

Finally, those laboratory studies on code understandability have as their main advantage the possibility to account for and control extraneous factors and objectively establishing relationships between independent and response variables. However, they are not able to account for elements such as project culture and guidelines, and developers' background, which directly impact how developers understand code. Not accounting for the influence of these factors may lead to results that are sound but have little external validity. **There is a lack of knowledge on what developers care about regarding code understandability and how they improve it, in a real-world setting (problem #3)**.

1.3 GOALS AND METHODS

The primary objective of this thesis is **to investigate alternative ways of writing code that improve its legibility and readability based on empirical studies and practical evidence**. Our long-term goal is to develop a coding style guide that is totally evidence based.

[4] <https://google.github.io/styleguide/javaguide.html>, last access May 22, 2023.
[5] <https://www.kernel.org/doc/html/v4.10/process/coding-style.html>, last access May 22, 2023.

In the context of this thesis, we have three goals to address each problem presented in the previous section.

The first goal is **to study what types of tasks are performed by human subjects in those laboratory studies, what cognitive skills are required from them, and what response variables those studies employ (goal #1)**. To achieve this goal, we conducted a systematic literature review, identifying studies of a human-centric nature that compared various code alternatives. These studies involved human subjects in tasks aimed at assessing their performance or gauging task difficulty.

The second goal is **to provide a comprehensive view of the existing knowledge within the literature about the impact of different code alternatives on code legibility and readability (goal #2)**. To do so, we examined empirical studies performed by researchers with human subjects aiming to find which code alternatives are the best for the legibility and readability of source code. We performed two systematic literature reviews, one focused on legibility and the other on readability. These reviews selected and evaluated papers that directly compared different code alternative levels and organized them through a card-sorting process.

The third goal is **to investigate how developers improve code understandability during software development regarding what code alternatives are the best for code understandability (goal #3)**. To accomplish this, we examined code understandability improvements grounded on real comments about code quality made during code review. More specifically, we investigated what code understandability issues are reported by developers during code review, and what changes developers apply to address those issues. Our key insight is that this knowledge is available through publicly-available code reviews in open-source software, where code reviewers act as project specialists in code quality.

1.4 CONTRIBUTIONS

This thesis comprises four studies. Two studies were published, one study is under review, and one study is ongoing. Figure 2 shows the title and the main question of each study.

In STUDY 0, we conducted a systematic review that selected 54 human-centric research papers that evaluated code legibility and readability. Our study identified the tasks and response variables used in these papers, and we also adapted an existing learning taxonomy to the context of program comprehension. The findings of this study improves our understanding of existing studies and can help researchers design new ones. This study contributes to goal #1.

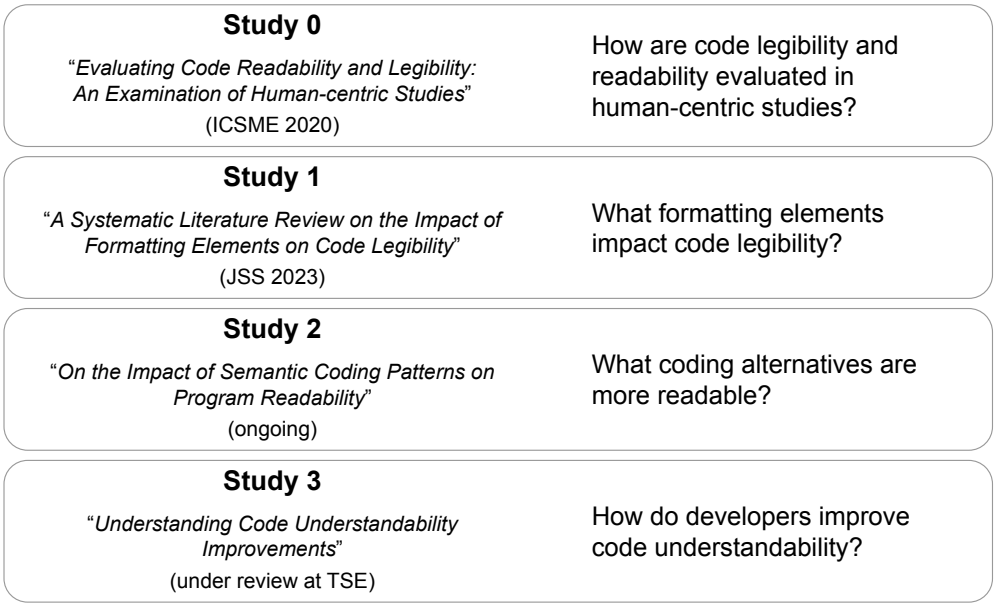


Figure 2 – Contributions of this thesis.

We also conducted systematic literature reviews in both STUDY 1 and STUDY 2 that selected 15 and 40 papers, respectively. These papers directly compared code alternatives in terms of legibility (STUDY 1) and readability (STUDY 2). We then identified and evaluated the different levels of code elements being compared and two aspects of human-centric studies that evaluate code legibility and readability: the activities performed by human subjects (which relate to the cognitive skills required from them) and the response variables employed by the researchers to collect data from the studies. This analysis was based on the outcomes of STUDY 0. The findings of these studies contribute to the creation of a body of knowledge toward the creation of guidelines and automated aids, e.g., linters and recommendation systems, to help developers during programming activities and make their code more legible and readable. These studies contribute to our goal #2.

In STUDY 3, we conducted a survey on comments written by developers during code review in open-source repositories. The goal was to identify suggestions for improving code understandability. We manually analyzed 2,401 inline code review comments and found 1,012 that were related to code understandability. We analyzed these comments and the corresponding remediation changes that the developers made to improve code understandability. We also evaluated the changes' prevalence and whether linters could warn about the need to improve the source code. This study provides practical guidance for creating tools that make code more understandable. For example, some improvements that could be automated are currently not available in any of the four popular linters considered in the study. Additionally, under-

standability improvements are rarely reversed, making them a reliable source of training data for specialized machine learning-based tools. Our dataset is available as a contribution of this study and can serve as a starting point to train such models. The findings of this study can also be used to develop evidence-based code style guides. This study contributes to our goal #3.

Data availability: The data produced in each study are available at <https://github.com/delanohelio/understanding-code-understandability>.

1.5 THESIS ORGANIZATION

The remainder of this document is organized as follows:

- Chapter 2 presents STUDY 0, which investigates what tasks are performed by human subjects and how their performance is evaluated in empirical studies aiming to evaluate legibility and readability. Additionally, this study shows program comprehension as a learning activity to identify and name the cognitive skills required by different tasks employed by code legibility and readability studies. This study was published at ICSME 2020 (56).
- Chapter 3 presents STUDY 1, which investigates the outcomes of empirical studies that evaluated which formatting elements are more legible than equivalent ones. This study was published at JSS 2023 (57).
- Chapter 4 presents STUDY 2, which investigates the outcomes of empirical studies that evaluated which programming constructs, coding idioms, and semantic cues are more readable than functionally equivalent ones. This study is ongoing.
- Chapter 5 presents STUDY 3, which investigates how developers improve code understandability during software development through code review comments. This investigation identifies types of patches that improve code understandability and evaluates the ability of four well-known linters to flag the identified code understandability issues. This study is under review at TSE.
- Chapter 6 presents the final considerations of this thesis and avenues for future work.

2 EVALUATING CODE READABILITY AND LEGIBILITY: AN EXAMINATION OF HUMAN-CENTRIC STUDIES

In this chapter, we investigate the different tasks and variables involved in studies that compare programming constructs, coding idioms, naming conventions, and formatting guidelines. To achieve this, we conducted a systematic literature review and found 54 relevant papers. Additionally, we have modeled program comprehension as a learning activity by adapting an existing learning taxonomy. This work aim to study what types of tasks are performed by human subjects in those laboratory studies, what cognitive skills are required from them, and what response variables those studies employ (goal #1). This study was published at ICSME 2020 (56).

2.1 METHODOLOGY

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

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

RQ2 What are the response variables of these studies?

To answer our research questions, we conducted a systematic literature review, which was designed following the guidelines proposed by Kitchenham et al. (58). Figure 3 presents the roadmap of our review including all steps we followed. First, we performed the selection of studies. We started with a manual search for papers to further be used as seed papers, so that a search string could be defined, and automatic search could be performed on search engines (Section 2.1.1). We retrieved 2,843 documents with the automatic search, which passed through 1) a triage for study exclusion (Section 2.1.2), 2) an initial study selection where inclusion criteria were applied (Section 2.1.3), and 3) a final study selection where

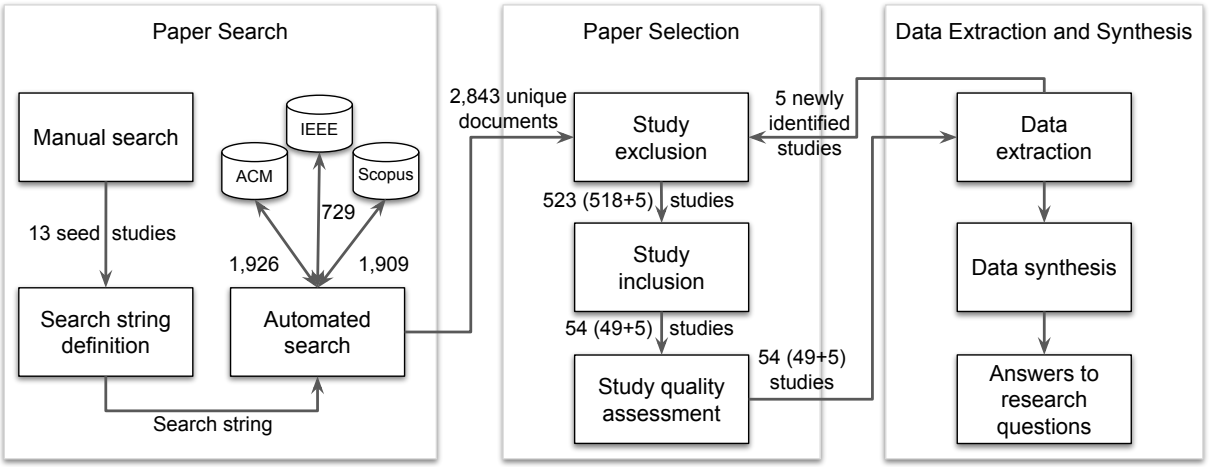


Figure 3 – Systematic literature review roadmap of the study about task and response variables in studies related to code legibility and readability.

we evaluated the quality of the studies based on a number of criteria (Section 2.1.4). Then, the selected 54 papers were analyzed (Section 2.1.5). We extracted data from the papers and synthesized it to answer our research questions. We detail these steps in the following sections. It is worth mentioning that we did not leverage systematic review tools, like Parsifal (59), because we were not aware of them at the time.

2.1.1 Search Strategy

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

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

$$Title(ANY(terms))\ OR\ Keywords(ANY(terms)),$$

where $terms = \{$ "code comprehension", "code understandability",
"code understanding", "code readability",
"program comprehension", "program understandability",
"program understanding", "program readability",
"programmer experience" $\}$

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

Finally, we performed an automatic search for studies using our generic search string adapted for three search engines: ACM Digital Library(60), IEEE Explore(61), and Scopus(62). We retrieved 1,926, 729, and 1,909 documents, respectively, in September 2, 2019. Since a given document might be retrieved from more than one engine, we unified the output of the engines to eliminate duplicates, which resulted in 2,843 unique documents. The 13 seed papers were returned by the automatic search.

2.1.2 Triage (Study Exclusion)

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

- EC1: The study is outside the scope of this study. It is not primarily related to source code comprehension, readability, or legibility, does not involve any comparison of different ways of writing code, neither direct nor indirect, or is clearly irrelevant to our research questions. For instance, we exclude studies focusing on high-level design, documentation, and dependencies (higher-level issues).
- EC2: The study is not a full paper (e.g., MSc dissertations, PhD theses, course completion monographs, short papers) or is not written in English: not considering these types of

documents in systematic reviews is a common practice (58). As a rule of thumb, we consider that full papers must be at least 5 pages long.

- EC3: The study is about readability metrics without an experimental evaluation.
- EC4: The study is about program comprehension aids, such as visualizations or other forms of analysis or sensory aids (e.g., graphs, trace-based execution, code summarization, specification mining, reverse engineering).
- EC5: The study focuses on accessibility, e.g., targets individuals with visual impairments or neurodiverse developers.

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

2.1.3 Initial Selection (Study Inclusion)

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

- IC1 (Scope): The study must be primarily related to the topics of code comprehension, readability, legibility, or hard-to-understand code.
- IC2 (Methodology): The study must be or contain at least one empirical study, such as controlled experiment, quasi-experiment, case study, or survey *involving human subjects*.
- IC3 (Comparison): The study must compare alternative programming constructs, coding idioms, or coding styles *in terms of code readability or legibility*.

- IC4 (Granularity): The study must target fine-grained program elements and low-level/limited-scope programming activities. Not design or comments, but implementation.

The application of the inclusion criteria to a paper often requires reading not only the title and abstract as in the triage step, but also sections of introduction, methodology, and conclusion. If a given paper violates at least one inclusion criterion, the paper is annotated with "not acceptable". When there are doubts about the inclusion of a paper, the paper is annotated with "maybe" for further discussion. We also performed this step in two rounds, but differently from the triage, all papers in this step were independently analyzed by two different authors. We calculated the Kappa coefficient (63) for assessing the agreement between the two analyzes. We found $k = 0.323$, which is considered a fair agreement strength (58). In the end of this step, the papers annotated with "acceptable" in both rounds were directly selected, and papers annotated with "not acceptable" in both rounds were rejected. All the other cases were discussed by the four authors in live sessions to reach consensus. We ended up with 54 papers.

2.1.4 Study Quality Assessment

After the search, exclusion, and inclusion of papers, the remaining 54 papers passed through a final selection step, aiming to assess their quality. In this step, we aim to identify low-quality papers for removal. To do so, we elaborated nine questions that were answered for each paper. We adapted these questions from the work of Keele (64). There were three questions about study design, e.g., "are the aims clearly stated?"; four questions about analysis, e.g., "are the data collection methods adequately described?"; and two questions about rigor, e.g., "do the researchers explain the threats to the study validity?". There are three possible answers for each question: yes (1), partially (0.5), and no (0). The sum of the answers for a given paper is its score. The maximum is, therefore, 9. If a paper scores 0.5 or better in all questions, its overall score is 4.5 or more. Thus, we defined that a paper should score at least 4.5 to be kept in our list of papers.

Each paper was assessed by one of the authors. At the beginning of this step, each author selected one paper, performed the quality assessment, and justified to the other authors the given score for each question in a live discussion. This procedure allows us to align our understanding of the questions and avoid misleading assessments. The scores of the studies were:

$min = 5$, $median = 8$, $max = 9$. Since the minimum score for keeping a paper is 4.5 and no paper scored less than 5, no studies were removed because of bad quality.

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

Deprecated studies. We identified some papers that we refer to as *deprecated*. A paper is deprecated if it was extended by another paper that we selected. For instance, the work of Buse and Weimer published in 2008 (65) was extended in a subsequent paper (30). In this case, we consider the former to be deprecated and only take the latter into account.

2.1.5 Data Analysis

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

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

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

2.2 RESULTS

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

2.2.1 Tasks Performed by Human Subjects (RQ1)

The essential code comprehension task is code reading. By construction, all the selected studies have at least one reading task where the subject is required to read a code snippet, a set of snippets, or even large, complete programs. Since all the studies compare two or more ways of writing code, the subjects often have multiple code reading tasks. In addition, the subjects are also expected to comprehend the code. However, there are different ways to measure subject comprehension performance. Subjects are asked to provide information about the code, act on the code, or provide personal opinion. In most studies, more than one kind of task was employed. Table 1 summarizes the identified tasks.

A large portion of the primary studies, 40 out of 54, required subjects to **provide information about the code**. For example, Benander et al.(54) asked the subjects to explain using free-form text what a code snippet does, right after having read it. Blinman et al.(46) asked the subjects to choose the best description for a code snippet among multiple options. Explaining what the code does is not an objective method to evaluate comprehension because someone has to judge the answer. We found 18 studies that employed this task.

The subjects of 27 studies were asked to answer questions about characteristics of the code. In some studies, the subjects were asked to predict the behavior of a source code just by looking at it. For example, Gopstein et al.(25) and Ajami and Feitelson (42) presented subjects with multiple code snippets and asked them to guess the outputs of these snippets. Dolado et

Table 1 – Task types and their corresponding studies. A study may involve more than one type of task.

Task type	Studies
provide information about the code (40 studies)	
explain what the code does	18 studies: (51, 54, 52, 66, 67, 68, 17, 69, 70, 71, 47, 72, 21, 73, 74, 75, 46, 76)
answer questions about code characteristics	27 studies: (42, 23, 67, 24, 77, 25, 78, 71, 79, 76, 80, 81, 82, 45, 83, 19, 84, 75, 85, 48, 66, 86, 21, 87, 49, 88, 89)
remember (part of) the code	7 studies: (48, 66, 47, 72, 73, 74, 68)
act on the code (15 studies)	
find and fix bugs in the code	10 studies: (52, 69, 90, 91, 50, 53, 17, 41, 80, 81)
modify the code	8 studies: (52, 90, 49, 77, 91, 69, 80, 84)
write code	3 studies: (92, 19, 84)
provide personal opinion (30 studies)	
opinion about the code (readability or legibility)	23 studies: (10, 30, 69, 70, 88, 89, 76, 93, 94, 95, 92, 43, 23, 51, 52, 16, 18, 19, 84, 49, 91, 82, 44)
answer if understood the code	4 studies: (76, 86, 82, 83)
rate confidence in her answer	3 studies: (47, 74, 85)
rate the task difficulty	7 studies: (17, 21, 23, 49, 69, 70, 85)

al.(24) asked the subjects to answer a set of questions about expression results, final values of variables, and how many times loops were executed. In other studies, the subjects were asked questions about higher-level issues related to source code. For example, Scalabrino et al.(86) asked the subjects if they recognize an element of a specific domain or about the purpose of using an external component in the snippet (e.g., JDBC APIs). Similarly, Binkley et al.(48) inquired the subjects about the kind of application or the industry where a specific line of code might be found. In addition, some studies required subjects to localize code elements of interest. For example, Binkley et al.(66) asked subjects to find identifiers in a snippet, marking each code line that has that identifier. Ceccato et al.(49) asked the subjects to pinpoint the part of the code implementing a specific functionality.

Some studies made the assumption that code that is easy to understand is also easy to memorize. Therefore, they attempted to measure how much subjects remember the code. For example, Love (72) asked subjects to memorize a program for three minutes and rewrite the program as accurately as possible in the next four minutes. Lawrie et al. (47) first presented

a code snippet to the subjects. Then, in a second step, they listed six possible identifiers and the subjects had to select the ones that they recalled appearing in the code. Overall, seven studies asked the subjects to remember the code.

On the other hand, some studies required the subjects to **act on the code**. In ten studies subjects were asked to find and fix bugs in the code. Scanniello et al. (50) asked subjects to do so in two programs with different identifier styles. In eight other studies the subjects were asked to modify the code of a working program, i.e., without the need to fix bugs. For example, Jbara and Feitelson(69) asked subjects to implement a new feature in a program seen in a previous task. Likewise, Schulze et al.(80) requested that subjects modify and delete annotated code with specific preprocessor directives, which also requires understanding the respective source code.

In a few studies, subjects were asked to write code from a description. Writing code *per se* is not a comprehension task, but it may be associated to a comprehension task. For example, Wiese et al.(19) first asked subjects to write a function that returns true if the input is 7 and false otherwise so that they could identify what code style (novice, expert, or mixed) the subjects preferred when coding. Afterwards, they asked the subjects to choose the most readable among three versions of a function, each one with a different code style. One of the goals of this study was to determine if the subjects write code in the same style that they find more readable.

Lastly, in 30 studies subjects were asked to give their **personal opinion**. In nine studies the subjects were inquired about their personal preferences or gut feeling without any additional task. For example, Buse et al.(30) asked them to rate (from 1 to 5) how legible or readable a code snippet is. Similarly, Arab(43) asked subjects to classify the legibility of three presentation schemes of Pascal code in descending order. In the study of Santos and Gerosa (16), the subjects chose the most readable between two functionally equivalent snippets. In other studies, 21 in total, the subjects were asked about their personal opinion while they performed other tasks. For example, O'Neal et al.(76) first asked subjects to read a code snippet, and then to state if they understood the snippet and to provide a description of its functionality. Similarly, Lawrie et al.(47) asked subjects to provide a free-form written description of the purpose of a function and to rate their confidence in their description. In addition, subjects were asked to rate the difficulty of the comprehension tasks they had to perform in some studies, e.g., in the study of Fakhoury et al.(17).

Key takeaways for RQ1. To assess code readability and legibility, researchers conduct studies where subjects are asked to provide information about the code, act on the code, or give their personal opinion. We found out that 16.7% of the studies only ask the subjects to provide personal opinion. Moreover, although most of the analyzed studies require subjects to provide information about source code, that information varies widely in nature. Subjects may be asked to predict the output of programs, identify code elements, or explain high-level functionality.

2.2.2 Response Variables (RQ2)

Depending on the goals, methodology, and subjects of a study, response variables vary. This section presents the results obtained in the analysis of the response variables used in the selected studies. Since there is considerable diversity of response variables, we have organized them into five categories. Figure 4 presents the frequency of all response variables. The numbers do not add up to 54, which is the overall number of analyzed studies, because most studies employed more than one response variable. In Table 2, we present a detailed synthesis of the identified response variables.

The performance of subjects was measured in some studies in terms of whether they were able to correctly provide information about programs just by looking at the source code. The response variables may pertain to code structure, semantics, use of algorithms, or program behavior, for instance. We aggregated response variables like these into a category called **correctness**. On the one hand, correctness can be objectively determined. For example, Bauer et al. (23) measured the subjects' code understanding by asking them to fill in a questionnaire with multiple-choice questions referring to program output. Gopstein et al. (25) and Ajami and Feitelson (42) asked the subjects to predict the outputs of the execution of short programs. On the other hand, correctness was subjectively determined in some studies, where there was some margin to interpret if the results produced by a subject were correct. For example, Blinman et al. (46) evaluated if the subjects' textual description of a program was correct. Similarly, Love (72) asked subjects to write a textual description of a program and scored it based on a subjective evaluation. Overall, correctness response variables were employed by 83.3% of the studies.

The second most often employed response variable, present in 30 studies, is the subjects'

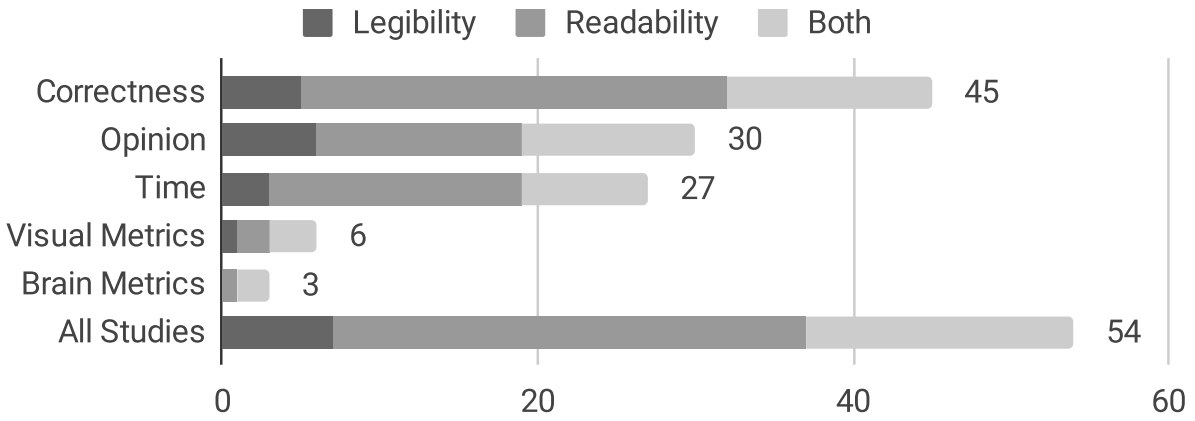


Figure 4 – Frequency of response variables.

personal opinion. What is common to all these studies is the use of the preferences and gut feeling of the subjects, instead of the results of what they do, to assess readability and legibility. We grouped these response variables in a category called **opinion**. Scalabrino et al. (86), for example, asked the subjects to state whether they understood a code snippet or not. Santos and Gerosa (16) presented pairs of functionally equivalent snippets to the subjects and asked them to choose which one they think is more readable or legible. Stefik and Gellenbeck (95) requested that subjects rate lists of words/symbols associated to programming constructs (e.g., conditionals and loops) based on how intuitive they think they are. Lawrie et al. (47) asked the subjects to rate their confidence in their understanding of the code.

The **time** that subjects spent to perform tasks was measured in multiple studies. This response variable category is the third most often used in the analyzed studies, with 27 instances. There is variety in the way the studies measured time. For Ajami and Feitelson (42), *"time is measured from displaying the code until the subject presses the button to indicate he is done"*. Hofmeister et al. (41) computed the time subjects spent looking at specific parts of a program. Geffen et al. (77) measured the response time for each question when subjects answer a multiple-choice questionnaire about the code. Instead of directly measuring time, Malaquias et al. (91) counted the attempts to fix syntactic and semantic errors in the code.

In some studies, information about the process of the task was collected instead of its outcomes. In particular, multiple studies employed special equipment to track what the subjects see, and employed **visual metrics** as response variables. Six of the analyzed studies employed some form of eye tracking. For example, Blinkley et al. (66) computed the visual attention, measured as the amount of time during which a subject is looking at a particular area of the screen. Bauer et al. (23) computed three visual metrics: fixation duration, fixation rate, i.e., the number of fixations per second, and saccadic amplitude, i.e., the spatial length of a saccade,

Table 2 – Response variables and their corresponding studies.

Category and type	Sub-type	Studies
Correctness (45 studies)		
Objective	Binary	37 studies: (42, 23, 48, 66, 46, 67, 49, 24, 17, 77, 25, 41, 90, 78, 69, 47, 72, 79, 91, 21, 88, 89, 76, 86, 50, 53, 81, 82, 45, 87, 83, 73, 75, 74, 19, 84, 85)
	Scale	3 studies: (80, 92, 52)
Subjective	Binary	11 studies: (51, 54, 67, 68, 17, 71, 21, 73, 74, 19, 84)
	Rate	7 studies: (52, 70, 47, 66, 69, 72, 75)
Opinion (30 studies)		
Personal preference	Rate	9 studies: (10, 30, 18, 88, 89, 93, 94, 95, 92)
	Choice	6 studies: (51, 52, 16, 44, 19, 84)
	Ranking	1 study: (43)
On understandability	Binary	2 studies: (86, 83)
	Rate	2 studies: (82, 76)
Professional opinion	Acceptability	2 studies: (18, 91)
Rate answer confidence	Rate	3 studies: (47, 74, 85)
Rate task difficulty	Ranking	1 study: (23)
	Rate	6 studies: (49, 17, 70, 69, 21, 85)
Time (27 studies)		
Time to complete task		20 studies: (42, 23, 54, 52, 46, 49, 24, 41, 78, 69, 70, 90, 91, 76, 50, 53, 89, 88, 80, 66)
Time reading code		5 studies: (51, 66, 67, 17, 86)
Time per question		4 studies: (48, 77, 67, 79)
Number of attempts		2 studies: (91, 79)
Visual Metrics (6 studies)		
Eye tracking		4 studies: (23, 66, 70, 17)
Letterboxing		2 studies: (41, 53)
Brain Metrics (3 studies)		
fMRI		1 study: (81)
fNIRS		1 study: (17)
EEG		1 study: (85)

that is, the transition between two fixations. Hofmeister et al. (41) employed a software tool that limits the subjects' view of the code to a few lines at a time. This frame can be shifted up and down using the arrow keys to reveal different parts of the code. This approach is called "letterboxing". Hofmeister et al.(41) called each code frame an area of interest (AOI), and measured the time subjects spent on an AOI, first-pass reading times, and AOI visits.

Recently, some researchers have resorted to leveraging brain monitoring tools to understand

what happens in the brain of the subjects during program comprehension. In total, three of the analyzed studies employed response variables based on brain monitoring. Siegmund et al. (81) used functional magnetic resonance imaging (fMRI) to measure brain activity by detecting changes associated with blood flow. Fakhoury et al. (17) employed functional near-infrared spectroscopy (fNIRS) to measure brain activity through the hemodynamic response within physical structures of the brain. Yeh et al. (85) leveraged electroencephalography (EEG) to monitor the electrical activity of the brain. These response variables were grouped into the **brain metrics** category.

The analyzed studies differ in the response variables they employed, depending on whether they aimed to assess readability, legibility, or both. As shown in Figure 4, readability studies leveraged all the response variable categories whereas no legibility study leveraged brain metrics. This is not surprising, considering the much lower number of legibility studies. In addition, a high proportion of legibility studies (86%) employed opinion response variables. This is not the case for studies only about readability (43%) or about both (65%).

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

Key takeaways for RQ2. There are five categories of response variables. Correctness is the most widely used, being employed in 83.3% of the analyzed studies. Time and opinion are also often employed (50% and 55.6% of the studies, respectively). A significant number of studies used the variable time and/or opinion associated with correctness. On the other hand, 30% of the studies employed a single response variable. The readability and readability+legibility studies used all the response variable categories while almost all the legibility studies used the opinion response variable.

2.2.3 Program Comprehension as a Learning Activity

The empirical studies analyzed in this work involve a wide range of tasks to be performed by their subjects (see Section 2.2.1). For example, they may ask subjects to memorize a program, follow its execution step by step, answer questions about it, or write a high-level explanation of what it does. All these tasks are conducted with the goal of evaluating readability and legibility. However, they demand different cognitive skills from the subjects and, as a consequence, evaluate different aspects of readability and legibility. We attempt to shed a light on this topic by analyzing the cognitive skill requirements associated with each kind of task.

According to the Merriam-Webster Thesaurus, to learn something is *"go gain an understanding of"* it. We follow this definition by treating the problem of program comprehension (or understanding) as a learning problem. In this section we propose an adaptation of the learning taxonomy devised by Fuller and colleagues (9) to the context of program comprehension. This taxonomy is itself an adaptation of Bloom's taxonomy of educational objectives (96) to the context of Software Development. Fuller et al. (9) state that *"learning taxonomies [..] describe the learning stages at which a learner is operating for a certain topic."* A learning taxonomy supports educators in establishing intended learning outcomes for courses and evaluate the students' success in meeting these goals. According to Biggs (97) and Fuller et al. (9), learning taxonomies can help with "understanding about understanding" and "communicating about understanding". Based on this description, they seem to be a good fit to help researchers better understand studies about program understanding and communicate about them. Our central idea is to use the elements defined by the taxonomy of Fuller et al. (9), with some adaptions, to identify and name the cognitive skills required by different tasks employed by code readability and legibility studies.

2.2.3.1 A Learning Taxonomy

Bloom's taxonomy (96) consists of three hierarchical models aiming to classify educational learning objectives in terms of complexity and specificity. In the context of this work, the cognitive domain of this taxonomy is the most relevant. It recognizes that learning is a multifaceted process that requires multiple skills with different levels of complexity and which build upon each other. This taxonomy has been later revised by Anderson et al. (98). The most visible aspect of the revised taxonomy is a list of six activities that define progressively more

Table 3 – Learning activities extended from Fuller et al. (9)—Inspect and Memorize are not in the original taxonomy. Opinion is not included because it is not directly related to learning.

Activity	Description	Example
Adapt	Modify a solution for other domains/ranges. This activity is about the modification of a solution to fit in a given context.	Remove preprocessor directives to reduce variability in a family of systems (80).
Analyze	Probe the [time] complexity of a solution.	Identify the function where the program spends more time running.
Apply	Use a solution as a component in a larger problem. Apply is about changing a context so that an existing solution fits in it.	Reuse of off-the-shelf components.
Debug	Both detect and correct flaws in a design.	Given a program, identify faults and fix them (50).
Design	Devise a solution structure. The input to this activity is a problem specification.	Given a problem specification, devise a solution satisfying that specification.
Implement	Put into lowest level, as in coding a solution, given a completed design.	Write code using the given examples according to a specification (92).
Model	Illustrate or create an abstraction of a solution. The input is a design.	Given a solution, construct a UML model representing it.
Present	Explain a solution to others.	Read a program and then write a description of what it does and how (68).
Recognize	Base knowledge, vocabulary of the domain. In this activity, the subject must identify a concept or code structure obtained before the task to be performed.	"The sorting algorithm (lines 46-62) can best be described as: (A) bubble sort (B) selection sort (C) heap sort (D) string sort (E) partition exchange. sort" (87).
Refactor	Redesign a solution (as for optimization). The goal is to modify non-functional properties of a program or, at a larger scale, reengineer it.	Rewrite a function so as to avoid using conditional expressions.
Relate	Understand a solution in context of others. This activity is about identifying distinctions and similarities, pros and cons of different solutions.	Choose one out of three high-level descriptions that best describe the function of a previously studied application (46).
Trace	Desk-check a solution. Simulate program execution while looking at its code.	Consider the fragment "x=++y": what is the final value of x if y is -10? (24).
Inspect*	Examine code to find or understand fine-grain static elements. Inspect is similar to Analyze, but it happens at compile time instead of run time.	"All variables in this program are global." [true/false] (21).
Memorize*	Memorize the code in order to reconstruct it later, partially or as a whole.	Given a program, memorize it in 3 minutes and then reconstruct it in 4 (72).

sophisticated levels of learning: remember, understand, apply, analyze, evaluate, and create.

Fuller et al. (9) proposed an adaptation of the revised version of Bloom's taxonomy for the area of Computer Science, with a particular emphasis on software development. They developed a set of activities that build upon Bloom's revised taxonomy and organized them in a model that emphasizes that some activities in software development involve acting on knowledge, instead of just learning. We leverage this taxonomy and apply it in the context of program comprehension. Table 3 presents the activities (i.e., cognitive skills) devised by Fuller et al. (9). Also, we introduce two activities (marked with "*") that stem directly from tasks

performed by subjects in some of our primary studies and require skills that are not covered by the original set of activities. Table 3 also presents examples of the activities, which are either extracted from tasks from our primary studies or general examples when no task involved that activity. In the next section we leverage these activities to gain a better understanding of the tasks conducted by subjects in our primary studies.

2.2.3.2 Mapping Tasks to Learning Activities

We analyzed the tasks that subjects performed in the primary studies (Section 2.2.1) and identified which activities (Table 3) they required. A task can require subjects to conduct one or more activities. For example, Avidan and Feitelson(51) asked subjects to explain what a function does. This is an instance of the Present activity. Miara et al.(21) applied a questionnaire that asked subjects to identify specific code elements and also inquired them about the results of expression evaluations. In this case, both Inspect and Trace activities were executed. Besides the activities in Table 3, we also considered Giving an Opinion (hereafter, "Opinion"), which is not part of the taxonomy because it is not a learning activity. We considered Opinion to reflect the intrinsically biased and unreliable nature of tasks that ask subjects to provide their opinion. Previous work (99, 100) has shown that evidence in software engineering studies often contradicts opinions. This kind of analysis falls outside the scope of this work.

Figure 5 presents a confusion matrix to show the frequency of studies in which tasks and learning activities co-occur. For instance, there are six studies that involves the "Remember the code" task in the context of the Memorize learning activity. The matrix shows that there is a direct correspondence between some tasks and activities. For example, all the instances of the "Find and fix bugs in the code" task involve the Debug activity, and all the tasks that require subjects to provide an opinion are connected to the Opinion activity. In addition, some tasks may be connected to various activities. For instance, "Modify the code" may require subjects to Implement, Trace, Inspect, or Adapt the code to be modified. This makes sense; to modify a program, one may have to understand its static elements and its behavior, as well as adapt code elements to be reused. Another example is "Answer questions about code", which often requires subjects to Trace and Inspect code. Furthermore, "Explain what the code does" is usually related to Present. Notwithstanding, in two studies (76, 46) subjects were presented with multiple descriptions for the same code snippet and asked which one is the most appropriate. This requires the subject to Relate different programs and descriptions.

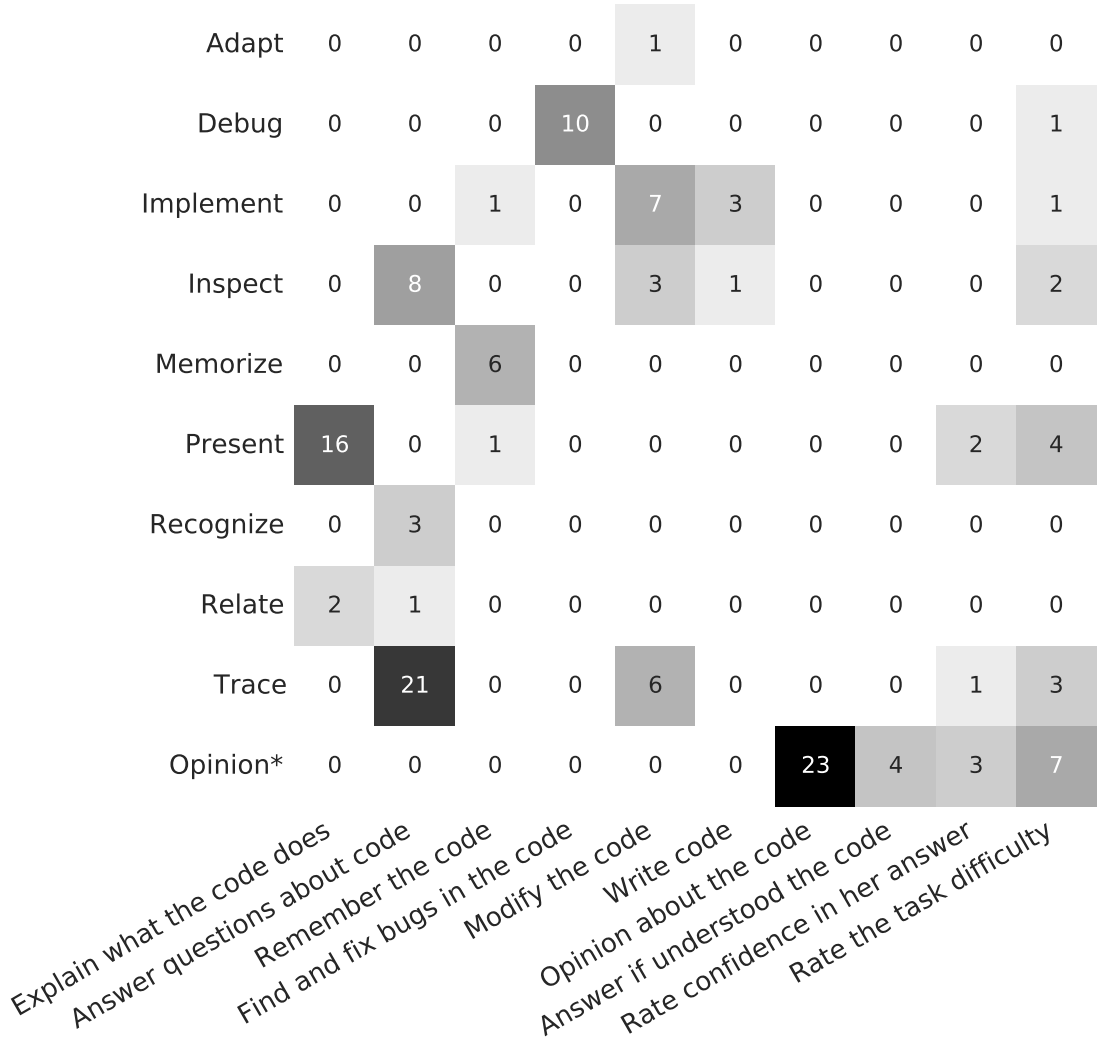


Figure 5 – Confusion matrix of tasks (columns) and learning activities (rows).

Finally, there are some nonintuitive relationships between tasks and activities. Chaudhary (68) asked the subjects to reconstruct a program after spending some time looking at it. This is a "Remember the code" task where the subjects have to explain what the program does by writing a similar program. It is a mix of Present and Implement. In another example, Lawrie et al. (47) asked the subjects to describe what a code snippet does and rate their confidence in their answers. Rating one's confidence in an answer is about Opinion but it also must be associated with some other activity, since some answer must have been given in the first place. In this case, the activity is to Present the code snippet, besides giving an Opinion. A similar phenomenon can be observed for studies where subjects were asked to "Rate the task difficulty" they had performed, e.g., (70, 17).

Figure 6 shows the frequency of the activities required in the analyzed studies, separating readability and legibility studies, as well as those targeting both attributes. Opinion was required by more studies than any learning activity. The two most widely used learning activities

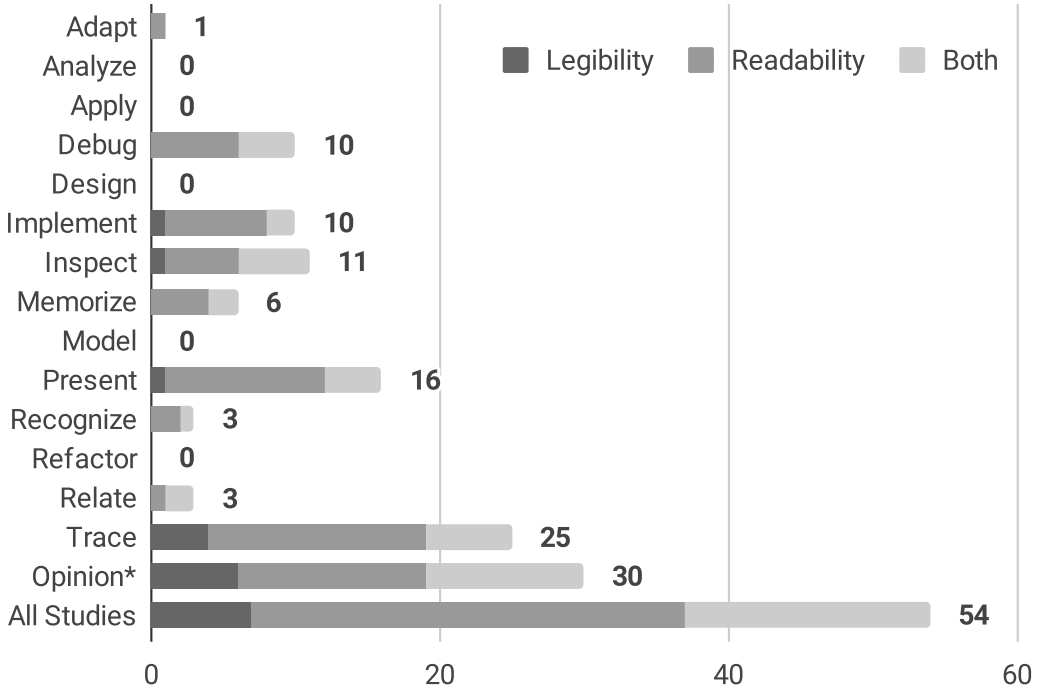


Figure 6 – Frequency of learning activities and subject opinion.

require that subjects extract information from programs. The most widely used one, Trace, was required in 25 studies, where subjects were asked to predict the output value of a program or the value of a variable. Coming in second, the Present activity was required in 16 studies, where usually subjects were asked to explain what the program does. It is also common for subjects to be asked to Inspect the code (11 studies), so as to determine, e.g., how many loops are there in the program. Subjects were required to Implement or Debug code in 10 studies each. Figure 6 also highlights that studies focusing solely on legibility and on readability and legibility combined require subjects to give Opinion more than any learning activity. Even considering the small number of legibility studies, this may be a hint that researchers have yet to figure out the effective ways to meaningfully assess legibility-related attributes.

Figure 7 highlights how many studies use combinations of two activities. More than one activity was employed by 34 of the studies, which combine mainly the most popular activities, i.e., Trace, Present, and Inspect. The numbers in the diagonal indicate how many studies required their subjects to use a single activity. In total, 20 studies employed a single activity. This number amounts to 37% of our primary studies. The use of a single learning activity in an evaluation of readability or legibility suggests a narrow focus, since these attributes are multi-faceted, as highlighted by Table 3.

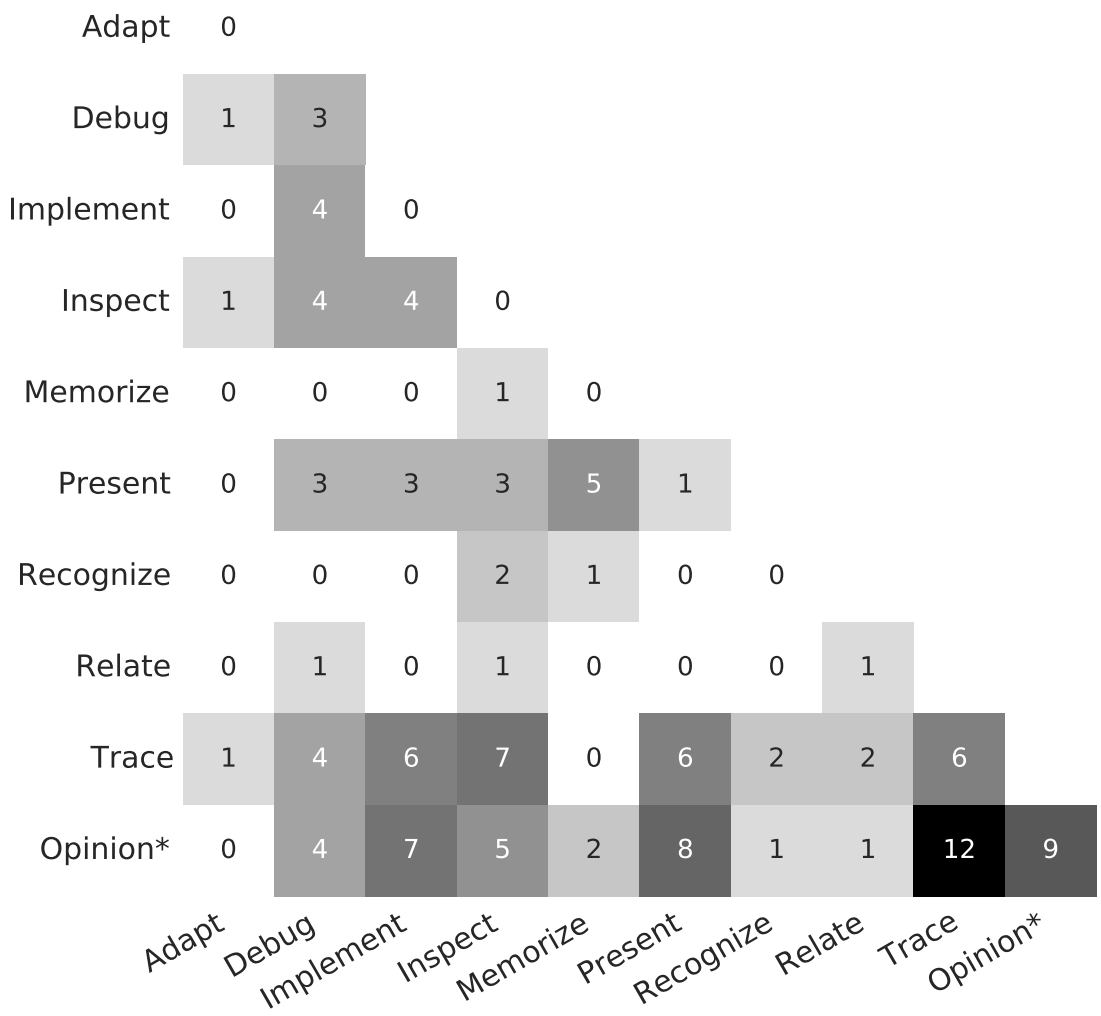


Figure 7 – Co-occurrence of learning activities and subject opinion—the main diagonal represents the number of studies that an activity or opinion is used alone.

2.2.3.3 A Two-Dimensional Model for Program Comprehension

Besides the list of activities, Fuller et al. (9) proposed a taxonomy adapted from the revised version of Bloom's taxonomy (98). It is represented by two semi-independent dimensions, Producing and Interpreting. Each dimension defines hierarchical linear levels where a deeper level requires the competencies from the previous ones. Producing has three levels (None, Apply, and Create) and Interpreting has four (Remember, Understand, Analyze, and Evaluate). Figure 8 represents this two-dimensional model. According to Fuller et al. (9), a level appearing more to the right of the figure along the Interpreting dimension (x-axis), e.g., Evaluate, requires more competencies than one appearing more to the left, e.g., Remember. The same applies to the levels appearing nearer the top along the Producing dimension (y-axis).

The activities in Table 3 were positioned in Figure 8 in conformance to their required

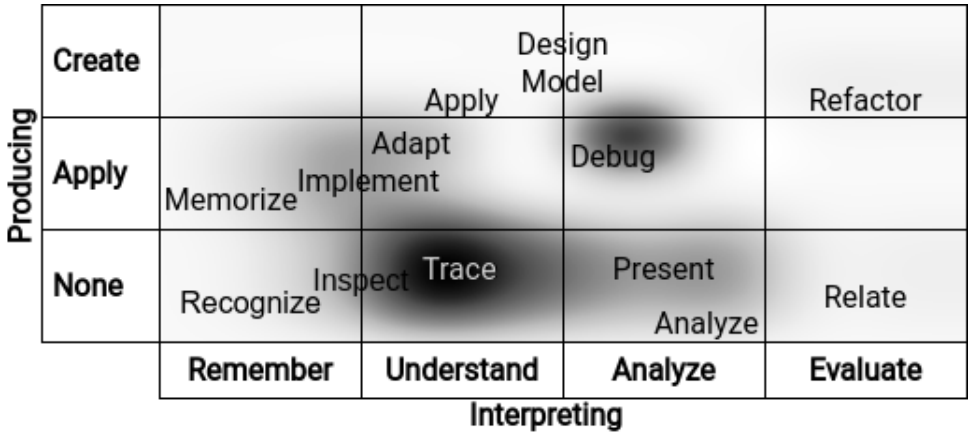
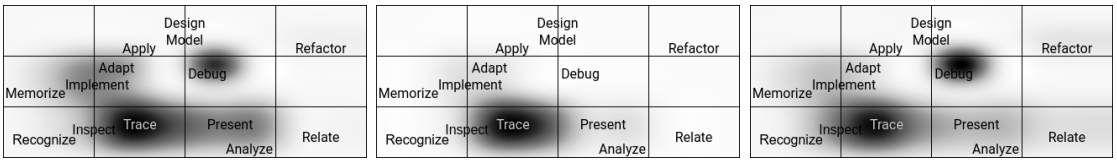


Figure 8 – Learning activities and their frequency.



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

Figure 9 – Learning activities by readability and legibility. This figure decomposes Figure 8 to show spatial disposition.

competencies (we did not consider Opinion as it is not part of the learning taxonomy). We complemented the two-dimensional model by including the two activities that we introduced, Inspect and Memorize. Figure 8 is a heatmap that presents how frequently each competence was required by the studies analyzed in this work. Dark regions are associated to more frequent activities.

The Interpreting dimension indicates that most activities employed in studies evaluating readability and legibility occur at the Understand level, followed by Analyze and, to a lesser extent, Remember. The higher competence level of the Interpreting dimension, Evaluate, is almost never used. Considering the Producing dimension, where the ability to design and build a new product, e.g., a program, is evaluated, we notice that None is the most representative, followed by Apply. This is to be expected since most of the studies focus on comprehension-related activities. However, program comprehension is often required for work that involves the Apply and Create levels. Nevertheless, these levels are rarely tackled by our primary studies. This reinforces the point raised in Section 2.2.3.2, that many of these studies have a narrow evaluation focus.

We built three additional heatmaps to emphasize the differences between readability and legibility studies when considering the two-dimensional model. Figure 9 presents the heatmaps for readability studies, legibility studies, and both studies in sequence. Figure 9a is very sim-

ilar to Figure 8. In contrast, Figure 9b shows that legibility studies are concentrated in the Understand level, where Trace is the main activity. Finally, Figure 9c presents the heatmap for studies that tackle both readability and legibility together. Albeit similar to Figure 8, the area in the intersection between the Analyze level of the Interpreting dimension and the Apply level of the Producing dimension is darker in this figure. This reflects the proportionally higher number of studies that employ the Debug activity.

2.3 THREATS TO VALIDITY

Construct validity. Our study was built on the selected primary studies, which stem from the search and selection processes. The search for studies relies on a search string, which was defined based on the seed papers. We chose only conferences to search for seed papers. A paper published in a journal is often an extension of a conference paper and, in Computer Science, the latest research is published in conferences. Furthermore, we focused on conferences only as a means to build our search string. Journal papers were considered in the actual review. Additionally, we only used three search engines for our automatic search. Other engines, such as Springer and Google Scholar, could return different results. However, the majority of our seed studies were indexed by ACM and IEEE, and then we used Scopus to expand our search. Moreover, while searching on the ACM digital library, we used the *Guide to the Computing Literature*, which retrieves resources from other publishers, such as Springer. Finally, we avoided Google Scholar because it returned more than 17 thousand documents for our search string.

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

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

Conclusion validity. According to Kitchenham et al. (58), conclusion validity is concerned with how reliably we can draw conclusions about the relationship between a treatment and the outcomes of an empirical study. In a systematic review, that relates to the data synthesis and how well this supports the conclusions of the review. The threats of our study related to it are therefore presented as internal validity.

2.4 CONCLUSION

We presented a systematic literature review on how code readability and legibility are evaluated in human-centric studies that compare different ways of writing equivalent code. Our goal was to investigate what tasks are performed by subjects and what response variables are employed in those studies, as well as what cognitive skills are required by those tasks. We presented comprehensive classifications for both tasks and response variables. Moreover, we adapted a learning taxonomy to program comprehension, mapping the tasks identified in the literature review to the cognitive skills that comprise the taxonomy. This study highlighted limitations of primary studies: (i) 37% of them exercised a single cognitive skill; (ii) 16.7% only employed personal opinion as a response variable; and (iii) few studies evaluated readability and legibility in a way that simulates real-world scenarios, where program comprehension is part of a more complex task, requiring higher-level cognitive skills. We also introduced a separation between code readability and legibility. This separation is common in other areas such as linguistics, but, to the best of our knowledge, we are the first to propose it in the context of software engineering. In a new study we will performing systematic literature reviews focusing on the different ways of writing code and which ones improve or hinder legibility (Chapter 3) and readability (Chapter 4).

3 A SYSTEMATIC LITERATURE REVIEW ON THE IMPACT OF FORMATTING ELEMENTS ON CODE LEGIBILITY

In this chapter, we conducted a systematic literature review aiming to find which formatting elements have been investigated in empirical studies and which alternatives were found to be more legible for human subjects. We identified 15 papers containing human-centric studies that directly compared alternative formatting elements. We then analyzed and categorized these formatting elements using a card-sorting technique. This work aim to provide a comprehensive view of the existing knowledge within the literature about the impact of different code alternatives on code legibility (goal #2). This study was published at JSS 2023 (57).

3.1 METHODOLOGY

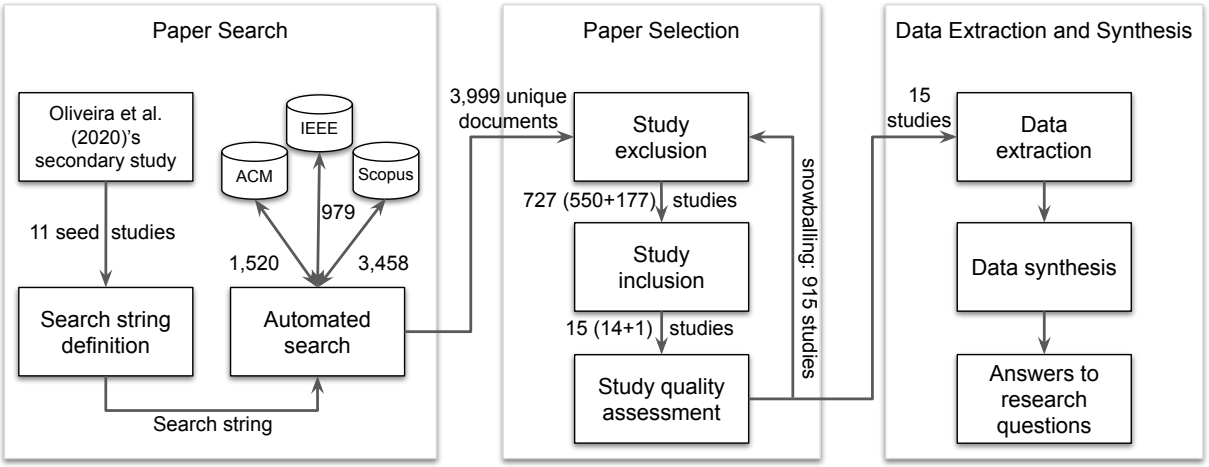


Figure 10 – Systematic literature review roadmap of the study about legibility of formatting elements.

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

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

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

To answer our research questions, we conducted a systematic literature review designed following the guidelines proposed by Kitchenham, Budgen & Brereton(58). The process follows the same step described in the previous work (Section 2.1). Figure 10 presents the roadmap of this review, considering the different numbers of papers resulting in each step. In the following, we describe the summary of the process and the different settings adopted for this study.

3.1.1 Paper Search

We selected 11 papers from our previous study, described in Chapter 2, to be considered as seeds in this one. These papers contain studies that compare two or more ways of writing code by only changing formatting elements to help code understanding in terms of legibility. This is the kind of study we are searching for in this work and, therefore, these papers form the ideal seed set for systematically searching other papers. We analyzed the title and keywords of the seed papers and extracted the general terms related to our research questions. We used the resulting terms to build the following search string:

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

We did not include terms with "legibility" in the search string. Most of the papers with this word in the title or keywords are related to linguistics or computational linguistics. In these fields, researchers use this term with a different meaning than what would be expected in a software engineering paper. Using it in our search string would drastically increase the number of false positives. Finally, we performed an automatic search for studies using our generic search string adapted for three search engines: ACM Digital Library[1], IEEE Explore[2], and Scopus[3]. We retrieved 1,520, 979, and 3,458 documents, respectively, on October 10, 2022. Since a given document might be retrieved from more than one engine, we unified the output of the engines to eliminate duplicates, which resulted in 3,999 unique documents. The 11 seed papers were returned by the automatic search.

[1] <http://dl.acm.org/>, last access May 22, 2023.
[2] <http://ieeexplore.ieee.org/>, last access May 22, 2023.
[3] <http://www.scopus.com/>, last access May 22, 2023.

3.1.2 Paper Selection

We retrieved 3,999 unique documents with the automatic search, which passed through a triage for study exclusion resulting in 550 papers remaining. The exclusion criteria we used took into consideration the rigor and format of the studies we were interested in. We adopted the same exclusion criteria used in the previous study, which is described in Section 2.1.2, since our interests were similar.

In following, we performed an initial study selection where inclusion criteria were applied, resulting in a total of 14. Considering the similarities to previous study, we adopted similar inclusion criteria, described in Section 2.1.3 in terms of methodology and granularity of the primary studies. The list of inclusion criteria is provided below, and the criteria marked with an asterisk (*) were adjusted to fit our study's context, while the others remained the same as the previous study.

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

IC2 (Methodology). The study must be or contain a controlled experiment, quasi-experiment, or survey *involving human subjects*.

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

IC4 (Granularity). The study must target fine-grained program elements and low-level/limited-scope programming activities. Not design or comments, but implementation.

Snowballing. We elaborated nine questions to identify low-quality papers adapted from previous guidelines (64). Three questions were about study design, four about analysis, and two about rigor. Each question was answered with yes (1), partially (0.5), or no (0). Papers were kept if they scored at least 4.5 out of 9. The scores of the papers were: $min = 5$, $median = 8$, $max = 9$. All papers scored above 5, so none were removed due to low quality.

To make sure our study is comprehensive, we carried out a snowballing process. To do so, we collected new papers by gathering all the references used in the included papers (backward snowballing) and all the papers that cited the included papers in our review (forward snowballing). We used Google Scholar to extract the citations of included papers. The newly collected papers passed through the triage (study exclusion), initial selection (study inclusion),

and final selection (study quality assessment). At this stage, only papers published from the 1990s onwards were included because that period marked a significant advancement in the field of software engineering with the emergence of object-oriented programming. We repeated this process until no new paper was found. We performed one complete iteration, and no new paper was found for a second one. With the initial set of 14 selected papers, we gathered 915 new documents. After analyzing them by considering the exclusion and inclusion criteria, we found one new paper, totaling 15 papers for our review.

3.1.3 Data Extraction and Synthesis

Finally, the selected 15 papers were analyzed for data extraction and synthesis to answer our research questions. We detail these steps in the following.

Data extraction. Initially, we read all the papers in full and extracted the necessary data to answer our research questions. The papers were equally divided among the authors, and periodic meetings were held for discussion. For RQ1, we extracted the independent variables of the studies, which are the formatting elements (e.g., indentation) and their levels (e.g., two-space indentation) being compared. For RQ2, for each formatting element, we collected 1) the tasks the human subjects were required to perform in the experiment, which we mapped to learning activities (see Table 3), 2) the dependent variables, which are the response variables (see Table 2), 3) the results and statistical analysis, 4) the characteristics of the human subjects, and 5) the programming languages considered.

Synthesis process. To better understand and present the results in an organized way, we performed card sorting (101) on the extracted formatting elements. With this method, we transformed the formatting elements into cards and then grouped the cards based on their similarities. This process was bottom-up, i.e., formatting elements $\rightarrow$ groups, as follows.

Formatting elements (factors). We used the formatting elements extracted from the studies as factors in the card-sorting process. Then, we created a card for each factor evaluated by the study (e.g., indentation). For synonymous factors, we created one unique card to represent them. For instance, Santos & Gerosa(16) evaluated whether using a beginning block delimiter in the same line of their corresponding statements (e.g., class declaration) is better than in its own line. Similarly, Arab(43) compared different presentation schemes (Peterson, Crider, and Gustafson) that define where the block delimiter should be placed. We created a unique

Table 4 – Card sorting results.

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

card named "Block delimiter location" for these factors. The levels (e.g., in the statement line and in a separate line) of each factor compared by the studies are described and evaluated separately.

Groups. The next step was to group similar cards. In live sessions, we discussed each card and included it in a representative group. When there was no representative group for a card, a new group was created. For example, the first card we analyzed was *"Vertical and horizontal spacing"*. We created the first group, named "Group 1", and included that card in it. The second card we analyzed was *"Formatting layout"*. Such a card is not similar to the card in "Group 1", so we created a second group named "Group 2" for it. The third card we analyzed was *"Indentation"*, which is similar to the card in "Group 1", so we included it in that group as well. After including all cards in some groups, we gave meaningful names to each group. We also split groups that seemed too generic into smaller ones. In cases where two groups were very similar and small, we combined them.

The card sorting process was initially performed by the first two authors of this paper, and the results were later refined with the collaboration of all authors in multiple live sessions. The identified groups and factors are presented in Table 4 and detailed in Section 3.2.

Full disclosure on novelty. Our methodology resembles the one employed in our previous work (56). However, the search for papers is different, which means the whole process started based on a new set of papers for this work. Moreover, the data analysis and synthesis methods are completely new because we answer different research questions in this paper.

3.2 RESULTS

In this section, we present the results of the study. We organized the subsections by the groups of formatting elements (factors) we found in our review, which are named formatting (Section 3.2.1), spacing (Section 3.2.2), block delimiters (Section 3.2.3), long or complex code line (Section 3.2.4), and word boundary styles (Section 3.2.5). In each section, we address both research questions. The first research question aims to identify the formatting elements that researchers investigated in their studies about code legibility, i.e., the characteristics of the code that make it easier or harder to identify its elements. For a given group, we present the formatting elements with their corresponding levels, separated per the primary studies included in our review. Table 4 presents the mapping of groups to formatting elements. With the second research question, we aim to synthesize the studies' findings related to legibility by comparing the levels of the formatting elements. For that, we consider data such as the studies' answers, statistical tests, activities performed by humans, and dependent variables. Table 5 presents an overview of the papers included in our study and summarizes the factors (from Table 4) investigated in each of them.

3.2.1 Formatting

This group gathers studies about different ways of formatting code from a global, higher-level perspective. Table 6 presents a summary of the results. For each factor, the table presents the references to the studies, the levels of the factors compared, the programming languages considered, the dependent variables of each study, the corresponding activities (see Table 3) performed by the subjects, and the main results found for each comparison of levels.

Formatting style. Oman & Cook(89) proposed the notion of Book Format Style for structuring source code and compared it to two well-known styles for the Pascal and C languages: the Lightspeed Pascal style (103) and the Kernighan & Ritchie style (104). The Book Format style, as the name implies, takes inspiration from how books are structured as an approach to organizing source code. In this format, programs include a preface, table of contents, chapter divisions, pagination, code paragraphs, sentence structures, and intramodule comments, among other elements. For the comparison with the Lightspeed Pascal style, the authors asked 36 students to answer 14 multiple-choice, short-answer questions about the characteristics of

Table 5 – Included papers and the factors analyzed, sorted in descending order by year of publication.

Study	Study title	Venue	PL	Formatting elements (factors)
Langhout & Aniche(20)	"Atoms of Confusion in Java"	ICPC'21	Java	– Appropriate use of indentation with blocks – Block delimiter visibility
Bauer et al.(23)	"Indentation: Simply a Matter of Style or Support for Program Comprehension?"	ICPC'19	Java	– Indentation
Medeiros et al.(18)	"An investigation of misunderstanding code patterns in C open-source software projects"	ICPC'21	C	– Block delimiter visibility – Statements per line
Santos & Gerosa(16)	"Impacts of Coding Practices on Readability"	EmSE'19	Java	– Indentation – Vertical spacing between related instructions – Block delimiter location – Line length – Statements per line
Siegmund et al.(81)	"Measuring Neural Efficiency of Program Comprehension"	FSE'17	Java	– Formatting layout
Gopstein et al.(25)	"Understanding Misunderstandings in Source Code"	FSE'17	C/C++	– Block delimiter visibility
Sampaio & Barbosa(40)	"Software readability practices and the importance of their teaching"	ICICS'16	Java	– Blank space around operators and parameters – Block delimiter visibility – Statements per line
Binkley et al.(66)	"The impact of identifier style on effort and comprehension"	EmSE'13	C/Java	– Identifier style
Sharif & Maletic(102)	"An Eye Tracking Study on camelCase and under_score Identifier Styles"	ICPC'10	None	– Identifier style
Furman, Boehm-Davis & Holt(22)	"A Look at Programmers Communicating through Program Indentation"	JSTOR'02	Pascal	– Indentation
Arab(43)	"Enhancing Program Comprehension: Formatting and Documenting"	SIGPLAN'92	Pascal	– Block delimiter location
Oman & Cook(89)	"Typographic Style is More than Cosmetic"	CACM'90	Pascal/C	– Formatting style
Miara et al.(21)	"Program Indentation and Comprehensibility"	CACM'83	Pascal	– Indentation – Block delimiter location
Sykes, Tillman & Shneiderman(82)	"The Effect of Scope Delimiters on Program Comprehension"	SP&E'83	Pascal	– Block delimiter visibility – Block delimiter style
Love(72)	"An Experimental Investigation of the Effect of Program Structure on Program Understanding"	LDRS'77	Pascal	– Vertical and horizontal spacing

a code snippet in a total of 10 minutes (both requiring the Trace activity), and to provide a subjective opinion about the understandability of the code snippet (Giving an opinion). The subjects were randomly assigned to two treatment groups, one for each formatting style, and

Table 6 – Summary of results for the Formatting group.

Factor – Study: Levels (Programming Language) – Dependent Variables (Activities): Results
Formatting style
Oman & Cook(89): book format and Lightspeed Pascal style (Pascal)
Correctness (Trace): The book format style is the best.
Time (Trace): No significant difference.
Opinion: The book format style is the best.
Oman & Cook(89): book format and Kernighan & Ritchie style (C)
Correctness (Trace): The book format style is the best.
Time (Trace): No significant difference.
Opinion: No significant difference.
Formatting layout
Siegmund et al.(81): pretty-printed and disrupted (Java)
Brain Metrics (Relate): No significant difference.

both received the same instructions. Then, the authors assessed (i) the number of questions the students answered correctly (score of 1 to 14 points), (ii) the time they spent answering the questions (1 to 10 minutes), (iii) their performance score (number of correct answers per minute), and (iv) their subjective opinions (rating on a five-point scale). The study concluded that the Book Format style is considered a better formatting style compared to the Lightspeed Pascal style based on the significant differences in correctness score (ANOVA, $p < 0.005$), performance (ANOVA, $p < 0.01$), and subjective understandability opinion (ANOVA, $p < 0.05$), but not for time. In the comparison between the Book Format style and the Kerningham & Ritchie style, the authors prepared a very similar experiment with 44 different students. The only difference is that the questionnaire had 10 questions instead of the 14 from the previous experiment. They assessed the same response variables and found out that the Book Format style was statistically better for correctness score (ANOVA, $p < 0.005$) and performance (ANOVA, $p < 0.005$), but there are no significant differences for time and subjective opinion.

Formatting layout. Siegmund et al.(81) compared a pretty-printed code layout with a disrupted layout. More specifically, they compared the code comprehension of subjects considering code snippets following common coding conventions for layout, e.g., indentation, line breaks, and correctly-placed scope delimiters, and a code snippet with defective layout, e.g., line breaks in the middle of an expression or irregular use of indentation. An interesting differentiating aspect of such a study is that it analyzed bottom-up program comprehension and the impact of beacons (105) and pretty-printed layout using functional magnetic resonance imaging (fMRI). Eleven students and professionals took part in the study. First, these subjects participated in a

training session in which they studied code snippets in Java, including semantic cues, to gain familiarity with them. Then, once in the fMRI scanner, the participants looked at other small code snippets to determine whether they implemented the same functionality as one of the snippets in the training session with a time limit of 30 seconds for each snippet (Relate). In the scanner, the level of blood oxygenation in various areas of the brain was measured. This process is called BOLD (Blood Oxygenation Level Dependent) (106). That metric is a proxy to assess the activation of the area of the brain. To evaluate the role of beacons and layout on comprehension based on semantic cues, they created four versions of the semantic-cues snippets in Java: (i) beacons and pretty-printed layout, (ii) beacons and disrupted layout, (iii) no beacons and pretty-printed layout, (iv) no beacons and disrupted layout. Finally, they extracted the Random-effects Generalized Linear Model (GLM) beta values for each participant and condition to identify differences in brain activation for each program comprehension condition. For that evaluation, we considered the study with code snippets in which there are no beacons to avoid the effect of another variable. Specifically, we avoid beacons because they are not related to formatting elements. The authors did not find significant differences in activation values in the brain areas when comparing code versions with pretty-printed layout and disrupted layout.

3.2.2 Spacing

This group assembles studies about different types of spacing in source code. Table 7 presents a summary of the analyzed results.

Indentation. Miara et al.(21) conducted an experiment with 54 novice students and 32 professional developers and experienced students (i.e., people with three or more years of programming experience) to evaluate the influence of indentation levels and blocked code on program comprehension. In the experiment, the authors used the following independent variables: level of indentation (zero, two, four, and six spaces), level of experience (novice and expert), and method of block indentation (blocked and non-blocked): the latter belongs to block delimiters, so it is presented in Section 3.2.3. Each subject received a program in Pascal using one of the levels of indentation and one of the methods of block indentation, accompanied by a quiz with 10 questions that required subjects to (i) select the correct answer about code characteristics (Trace and Inspect), (ii) explain what the code does (Present), and

Table 7 – Summary of results for the Spacing group.

Factor – Study: Levels (Programming Language) – Dependent Variables (Activities): Results
Indentation
Miara et al.(21): 0, 2, 4, and 6 (Pascal)
Correctness (Trace, Inspect, Present): Two-space indentation is the best.
Opinion: Two-space indentation is the best.
Furman, Boehm-Davis & Holt(22): left [0], normal [2-4], and random (Pascal)
Correctness (Trace, Inspect, Present): No significant difference.
Visual Metrics (Trace, Inspect, Present): Left and normal are better than random indentation on line-look time, normal is better than the other levels on revealed lines, and there is no difference on line-search time.
Opinion: Normal is considered less difficult than other indentation levels for subjective difficulty and preference. However, for fatigue, normal is considered only less difficult than random indentation.
Santos & Gerosa(16): 2 and 4 (Java)
Opinion: No significant difference.
Bauer et al.(23): 0, 2, 4, and 8 (Java)
Correctness (Trace): No significant difference.
Time (Trace): No significant difference.
Visual Metrics (Trace): No significant difference.
Opinion: No significant difference.
Appropriate use of indentation with blocks
Langhout & Aniche(20): with and without (Java)
Correctness (Trace): Appropriate use of indentation is the best only when it is not preceded by curly braces.
Opinion: Appropriate use of indentation is the best.
Vertical and horizontal spacing
Love(72): paragraphed and unparagraphed (Pascal)
Correctness (Memorize and Present): No significant difference.
Vertical spacing between related instructions
Santos & Gerosa(16): with and without (Java)
Opinion: No significant difference.
Blank space around operators and parameters
Sampaio & Barbosa(40): with and without (Java)
Opinion: No significant difference.

(iii) provide an opinion on the difficulty of the task (Giving an opinion). The authors considered two dependent variables: the number of questions correctly answered (score between 1-10) and the rating of the subjective opinion. The results showed that there are significant differences between novices and experts (ANOVA, $p < 0.001$) and between indentation levels (ANOVA, $p = 0.013$), where two-space indentation resulted in the best results.

Furman, Boehm-Davis & Holt(22) also evaluated the effect of the left (no space), normal (2-4 spaces), and random indentation on code legibility. They conducted an experiment with

24 inexperienced and 18 experienced programmers in Pascal. The subjects had to understand three sorting programs, one at a time, presented in one indentation version. The authors used DISCOVERY (a tool developed by them) to measure visual metrics. This tool printed the programs to mask the code lines with the character 'X' and only allowed subjects to look at one line at a time. After understanding the program, the subjects had to answer eight multiple-choice questions for each of the three sorting programs (Trace, Inspect, and Present). The authors evaluated the correctness of the answers, the average line-look time (i.e., time spent looking to a revealed code line), the average line-search time (i.e., time spent choosing a code line to be revealed), and the total of code lines revealed in a program. Also, the authors asked subjects to answer a subjective questionnaire (Giving an opinion) using the Likert scale to evaluate the difficulty of performing the task, the subjective preference, and the level of fatigue. They found a statistical difference between different indentation versions in terms of correctness. With the visual metrics, they found a statistical difference in line-look time (F-test, $p < 0.05$) and in the total of revealed lines (F-test, $p < 0.05$). For the line-look time, subjects spent less time in program versions with left and normal indentation than in programs with random indentation, and for the total of revealed lines, they spent less time in programs with normal indentation than in other versions. The authors did not find a statistical difference for line-search time. Finally, the subjects considered the version with normal indentation less difficult than others (F-test, $p < 0.05$), as well as their subjective preference (F-test, $p < 0.05$). However, they considered that version with normal indentation only was less fatigued than the random version (F-test, $p < 0.05$).

Santos & Gerosa(16) performed a survey with 55 students and 7 professionals to investigate the impact of a set of Java coding practices on code understandability. Among other practices, subjects had to choose (Giving an opinion) between two alternatives for indentation level: two and four spaces. The authors evaluated the results by comparing the proportions of the votes to each alternative. The results showed that there is no statistical difference (Two-tailed test for proportion, $p = 0.32$) between these alternatives.

Bauer et al.(23) conducted an experiment with 7 developers and 15 students to investigate the influence of levels of indentation (zero, two, four, and eight spaces) on code comprehension. They designed an experiment similar to the one of Miara et al.(21), with the differences that they asked subjects for the output of code snippets in an open box instead of providing subjects with multiple choices, and did not ask for a description of code functionality. They assessed code comprehension by (i) the correctness of answers (Trace) and (ii) the rank of perception of

task difficulty (Giving an opinion). Differently from the work of Miara et al.(21), this experiment included eye-tracking to measure visual effort using fixations, which occur when the gaze is resting on a point, and saccades, which are the transition between two fixations. Furthermore, it measured the time subjects spent providing an answer. The code snippets in this experiment were in Java. The authors first applied Mauchly's sphericity test to evaluate which test was more adequate for each dependent variable, i.e., one-way ANOVA with repeated measures or Friedman's test. The results of this experiment were: the correctness of answers (Friedman's test, $p = 0.36$), the log-transformed response time (ANOVA, $p = 0.72$), perceived difficulty (Friedman's test, $p = 0.15$), fixation duration (ANOVA, $p = 0.045$), fixation rate (Fridman test, $p = 0.06$), saccadic amplitude (ANOVA, $p = 0.18$). Despite the $p < 0.05$ for fixation duration, the authors could not confirm this difference with a post hoc test. Therefore, unlike in the previous study, there was no statistically significant difference between indentation levels.

Appropriate use of indentation with blocks. In the work by Langhout & Aniche(20), a partial replication of the study presented by Gopstein et al.(25) (further explained in Section 3.2.3) was conducted, focusing on the Java language instead of C/C++. These papers measure the impact of atoms of confusion in code understandability. The authors recruited 132 novice developers for an experiment conducted in two parts, in which participants had to (i) predict the output of code snippets with and without atoms (Trace) and (ii) give their opinions about which ones are confusing among two functionally-equivalent code snippets, one with an atom of confusion and another without (Giving an opinion). For the first part, the authors calculated the odds ratio of subjects correctly predicting the output of the code snippets as a measure of effect size. Two atoms of confusion related to inappropriate use of indentation were investigated: the Remove Indentation atom and the Indentation atom. In both cases, there exist indented statements after the end of a block, which makes it look like the statements are part of the block. In the latter case, the incorrectly indented statement is preceded by curly braces, whereas in the former, it is not. The authors found out that Remove Indentation is associated to misunderstanding (odds ratio $= 56.21$, $p < 0.05$), and for 57.1% of the subjects, this atom is unfavorable for legibility. There was no statistical difference in answers for Indentation, although 72.7% of the subjects considered it confusing.

Vertical and horizontal spacing. Love(72) evaluated the use of paragraphed vs. unparagraphed code, i.e., the disciplined use of vertical and horizontal spacing to organize the code in text-like paragraphs. In the paragraphed version, the source code contains blank lines between

different instructions and indentation to represent blocks. Unlike, there are no blank lines or indentation in the unparagraphed version. The author experimented with 19 undergraduate and 12 graduate students to investigate the effect of program indentation (paragraphed or unparagraphed) and control flow (simple or complex) on code comprehension. The subjects were asked to memorize a program in Pascal in 3 minutes, rewrite the program in 4 minutes (Memorize), and describe the program functionality (Present). The reconstructed programs were scored based on the percentage of lines of source code correctly recalled in the proper order. The descriptions of the program functionality were rated with a 5-point scale, where one point means that a description has nothing right about the program functionality and five points means that a description shows a complete understanding of it. The author did not find a statistically significant difference between paragraphed and unparagraphed programs, considering the correctness scores in the Memorize activity (ANOVA, $p = 0.79$) and in the Present activity (ANOVA, $p = 0.6$) for both undergraduate and graduate students.

Vertical spacing between related instructions. Another coding practice analyzed by Santos & Gerosa(16) pertains to the use of vertical spaces. More specifically, the authors asked the subjects (by presenting code examples) whether blank lines must be used to create a vertical separation between related instructions (Giving an opinion). The results did not show a statistically significant difference (Two-tailed test for proportion, $p = 1.0$) between subjects who agree and disagree with this practice.

Blank space around operators and parameters. Sampaio & Barbosa(40) investigated the importance of teaching a set of best practices for code understandability. For this, the authors conducted a survey to ask object-oriented programming teachers to provide a subjective rating on a 5-point Likert scale about the importance of coding practices for code understandability (Giving an opinion). The authors considered that practices with a median rating greater than 3 are relevant to code understandability. Four practices related to legibility were investigated, among others. The practices related to spacing, which are the ones relevant to this section, are about blank space around operators and arguments/parameters: *"The assignment operator include blank space before and after"* and *"Blank space between the arguments/parameters of functions"*. The authors found out that the first ($median = 3$, $range = 4$, $interquartile = 2$) and the second ($median = 2$, $range = 4$, $interquartile = 2$) practices are not relevant for legibility.

Table 8 – Summary of results for the Block Delimiters group.

Factor – Study: Levels (Programming Language) – Dependent Variables (Activities): Results
Block delimiter location
Miara et al.(21): blocked and non-blocked (Pascal)
Correctness (Trace, Inspect, Present): No significant difference.
Opinion: No significant difference.
Arab(43): in the statement line and in separate line (Pascal)
Opinion: Block delimiters in their own lines is the best.
Santos & Gerosa(16): in the statement line and in separate line (Java)
Opinion: Block delimiter in its own line is the best.
Block delimiter style
Sykes, Tillman & Shneiderman(82): `ENDIF/ENDWHILE` (without `BEGIN`) and `BEGIN-END` in all blocks (Pascal)
Correctness (Trace): `ENDIF/ENDWHILE` is the best.
Opinion: `ENDIF/ENDWHILE` is the best.
Sykes, Tillman & Shneiderman(82): `ENDIF/ENDWHILE` (without `BEGIN`) and `BEGIN-END` only in compound statement blocks (Pascal)
Correctness (Trace): `ENDIF/ENDWHILE` is the best.
Opinion: `ENDIF/ENDWHILE` is the best.
Block delimiter visibility
Sykes, Tillman & Shneiderman(82): omitted and present (Pascal)
Correctness (Trace): No significant difference.
Opinion: No significant difference.
Sampaio & Barbosa(40): omitted and present (Java)
Opinion: No significant difference.
Gopstein et al.(25): omitted and present (C/C++)
Correctness (Trace): Present block delimiters is the best.
Medeiros et al.(18): omitted and present (C)
Opinion: Present block delimiters is the best.
Langhout & Aniche(20): omitted and present (Java)
Correctness (Trace): Present block delimiters is the best.
Opinion: Present block delimiters is the best.

3.2.3 Block Delimiters

This group gathers studies about different types of block/scope delimiters. Table 8 presents a summary of the analyzed results.

Block delimiter location. Miara et al.(21) compared two styles of block indentation: (i) blocked, where the code within a block is at the same indentation level as the block delimiters (begin-end), and (ii) non-blocked, where the code within the block appears at least one more indentation level to the right of the block delimiters. This comparison is different from what has been previously reported about indentation (Section 3.2.2) because this one evaluated when the indentation of the block starts (i.e., in the line of the block delimiter itself or in the first

statement after the block delimiter), while the other one focuses solely on the indentation level. The design of this study was explained in Section 3.2.2. The analysis of variance (ANOVA) of the quiz scores and the program ratings showed no significant effect with the blocked and non-blocked styles.

Arab(43) compared three combinations of the use of block delimiters in the same line as their associated statements with the use of block delimiters in a separate line. More specifically, they evaluated three different code presentation schemes in Pascal that propose different patterns to place the block delimiters: (i) Peterson's scheme: BEGIN and END in their own lines (107); (ii) Crider's scheme: BEGIN and END in the same line of the last statement or declaration (108); and (iii) Gustafson's scheme: BEGIN in the same line as the last statement but END in its own line (109). The study consisted of an opinion survey with 30 subjects (22 novices and 8 experts), who were asked to classify the three schemes in descending order (Giving an opinion). The results showed that the block delimiters in their own lines, i.e., Peterson's scheme, was chosen as the best by most participants. However, they did not apply any statistical test to evaluate the results. In a similar vein, Santos & Gerosa(16) asked subjects whether they prefer to have Java's opening curly braces in their own lines or in the same line as their corresponding statements (e.g., class declaration). They found a statistically significant difference (Two-tailed test for proportion, $p = 0.006$) between subjects who preferred the first alternative and subjects who preferred the second alternative, where most participants consider that opening braces in their own lines are easier to read, similar to results of Arab(43).

Block delimiter style. In the study reported by Sykes, Tillman & Shneiderman(82), different styles of scope delimiters in Pascal were investigated: (i) ENDIF, which uses `IF-ENDIF` and `WHILE-ENDWHILE` scope delimiters for the bodies of conditional and iterative statements, respectively; (ii) REQ-BE, which always uses `BEGIN-END` scope delimiters for simple and compound statements within a conditional structure; and (iii) BE, which uses `BEGIN-END` scope delimiters only for compound statements. The study employed a questionnaire where most of the questions asked subjects to determine the values of variables by simulating the execution of program segments in Pascal with initial values given in the questions (Trace). In total, 36 students participated in that study, and they had 25 minutes to finish the questionnaire. The subjects were divided into advanced (at least two years of programming experience) and intermediate. The authors evaluated the responses to the questionnaire considering the type of delimiter and the experience of the subjects. ANOVA indicated that experience was significant

($p = 0.005$), and advanced subjects did much better than intermediate ones. Also, a paired t-test (considering an $\alpha = 0.1$) showed that the subjects performed better using the ENDIF delimiters than using the REQ-BE ($p = 0.074$) and BE ($p = 0.073$) delimiter styles.

Block delimiter visibility. Also, in the work of Sykes, Tillman & Shneiderman(82), we found an evaluation of block delimiter visibility in Pascal, i.e., whether omitted or present block delimiters impact code legibility. We limited the comparison analysis between REQ-BE and BE to evaluate the block delimiter visibility. The details of the study are explained in the previous paragraph. The authors did not find a difference between the REQ-BE and BE delimiter styles (t-test, $p = 0.955$).

The work of Sampaio & Barbosa(40) (introduced in Section 3.2.2) evaluated the relevance of the practice "*use { and } to enclose the statements in a loop*" for code legibility, in the context of Java code. They found that this practice is not relevant ($median = 2$, $range = 4$, $interquartile = 2$) for legibility.

Gopstein et al.(25) compared the use of omitted and present block delimiters in the specific context of small C programs. The authors conducted a broad-scoped experiment[4] with 73 programmers with three or more months of experience in C/C++ to evaluate small and isolated patterns in C code, named atoms of confusion, that may lead programmers to misunderstand code. They asked subjects to analyze tiny programs (∼8 lines), where the control version contained a single atom of confusion candidate, and the treatment version contained a functionally-equivalent version where the atom does not exist. For each program, the participants had to predict its output (Trace). The authors compared the correctness of the answers of the version with the atom of confusion and their counterparts without the atoms. One of these atoms is called *Omitted Curly Braces*. In C programs, this is used when the body of an `if`, `while`, or `for` statement consists of a single statement. The study found a statistically significant difference (McNemar's test adjusted using Durkalski correction, $p < 0.05$) between the programs where the curly braces were omitted and the ones where they were not. Subjects analyzing programs with omitted curly braces made more mistakes. Langhout & Aniche(20) (introduced in Section 3.2.2) also evaluated the atom *Omitted Curly Braces*, but in Java code. They found a statistically significant difference (odds ratio $= 4.62$, $p < 0.05$), indicating that omitting curly braces is associated with misunderstanding. Furthermore, 93.3% of the subjects agreed that a code snippet with this atom is hard to read.

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

Table 9 – Summary of results for the Long or Complex Code Line group.

Factor – Study: Levels (Programming Language) – Dependent Variables (Activities): Results
Line length
Santos & Gerosa(16): limit of 80 characters and exceed of 80 characters (Java)
Opinion: Line length within the limit of 80 characters is the best.
Statements per line
Sampaio & Barbosa(40): multiple and one (Java)
Opinion: One statement per line is relevant for code legibility.
Santos & Gerosa(16): multiple and one (Java)
Opinion: One statement per line is the best.
Medeiros et al.(18): multiple and one (C)
Opinion: No significant difference.

Similarly, Medeiros et al.(18) investigated what they call misunderstanding code patterns, which are very similar to atoms of confusion. One study was a survey with 97 developers, where the subjects should determine the negative or positive impact of using one code snippet version containing one misunderstanding pattern instead of a functionally-equivalent alternative without that pattern on a 5-point Likert scale (Giving an opinion). In another study, they submitted 35 pull requests suggesting developers remove several misunderstanding patterns. This study focused on the C language. In this section, we analyze the results of the misunderstanding pattern named *Dangling Else*, which happens when an `if` statement without braces contains one `if`/`else` statement without braces. The reader of the code needs to know that the `else` clause belongs to the innermost `if` statement. The first study found that *Dangling Else* is perceived as negative by 71.73% of the developers. In the second study, they submitted 10 pull requests to remove instances of the *Dangling Else* pattern, where two of them were accepted, one was rejected, three were not answered, and four were ignored by developers (according to the authors, due to the pattern instances being in third party or deprecated code).

3.2.4 Long or Complex Code Line

This group comprises studies that evaluate elements related to the size and complexity of code lines. Table 9 presents a summary of the analyzed results.

Line length. In the study of Santos & Gerosa(16) (details about the methodology are in Section 3.2.2), the subjects were asked if they agree that the practice "*line lengths not exceeding 80 chars*" has a positive impact on code legibility. The authors found out that students and

professionals find code snippets containing lines of up to 80 characters more legible than code snippets where there are longer lines (Two-tailed test for proportion, $p < 0.001$).

Statements per line. Three studies tackle the question of whether lines of code including a single statement are more legible than ones with multiple statements. Sampaio & Barbosa(40) evaluated the relevance of the practice of *"breaking the line after semicolon"* for code legibility (the methodology is detailed in Section 3.2.2). The authors found that this practice is relevant for code legibility ($median = 4$, $range = 3$, $interquartile = 1.5$).

Santos & Gerosa(16) found out that students and professionals agreed with the practice "*avoid multiple statements on a same line*" (Two-tailed test for proportion, $p < 0.001$), i.e., the subjects prefer code snippets where lines of code do not include multiple statements (details about the methodology are in Section 3.2.2). As discussed before, the paper of Medeiros et al.(18) (Section 3.2.3) introduced and evaluated what the authors call misunderstanding code patterns. Among the studied patterns, one specifically refers to a scenario where multiple variables are initialized on the same line. The study asked subjects their opinion about the use of this pattern (Giving an opinion). The authors found out that the use of *multiple initializations on the same line* was neither negative nor positive for most of the subjects. In the second study, they submitted one pull request to remove this pattern, which was rejected by developers.

3.2.5 Word Boundary Styles

The fifth and last group we have identified comprises studies of word boundary styles for identifiers. Table 10 presents a summary of the analyzed results.

Identifier style. Sharif & Maletic(102) investigated the effect of the camel case and underscore identifier styles on code legibility. They compared the identifier styles using phrases formed by two and three words from code, i.e., phrases that are likely to be in source code (e.g., "start time"), and non-code, i.e., phrases that are not likely to be in source code (e.g., "river bank"). In the experiment, 15 students had to read a phrase and, on the next screen, choose an identifier (from four choices) that exactly matches the phrase they just saw (Memorize), and answer it verbally. Only one of the choices is correct, and the rest are distracters that change the beginning, middle, or end of the identifier. It was repeated for eight phrases. The authors measured the correctness, time, and visual effort for each phrase. They found no difference between identifier styles for correctness. Using a simple linear mixed model, they found a

Table 10 – Summary of results for the Word Boundary Styles group.

Factor – Study: Levels (Programming Language) – Dependent Variables (Activities): Results
Identifier style
Sharif & Maletic(102): camel case and underscore (None)
Correctness (Memorize): No significant difference.
Time (Memorize): Underscore is the best.
Visual Metrics (Memorize): Underscore is the best for average fixation duration.
Binkley et al.(66): camel case and underscore (C and Java)
Correctness (Inspect, Memorize, Present): Camel case is the best only in the Inspect activity.
Time (Inspect, Present): Camel case is the best only in the Inspect activity.
Visual Metrics (Memorize): Camel case is the best.

significant difference ($p = 0.0001$) in time where subjects took 13.5% longer for camel-cased identifiers than underscored. For visual effort, they employed two metrics: fixation rate and average fixation duration. They were measured when the subjects looked at the correct answer and the distracters. Using Wilcoxon test, they only found a significant difference in the average fixation duration (correct: $p = 0.015$; distracters: $p = 0.026$), where the distribution showed that camel-cased identifiers required a higher average duration of fixations than underscored.

Binkley et al.(66) performed five studies to evaluate the effect of identifier style on code comprehension. Only three are related to source code: Where's Waldo study, Eye Tracker Code study, and Read Aloud study. In the first study, 135 programmers and non-programmers attempted to find all occurrences of a particular identifier in code fragments (Inspect) written in C and Java. The authors measured the number of lines where the subjects misread occurrences of the identifier and the time spent on that task for the subjects that found all identifier occurrences. Using a Linear mixed-effects regression, they found a significant difference ($p = 0.0293$) in favor of camel-cased identifiers, to an estimated 0.24 fewer missed lines than with underscored ones. For the time, it was marginally significant ($p = 0.0692$) and indicated that camel-cased identifiers took, on average, 1.2 fewer seconds to be found than underscored ones.

In the second study, they asked 15 programmers (undergraduate and graduate students) to study two C++ code snippets, reproduce them, and answer questions by selecting the identifiers they remember from the code (Memorize). The authors considered the correctness of the answers. They also used an eye tracker to measure eye fixations and gaze duration, which is the duration of a fixation within a specific area of interest (boxes around the identifiers with some extra padding). The authors compared differences in visual effort using two pairs of

similar identifiers, *row_sum* and *colSum*, and *c_sum* and *rSum*. They found out that *row_sum* required significantly more fixations than *colSum* (1-tailed, $p = 0.007$), and that its average gaze duration was 704ms longer than that of *colSum* (1-tailed, $p = 0.005$). No significant differences were found between *c_sum* and *rSum*. Furthermore, they did not find a statistical difference (Linear mixed-effects regression, $p = 0.451$) for correctness.

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

3.2.6 Addressing the two research questions

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

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

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

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

For RQ2, researchers found statistically significant results in 17 comparisons and no significant results in 10 comparisons. Considering the significant differences, it was found that the best alternative levels for four factors were: book format style (cf. Lightspeed Pascal and Kernighan & Ritchie styles), appropriate use of indentation with blocks (cf. inappropriate use),

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

3.3 DISCUSSION

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

3.3.1 Contrasting empirical results with existing coding style guides

In this section, we discuss the results of this study in the light of five existing style guides for Java (Google Java Style Guide[5]), JavaScript (AirBNB JavaScript Style Guide[6]), Python (Style Guide for Python Code[7]), C (Linux Kernel Coding Style[8]), and Pascal (Free Pascal[9]), and contrast their recommendations against the findings of this study.

Formatting. We found three comparisons of formatting styles, but only two of them showed statistically significant results. In particular, the Book Format style (89) exhibited better results

[5] <https://google.github.io/styleguide/javaguide.html>, last access May 22, 2023.
[6] <https://github.com/airbnb/javascript>, last access May 22, 2023.
[7] <https://peps.python.org/pep-0008/>, last access May 22, 2023.
[8] <https://www.kernel.org/doc/html/v4.10/process/coding-style.html>, last access May 22, 2023.
[9] <https://wiki.freepascal.org/Coding_style>, last access May 22, 2023.

compared to the Lightspeed Pascal and Kernighan & Ritchie styles. However, one of these results is from a study that used Pascal, which is not a popular programming language in 2022[10,11], and at the time this comparison was performed, formatting tools were still scarce. Such results may not be applicable in our current context because considerable advances have been made in programming languages and tools. Furthermore, no existing coding style guide supports the use of the Book Format style.

Spacing. Most studies that evaluated spacing did not produce statistically significant results. The results of one study (72) indicate that the disciplined use of vertical and horizontal spacing to organize code into text-like paragraphs is not relevant. We hypothesize that this may stem from a lack of statistical power: different spacing approaches are not likely to yield big effect sizes, and the power required to detect such small effect sizes implies large sample sizes (Section 3.3.2). Two studies (16, 23) did not find a significant difference between different indentation levels, e.g., two versus four spaces, in Java code. An older study (21) comparing zero, two, four, and six spaces in Pascal code snippets found that two spaces exhibited the best result. The coding style provided by Free Pascal recommends two spaces for indentation. Although Miara et al.(21)'s finding is for Pascal, it is also consistent with some coding guides such as the Google Java Style Guide, the AirBNB JavaScript Style Guide, and the Style Guide for Python Code. Others, such as the Linux Kernel Coding Style, recommend eight spaces and explicitly argue against two or four spaces. Currently, the developer community uses spacing patterns, both vertical and horizontal, because they believe in their usefulness.

Block delimiters. The majority of the comparisons related to block delimiters (7/10 comparisons) showed statistically significant results. They evaluated the location, visibility, and style of block delimiters. The studies (in Pascal, C, C++, and Java) suggest that block delimiters are relevant to legibility, and should be visible and stay in their own line. The Google Java Style Guide is consistent with this result, suggesting the use of braces even when the block is empty or contains only a single statement (block delimiter visibility), but not about the placement of braces (block delimiter location). The Linux Kernel Coding Style agrees with this for functions but disagrees for statements. In addition, the AirBNB JavaScript Style Guide prescribes that the opening brace of a statement or declaration should be placed in the same line as the statement or declaration and not in a separate line.

Some languages allow omitting the block delimiters in some cases, e.g., when the block

[10] <https://www.tiobe.com/tiobe-index/>, last access May 22, 2023.
[11] <https://redmonk.com/sogrady/2022/03/28/language-rankings-1-22/>, last access May 22, 2023.

consists of only one line, to make the code less verbose. The Linux Kernel Coding Style argues in favor of this practice. The Google Java Style Guide and the AirBNB JavaScript Style Guide argue in the opposite direction. The results obtained by Gopstein et al.(25) suggest that omitting braces may be bug-prone. Furthermore, using names to block delimiters that indicate the context of the block seems to increase code comprehension. It becomes apparent when the code has nested blocks, where the information about the block in the name of delimiters could avoid misunderstanding. To the best of our knowledge, no programming language in widespread use supports this approach.

Long or complex code line. One study investigated and found that keeping line lengths within 80 characters in Java is considered positive for legibility (16). On the other hand, Sampaio & Barbosa(40), Santos & Gerosa(16), and Medeiros et al.(18) investigated the use of one vs. multiple statements per line and obtained different results. Sampaio & Barbosa(40) and Santos & Gerosa(16) found out that one statement per line in Java is preferred by the participants, but Medeiros et al.(18) did not find a significant difference between one or more statements per line in C. In these three studies, only the subjects' opinions were used as the dependent variable. The Style Guide for Python Code and Linux Kernel Coding Style agree with the 80-character limit. The Google Java Style Guide recommends one statement per line but recommends 100-character lines. The AirBNB JavaScript Style Guide also establishes 100 characters as the limit for lines, but it makes no prescription about the number of statements per line.

Word boundary styles. The two studies that investigated word boundary styles for identifier names revealed divergent results. While Binkley et al.(66) found that camel case is a positive standard for legibility in comparison to snake case (a.k.a. underscore) in Java and C, Sharif & Maletic(102) found that snake case is the best alternative (no specific programming language was considered). The camel case style is adopted in the examined coding style guides for Java and JavaScript. Python uses camel case for class names and for method names *"where that's already the prevailing style [...] to retain backwards compatibility"*. For most names, it suggests the use of snake case, with additional trailing or leading underscores for special identifiers. The Linux Kernel Coding Style explicitly argues against camel case and prescribes the use of snake case.

3.3.2 Statistical power of the analyzed studies

The power of a statistical test describes the probability that a statistical test will correctly identify a genuine effect (110). In other words, a statistical test with sufficient power is unlikely to accept a null hypothesis when it should be rejected. A scenario where a null hypothesis is rejected when it should not have been rejected is called a Type II error (111). The calculation of the sample size required to achieve a certain level of statistical power is called power analysis. It depends on the significance criterion, the sample size, and the population effect size.

For nine of the comparisons we have investigated, it was not possible to reject the null hypothesis. However, it is not clear whether these results stem from the absence of a statistically significant difference or from a lack of power. Among the analyzed studies, only the works of Gopstein et al.(25) and Bauer et al.(23) report a concern with this aspect. In both cases, the authors have calculated the sample sizes that would yield what they considered an acceptable level of statistical power. Null results reported by other studies are difficult to judge because they have no associated confidence level. This is due to the lack of power analysis and reporting of effect sizes, and the generally low sample sizes. These low sample sizes imply that these studies are only able to detect large effect sizes. The study of Siegmund et al.(81) presents an illustrative example. It attempted to compare the differences in brain activation of subjects exposed to pretty-printed code and code whose layout is disrupted. Since this study involved only 11 participants, its power is very low, and it can only detect statistically significant differences if the effect size is large.

Another example is the pair of studies performed by Miara et al.(21) and Bauer et al.(23). Both compared different indentation levels, with the latter being inspired by the former. On the one hand, the study of Miara et al. involved 86 subjects and found a statistically significant difference between different indentation levels and also between novices and experts. On the other hand, the study conducted by Bauer et al. had only 22 participants and could not find statistically significant differences. Although still not common in Software Engineering, meta-analyses (110) could be leveraged to combine the results of these different studies. In this manner, it would be possible to identify subtler differences if they exist.

3.3.3 Limitations of our study and of existing studies

The studies analyzed in this work were conducted between 1977 and 2021 and covered several formatting elements. Nevertheless, there are other aspects beyond what we captured. For example, in our work, we did not consider studies that investigate the influence of typography, colors, contrast, or dynamic presentation of program elements on the ability of developers to identify program elements. In summary, aspects that fall outside what can be tinkered with at the source code level, in an ASCII text editor, are beyond the scope of this work. Investigating aspects that go beyond these limits is left for future work.

Even within the strictly-defined boundaries we adhere to, much is still unknown. This study highlights the limitations in our current understanding of the impact of formatting elements on code legibility. We have found only 15 papers containing direct comparisons between alternative levels of formatting elements and styles out of 4,914 examined documents. Replications, even partial ones, are almost non-existent. This suggests that this area of research is **immature** and can greatly benefit from new studies.

As pointed out when discussing statistical power, many previous studies are **inconclusive**, and it is still not clear if well-established practices actually have any effect on legibility. For example, we identified only one paper that reports on the comparison of different types of block delimiters (82), and that paper was published 40 years ago, at a time when software development was very different from what it is today. Another example is indentation. Although many style guides provide directives about how to indent code, most of the studies on the topic found inconclusive results (Section 3.2.2). Notwithstanding, the importance of the topic for software developers in practice emphasizes that more investigation is required. One point that can be raised against this kind of study is whether such choices matter at all, especially for experienced developers, since these can be seen as minute details. Previous work (92, 25, 112, 18) suggests that they do, though, even in very mature, high-complexity projects (112).

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

disconnected from current practice and software development methods and tools.

The learning activities involved in the analyzed studies (Table 3) required participants to Memorize, Inspect, Trace, Present, and Relate, plus the task of Giving an opinion. The learning activities were modeled in two semi-independent dimensions by Fuller et al.(9) and extended by Oliveira et al.(56). Each dimension defines hierarchical linear levels where a deeper level requires the competencies from the previous ones. This means that some activities require more competencies than others. The majority of the comparisons of formatting element levels we found in the literature were based on the Giving an opinion task, which is not modeled because it is not a learning activity. Trace is the most used learning activity, but it does not even require an intermediate level of competencies. The activity used in the analyzed studies that requires more competencies is Relate, which was used for only one formatting element evaluation. More generally, the analyzed studies focused on "lower" cognitive skills according to the model of Fuller et al.(9), emphasizing their **narrow scope**. Program comprehension aims to support other activities such as performing maintenance, fixing bugs, and writing tests. The absence of studies requiring participants to conduct these activities points to a possible direction for future work.

Since our ultimate goal is to create a coding style guide based on empirical evidence, an analysis of the programming languages used in the experiments of the primary studies included in our review is necessary. Five studies used the Java language (40, 81, 16, 23, 20). Another five studies used the Pascal language (72, 21, 82, 43, 22). Two more targeted the C or C++ languages (25, 18). The paper by Oman & Cook(89) used the languages C and Pascal, and the paper by Binkley et al.(66) used the languages C and Java. Sharif & Maletic(102) did not specify a programming language. This characteristic of the studies makes it impossible to generalize the results of our study and emphasizes that very few languages have been targeted by previous work. For example, none of these studies examined a scripting language, such as Python or JavaScript, a functional programming language, e.g., Haskell or Elixir, or even more contemporary multiparadigm languages that receive strong influence from functional languages, e.g., Kotlin or Swift.

3.4 THREATS TO VALIDITY

In this section, we discuss the threats to the validity of this study.

Construct validity. Our study was built on the selected primary studies, which stem from the search and selection processes. We only used three search engines for our automatic search. Other engines, such as Springer and Google Scholar, could return different results. However, the majority of our seed studies were indexed by ACM and IEEE, and then we used Scopus to expand our search. Moreover, while searching on the ACM digital library, we used the *Guide to the Computing Literature*, which retrieves resources from other publishers, such as Springer. Finally, we avoided Google Scholar because it returned more than 17 thousand documents for our search string, and we did not have the resources to analyze so many papers.

Internal validity. This study was conducted by multiple researchers. We understand that this could pose a threat to its internal validity since each researcher has a certain knowledge and way of conducting her research activities. However, each researcher conducted her activities according to the established protocol, and periodic discussions were conducted between all researchers. Moreover, a few primary studies do not report in detail the tasks the subjects perform. Because of that, we might have misclassified these studies when mapping studies to task types. Finally, comparing different studies poses some threats due to the many differences in their setups, such as materials, programming languages, number of subjects, the subjects' expertise, and tasks to be performed by the subjects. It is not feasible to consider all differences in the studies' setups to compare the studies; some of them do not even provide fine-grained information, such as the size of source code snippets used in the experiments. In our study, however, we mitigated the threat of directly comparing studies by considering the activities performed by subjects in the experiments (which require different levels of cognitive skills), the dependent variables of the studies, the programming languages, the number of subjects, and statistical power.

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

3.5 CONCLUSION

In this work, through a systematic review of the literature, we investigated what formatting elements have been studied and which levels of formatting elements were found to have a positive impact on code legibility when compared to functionally equivalent ones in human-

centric studies. We present a comprehensive categorization of the elements and levels found. In addition, we analyzed the results considering the subjects, activities, dependent variables, and programming languages involved in the studies.

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

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

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

4 WHAT CODING ALTERNATIVES ARE MORE READABLE? A SYSTEMATIC LITERATURE REVIEW

In this chapter, we carried out a systematic literature review with the aim of identifying which semantic and structural elements were evaluated in empirical studies and which alternatives were found to be more readable for human subjects. We found 38 papers that contained human-centric studies, which directly compared alternative formatting elements. We then used a card-sorting technique to analyze and classify these formatting elements. This work aim to provide a comprehensive view of the existing knowledge within the literature about the impact of different code alternatives on code readability (goal #2).

4.1 METHODOLOGY

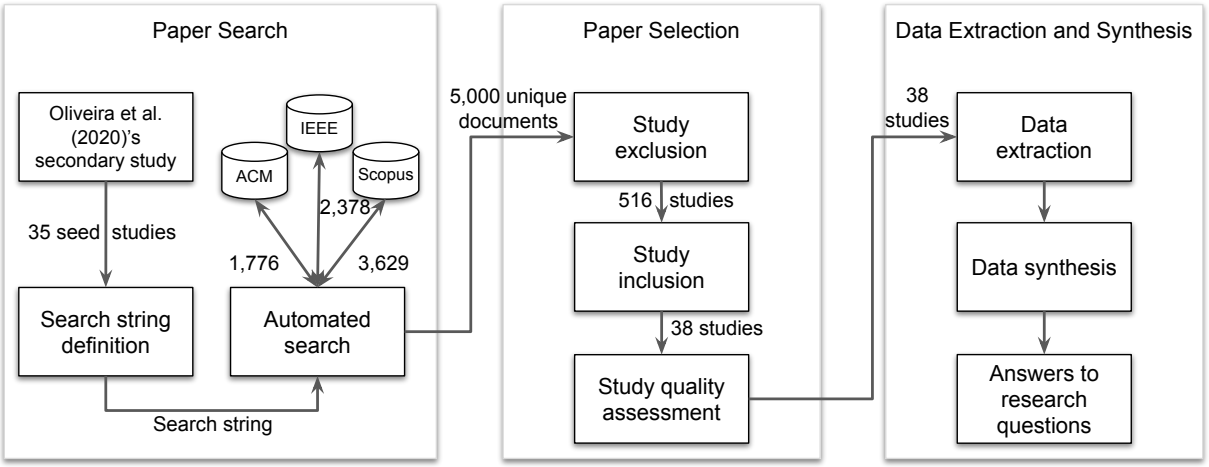


Figure 11 – Systematic literature review roadmap of the study about readability of code elements.

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

RQ1 What code elements have been investigated in human-centric studies aiming to compare the readability of these elements?

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

To answer our research questions, we conducted a systematic literature review designed following the guidelines proposed by Kitchenham, Budgen & Brereton(58). The process follows the same step described in the previous work (Section 3.1). Figure 11 presents the roadmap of this review, considering the different numbers of papers resulting in each step. In the following, we describe the summary of the process and the different settings adopted for this study.

4.1.1 Paper Search

Similarly to the previous study (Chapter 3), we selected 35 papers from the study described in Chapter 2 to be considered as seeds in this one. These papers contain studies that compare two or more ways of writing code by only changing code elements to help code understanding in terms of readability. We extracted relevant terms from the titles and keywords of the seed papers, however, for four papers ((54, 69, 50, 92)), we could not do so as the titles and keywords were either overly general or too specific. Using the gathered terms, we created the search string below. The words are constructed by taking the cartesian product (represented by x) between sets, and applying the union (represented by $+$) to the two resulting sets of words. This results in words such as *"code readability"*, *"program construct"*, *"coding style"*, and *"programming idiom"*.

$$Title(ANY(terms))\ OR\ Keywords(ANY(terms)), \text{ where } terms = \{$$
$$(\{\text{code, program}\} \text{ x } \{\text{comprehension, comprehensibility, understandability, understanding, misunderstanding, misunderstandings, readability, idiom, idioms, style, styles, convention, conventions, construct, constructs}\})$$
$$+$$
$$(\{\text{coding, programming}\} \text{ x } \{\text{idiom, idioms, style, styles, convention, conventions, practice, practices, construct, constructs}\})\ \}$$

In this study we also adapted our generic search string to perform an automatic search in the ACM Digital Library, IEEE Explore, and Scopus. We retrieved 1,776, 2,378, and 3,629 documents, respectively, on September 8, 2023. After removing duplicates, we were left with 5,000 unique documents. The seed papers that we used to build the string search were returned by the automatic search.

4.1.2 Paper Selection

Through the automated search, we obtained a total of 5,000 unique documents. We used the same exclusion criteria as a previous study, described in Section 2.1.2, because we were interested in the same format and rigor of primary studies. After applying the exclusion criteria, we were left with 516 papers.

After triage, we applied the following inclusion criteria to the 516 papers, resulting in a total of 38 papers. In addition to these, we also included the four seed papers that were not found through automatic searching, which brought the total number of papers to 38. Similar inclusion criteria to a previous study were adopted, described in section Section 2.1.3 regarding methodology and granularity. Criteria marked with an asterisk (*) were adjusted to fit the study's context. The list of inclusion criteria is presented below.

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

IC2 (Methodology). The study must be or contain a controlled experiment, quasi-experiment, or survey *involving human subjects*.

IC3 (Comparison)*. The study must directly compare alternative programming constructs, coding idioms, and semantic cues *in terms of code readability*, and the alternatives must be clearly identifiable.

IC4 (Granularity). The study must target fine-grained program elements and low-level/limited-scope programming activities. Not design or comments, but implementation.

We also conducted a final quality assessment of the selected papers using nine questions adapted from previous guidelines (64). These questions covered study design, analysis, and rigor, with each question receiving a score of yes (1), partially (0.5), or no (0). Papers needed to score at least 4.5 out of 9 to be retained. Each paper was evaluated by one author, with scores discussed among authors to ensure alignment. The assessed papers had scores ranging from 5 to 9, all above the 4.5 threshold, resulting in no papers being excluded due to low quality. The scores of the studies were: $min = 5$, $median = 8$, $max = 9$

4.1.3 Data Extraction and Synthesis

We conducted an analysis of a total of 38 papers to answer the research questions. The analysis involved two steps - data extraction and synthesis. The steps of this process was the same to that of our previous study, which is described in Section 3.1.3. The identified groups and factors are listed in Table 11 and detailed in the following sections.

Table 11 – Card Sorting results.

Groups (summary of the results)	Factors
Control Flow (Table 12)	Conditional Constructs and Styles
	Repetition Constructs and Idioms
Expressions (Table 13)	Boolean Expressions as Conditions
	Implicit Intent vs Not Implicit Intent
	Expressions with vs. without Side-effects
	Conflated vs. Decomposed Expressions
	Pointer and Arrays
Declarations (Table 14)	Lexical Order
	Preprocessor Usage
	No Modularization vs Types of Modularization
	Variable Usage
Identifiers and Names (Table 15, Table 16, Table 17)	Identifier Length
	Fully-qualified Names
	Meaningful Names vs Obfuscated Name
	Meaningful Name vs Another Meaningful
	Linguistic Antipatterns
	Linguistic Patterns
Miscellaneous (Table 18)	Typing
	Verbose or Unnecessary Code vs. Concise or Required Code
	Others

4.2 CONTROL FLOW

This sub-category comprises work performing comparisons aiming to evaluate the readability of constructs and idioms that impact the control flow of a program. This pertains to the usage of loops, conditional statements and expressions, recursion, and others. Arguably, conditional expressions could also be discussed in Section 4.3. The summary of results is presented in Table 12. In this section, we just describe the comparisons that presented statistical differences.

Table 12 – Summary of Control Flow.

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

Conditional Constructs and Styles

Gopstein et al. (25): ternary operators vs. `if` statements (C/C++)

Correctness (Trace): `if` statement is the best.

Medeiros et al. (18): ternary operators vs. `if` statements (C)

Opinion: ternary operator is neither negative nor positive. No statistical test was performed

Gopstein et al. (25): logical operators as control flow vs. control flow statements (C/C++)

Correctness (Trace): control flow statements statement is the best.

Medeiros et al. (18): logical operators as control flow vs. control flow statements (C)

Opinion: logical operator as control flow is neither negative nor positive. No statistical test was performed

Wiese et al. (19): multiple conditions with `&&` ('and' operator) vs. nested `if` statements (C/C++)

Correctness (Trace, Implement): No significant difference.

Opinion: multiple conditions with `&&` is the best.

Wiese et al. (84): multiple conditions with `&&` ('and' operator) vs. nested `if` statements (C/C++)

Correctness (Trace, Implement, Refactor): No significant difference.

Opinion: multiple conditions with `&&` is the best.

Oman and Cook (88): Multiple conditions in three styles (Pascal)

Correctness(Inspect): Sequential `if` statements (as opposed to nested ones) are better for readability.

Time (Inspect): Sequential `if` statements (as opposed to nested ones) are better for readability.

Opinion (Inspect): Sequential `if` statements (as opposed to nested ones) are better for readability.

Ajami et al. (42): `if` statements vs. `for` statements (Not specified)

Correctness (trace): `if` statement was better for readability.

Time (trace): `if` statement was better for readability.

Love (72): Binary conditionals vs. ternary conditionals (Fortran)

Correctness (Memorize): Binary conditionals were more readable for graduate students, but there was no difference for undergraduate students.

Boysen and Keller (67): `if` statements vs. `case` statements (Not specified)

Correctness (Trace): No difference was found.

Table 12 continued

Time (Trace): No difference was found.

Boysen and Keller (67): Nested `if` statements vs. sequential `if` statements vs. `case` statements (Not specified)

Correctness (Trace): No difference was found.

Time (Trace): No difference was found.

Wiese et al. (19): Multiple `if` statements vs. one `if` statement with multiple `else if` for exclusive cases* (Java)

Correctness (Implement and Trace): No difference was found.

Opinion: No difference was found.

Santos and Gerosa (16): Code block nesting with at most three levels vs. code block nesting with more than three levels (Java)

Opinion: No difference was found.

Repetition Constructs and Idioms

Iselin (78): Loop style read/process vs. process/read (COBOL)

Correctness (Trace): Read/process was more readable for students, but no difference was found for programmers.

Time (Trace): Read/process was more readable for students, but no difference was found for programmers.

Ajami et al. (42): Loop counting down vs. counting up (Not specified)

Correctness (Trace): Looping counting up was better for readability.

Time (Trace): Looping counting up was better for readability.

Benander et al. (54): Iteration vs. recursion (Pascal)

Correctness (Present): Recursion was better for readability.

Time (Present): Recursion was better for readability.

Maćkowiak et al. (79): Imperative vs. vectorized array operations in a visual language (Board Programming)

Correctness (Trace): Vectorized array operations were better for readability.

Time (Trace): Vectorized array operations were better for readability.

Wiese et al. (19): `for` statements vs. `while` statements (Java)

Correctness (Implement and Trace): No difference was found.

Opinion (Implement and Trace): No difference was found.

4.2.1 Conditional Constructs and Styles

Two papers evaluate the readability of code using the ternary/conditional expressions, e.g., a?b:c in C, C++, Java, vs. `if` statements. Gopstein et al. (25) carried out an experiment with 73 programmers with over 3 months of experience in C/C++ languages. This study presented the subjects with very small functionally-equivalent code snippets with alternative implementations, one with conditional expressions ("an atom of confusion") and the other using `if` statements. The subjects were asked to determine the output of each program (Trace). Then, they evaluated the correctness and the time required to answer. This study concluded that for both dependent variable the conditional operator is more confusing than the if statement (McNemar's test with Durkalski correction, p-value=$1.74e{-}05$, effect-size=0.36).

The work by Medeiros et al. (18) evaluates the impact "misunderstanding code patterns", an alternative name for atoms of confusion. The 97 subjects of this study provided their personal opinions using a 5-point Likert scale (Opinion) about snippets with and without the ternary operator. The majority of the subjects considers that the use of the ternary operator is neither negative nor positive. Only 8.25% of the subjects exhibited a negative perception of this operator. These results differ from the result obtained by Gopstein et al. (25). The authors of this study have also submitted 6 patches to real software projects suggesting the removal of the aforementioned "misunderstanding code pattern". Among these, two were accepted, two were rejected, and two were not answered by developers.

In the same papers cited above (25, 18), the authors evaluated the difference between logical operators and control flow statements. For example, for Gopstein et al. (25), the && and || operators are used for logical conjunction and disjunction, respectively. However, due to short circuit, they can also be used for conditional execution. The results of the paper by Gopstein et al. (25) show that using logical operators as control flow negatively influences code comprehension (McNemar's test with Durkalski, p-value=$5.62e-09$, effect-size=0.48). Medeiros et al. (18) concluded that using a logical operator to implement control flow has a high negative perception by developers (88.66%). The authors submitted one patch removing uses of this idiom but it was rejected by developers.

Wiese and colleagues (19) carried out a survey with 231 university students to compare "novice" and "expert" styles of conjoining conditions in `if` statements. The novice style consists of using nested `if` statements. The expert style consists of using the && operator. The authors asked the subjects to (i) write a small function based on a description (Implement), (ii)

provide an opinion about which style is more readable and which style is the best overall, and (iii) determine the outputs of functions (Trace). They assessed the accuracy of the responses and the subjects' opinions. The authors evaluated three hypotheses: H1) At least 20% of the population of students choose non-expert code as the most readable; H2) students are more likely to choose the expert code as being the best styled rather than being the most readable; and H3) students are more accurate at comprehension questions for novice code. They did not find a statistical difference (z-test with Bonferroni correction, p-value> 0.99) for H1, and 90% of the subjects agreed that the use of logical operators (&&) is more readable. Also, they did not find a statistical difference for H2 (McNemar's test with Bonferroni correction, p-value$= 0.86$, $x^2 = 0.2$), where students made the same choice for style and readability more than 80% of the time. Finally, they did not find evidence that the accuracy on the comprehension questions was affected by the use of expert or non-expert style (logistic regression, t(7287)$=0.379$, p-value$=0.70$), where the subjects answer correctly in 75% and 77% for functions with logical operators (&&) and nested `if` statements, respectively.

Wiese et al. (84) replicated their previous study (19) with a group of 32 students at a liberal arts college. The replication focuses on students who are earlier in their computer science courses and are less likely to be influenced by any style. The main differences from the prior study were that they added an editing task whose aim is to change a function (Refactor). They only performed statistical analysis for H1 (At least 20% of the population of students choose non-expert code). To compare styles of conjoining conditions in `if` statements, in the task to implement a function, they classified 14 students who wrote in expert style (using && operator), 2 students who wrote in novice style (using only nested `if` statements), and 2 students who wrote an incorrect function. In the editing task, they found that 13/18 students used the && operator. For the comprehension questions, they employed a function that would lead to an exception and a function that would not. They found out that for the first function, 6/18 and 1/18 answered correctly for && and for nested `if`, respectively. For the second function it was 12/18 and 13/18 for && and for nested `if`. There is no statistical difference (z-test with Bonferroni correction, p-value$= 0.9$) for H1. They found out that 11/18 and 8/18 of students think that && and nested `if` is the best style, respectively. In this experiment, the students were less likely to find the && operator more readable (61% here versus 90% in the previous experiment).

Oman and Cook (88) performed two experiments to compare different ways of writing nested `if` statements extracted from the Kernighan & Plauger's Elements of Programming

Style (113). The first of these experiments compared three styles of `if` statements in Pascal programs, realized as different versions of the same example program. Version 1 uses `if` statements that include nesting, have indentation, and the statement `then` has its own line. Version 2 is a reorganization of version 1 to include each statement `if` or `else` align in one line with the statement `then` and use of embedded spaces (regular white spaces). In version 3 the authors combine logical conditions, thus eliminating some nested blocks and organized one line of code per conditional case. This experiment involved 36 computer science students with junior experience levels. The authors asked the subjects to read a snippet, answer questions about the `if` statements (Inspect), and rate the indentation and structure of the code (Subject Opinion). Then, they evaluated the (i) score of the responses, (ii) the response time, and (iii) the rate provided in the questions involving opinion. This study concluded that there is a significant difference in time (ANOVA and MANOVA, $F = 4.99, p < 0.01, d.f. = 2.33$), where version 3 was better. They state that "the stylistic variations affect the comprehension". The results to score and opinion were not significant. The second experiment employed a similar configuration, but it was carried out with 33 students, and three different formats of `if` statements were used. Versions 1 and 2 of this experiment were analogous to versions 2 and 3 of the first one, respectively. Version 3 made the conditions of the `if` statements more explicit (and longer), therefore avoiding the use of `else` clauses and thus nesting. The results reveal that the subjects had a better result for version 3 (ANOVA and MANOVA, $F = 5.02, p < 0.01, d.f. = 2, 30$) and time (ANOVA and MANOVA, $F = 3.12, p < 0.05, d.f. = 2, 30$). On the other hand, the results to opinion show no significant difference.

Ajami et al. (42) carried out an experiment with 220 professionals to evaluate what the complexity to read the following control flow constructs: `if` and `for`. For this, the authors asked the subjects to answer questions about the code (Trace). They constructed three variants and evaluated the correctness of the answers and the time taken to reach a correct answer. This study concluded that `if` statements are easier to understand than `for` statements (Wilcoxon with Bonferroni correction test, p-value < 0.0016).

The work of Love (72) aim to investigate the effect of simple control flow and complex control flow for readability the program in Fortran. The simple control flow is composed by the use of `if` statement with `goto` (unconditional statement) where whether the condition is satisfied the flow of program jump to the indicate line, otherwise, the flow of program go to the next line. The complex control flow consists of a custom `if` statement that takes an arithmetic expression and depending on the result the program flow goes to one of the three

line numbers passed right after the expression. The result of the expression can be negative, zero, or positive. For this investigation, the authors requested that 31 students (i) memorize a program in 3 minutes and rewrite the program memorized in 4 minutes (Memorize) and (ii) describe the functionality of the program (Present). They assessed the correctness of the task. For the Memorize activity, they found out that simple flow control has better results for graduate students (ANOVA test, p-value < 0.05), but there is no difference for undergraduate students. They did not find a statistical difference (ANOVA, p-value $= 0.57$) for the description of program functionality (Present) for any of the groups.

4.2.2 Repetition Constructs and Idioms

In the control flow group, we can find comparisons about constructs and idioms for repetition/loops. For example, Iselin's paper (78) evaluated the effects of loop styles *read/process* and *process/read* on understandability. Iselin considered that "*in the read/process loop all the input was read at the beginning of the loop, and the test for loop termination was in the middle of the loop. In the process/read loop the test for loop termination was at the beginning of the loop, the first input was read prior to entering the loop, and all subsequent input was read at the end of the loop*" (78). The author asked 168 students and programmers to answer questions about the code (Trace) in two trials. She rated the (i) tracking time, (ii) the number of errors in the output, and (iii) the magnitude of the error in the output variable. In the statistical tests the author considered $\alpha = 0.15$. For $\alpha = 0.05$, she did not find a statistically significant result on trace time (ANOVA, p-value $= 0.12$), number of errors (MANOVA, p-value $= 0.131$) or the magnitude of the error, on average (Univariate F-test, $\alpha = 0.15$, p-value $= 0.062$) for students or programmers.

The work of Ajami et al. (42), previously mentioned in Section 4.2.1, investigated the effect of the loop counting down and counting up extracted on understandability. For this comparison, the authors asked 220 professionals to answer questions about the code (Trace) when examining equivalent versions of programs with loops counting up and down. They evaluated the time to answer the questions and their correctness. The authors found out a statistically significant difference (Wilcoxon test with Bonferroni correction, p-value < 0.0016), where the loop counting up was easier to understand.

The paper by Benander et al. (54) compares iteration and recursion. The authors divided 275 students into three groups and asked them to explain functions (Present) in Pascal.

The functions were: (task 1) search a value in a linked list, and (task 2) make a copy of a linked list. The all three groups received a task with each function in different days. They assessed the correctness and timing of responses. The authors found out that for task 1 there was a statistically significant difference for correctness (Chi-Squared test, $\chi^2 = 7.487$, p-value$= 0.006$) where the recursive function was the best. For correctness in task 2 , the results were not statistically significant (Chi-Squared test, $\chi^2 = 1.795$, p-value$= 0.180$), but overall the iterative version had 51.4% (38/74) correct answers while recursive version had 40.0% (26/65). When considered the time variable, the authors did not find a significant difference (t-test, p-value$= 0.96$) for task 1, except for a specific group (t-test, p-value$= 0.036$), where the recursive pattern was the best. For task 2 the recursive pattern was also better for a specific group (t-test, p-value$= 0.033$) and also in the aggregated data (t-test, p-value$= 0.045$). As general conclusions, although differences in significance were found, the recursive pattern was better ($p = 0.006$).

The paper by Maćkowiak et al. (79) evaluated the understandability of a graphical programming notation for end-users (Board Programming). It compared, among others, programming constructs for imperative and data-driven iteration. The authors separated 70 students in two groups: control group (imperative) and experimental group (data-driven). Each group received one version of the source code for a polynomial function (a function which returns the value of a polynomial when the array of co-efficients and the value of the variable are given), a set of examples, and input data. The participants had to determine the output of the function (Trace). The authors measured (i) whether the subjects failed in the first attempt, (ii) the mean of the number of attempts, (iii) whether the subject completed the task, and (iv) the mean attempt time. They found out a significant difference for time (Mann–Whitney–Wilcoxon test, Cliff's $\delta = 0.858$, p-value< 0.001), for the cancellation rate (Fisher exact test, p-value< 0.001), and first attempt failed (Fisher exact test, p-value$= 0.001$) where the experimental group was faster and committed less mistakes. They did not find a statistically significant difference for the number of attempts.

4.3 EXPRESSIONS

In this group we have studies that compare different ways to write expressions in programs. This pertains to the usage of boolean expressions, literal encoding, conflated expressions, and others. The summary of results is presented in Table 13. In this section, we just describe the

Table 13 – Summary of Expressions.

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

Boolean Expressions as Conditions

Iselin (78): Equality operators and boolean expressions resulting in true and false (Cobol)

Correctness (Trace): Expressions with positive equality operator were more readable for students. However, for other evaluations, there was no statistical difference for students or programmers.

Time (Trace): Expressions with positive equality operator were more readable for students. However, for other evaluations, there was no statistical difference for students or programmers.

Boysen and Keller (67): Expressions resulting true vs. false (Not specified)

Correctness (Trace): Expressions resulting in true were better for readability, except in expressions involving negation operators (NE, NOT) where there was no significant difference.

Time (Trace): Expressions resulting in true were better for readability, except in expressions involving negation operators (NE, NOT) where there was no significant difference.

Iselin (78): Boolean operators (Cobol)

Correctness (Trace): Students were quicker to perform the activity with operator equal (cf. not equal), but no difference was found in terms of correctness.

Time (Trace): Students were quicker to perform the activity with operator equal (cf. not equal), but no difference was found in terms of correctness.

Boysen and Keller (67): Boolean operators (Not specified)

Correctness (Trace): There was a statistical difference on reaction time of correct answers where the processing time for *EQUAL* was smaller than *NOT EQUAL*.

Time (Trace): There was a statistical difference on reaction time of correct answers where the processing time for *EQUAL* was smaller than *NOT EQUAL*.

Ajami et al. (42): Negation expression (Not specified)

Correctness (Trace): Expressions with four conjunctions where two are negated was better for readability than other negation expressions.

Time (Trace): Expressions with four conjunctions where two are negated was better for readability than other negation expressions.

Iselin (78): Expressions resulting true vs. false (Cobol)

Correctness (Trace): No difference was found.

Time (Trace): No difference was found.

Table 13 continued

Implicit Intent vs Not Implicit Intent
Gopstein et al. (25): Change of literal encoding vs. use the literal encoding directly (C/C++)
Correctness (Trace): Use the literal encoding directly was better for readability.
Gopstein et al. (25): Implicit predicate vs. explicit predicate (C/C++)
Correctness (Trace): Explicit predicate was better for readability.
Gopstein et al. (25): Use vs. omission of parentheses for operator precedence (C/C++)
Correctness (Trace): Use parentheses for operator precedence was better for readability.
Medeiros et al. (18): Use vs. omission of parentheses for operator precedence (C)
Opinion: Use parentheses for operator precedence was better for readability. No statistical test was performed.
Gopstein et al. (25): Arithmetic as logic (C/C++)
Correctness (Trace): No difference was found.
Expressions with vs. without Side-effects
Dolado et al. (24): Pre/pos-increment vs. separated operations (C)
Correctness (Trace): Separated operations was better for readability.
Time (Trace): Separated operations was better for readability.
Gopstein et al. (25): Pre/pos-increment vs. separated operations (C/C++)
Correctness (Trace): Separated operations was better for readability.
Medeiros et al. (18): Pre/pos-increment vs. separated operations (C)
Opinion: Pre/pos-increment did not perceived as negative for readability by subjects. No statistical test was performed.
Gopstein et al. (25): Assignment as value vs. separated assignment statements (C/C++)
Correctness (Trace): Separated assignment statements was better for readability.
Medeiros et al. (18): Assignment as value vs. separated assignment statements (C)
Opinion: No difference was found. No statistical test was performed.
Conflated vs. Decomposed Expressions
Gopstein et al. (25): Comma operator vs. separated statement (C/C++)
Correctness (Trace): Separated statement was better for readability.
Medeiros et al. (18): Comma operator vs. separated statement (C)
Opinion: Comma operator had a high negative perception by developers. No statistical test was performed.
Santos and Gerosa (16): Direct vs. indirect references to sub-properties (Java)

Table 13 continued

Opinion: Indirect references to sub-properties were better for readability.
Santos and Gerosa (16): Monolithic expressions vs. multiple shorter expressions (Java)
Opinion: No difference was found.
Pointer and Arrays
Gopstein et al. (25): Reversed subscripts vs. access through index (C/C++)
Correctness (Trace): Access through index were best for readability.
Medeiros et al. (18): Reversed subscripts vs. access through index (C)
Opinion: Reversed subscripts had a high negative perception by developers. No statistical test was performed.
Maćkowiak et al. (79): Indices vs. heads in a visual language (Board Programming)
Correctness (Trace): Heads were better for readability.
Time (Trace): Heads were better for readability.
Attempt (Trace): No difference was found.
Medeiros et al. (18): Indexing vs. pointer arithmetic (C)
Opinion: No difference was found. No statistical test was performed.
Gopstein et al. (25): Indexing vs. pointer arithmetic (C/C++)
Correctness (Trace): No difference was found.

comparisons that presented statistical differences.

4.3.1 Boolean Expressions as Conditions

This group brings together all the comparison referring to ways of writing boolean expressions. For example, the paper of Iselin (78), mentioned previously in Section 4.2.2, to evaluate the effects of boolean expressions on program comprehension. In this study the author considered positive expressions (a == b), negative expressions (a != b), expressions that return true (1 < 2), and false (1 > 2). The first comparisons were (i) positive true vs positive false and (ii) negative true vs negative false. This study follows the same methodology of the study of loops. We also considered report as significant only results where p-value< 0.05. The author had considered $\alpha = 0.15$. She found out a statistical difference on trace time (ANOVA, p-value < 0.05) for the students only in trial 1, where positive conditions was the best. There is no statistical difference on error data for students and on trace time and error data for

programmers. Iselin also evaluated the positive/negative conditions resulting in true/false values. She did not find statistically significance for *positive/true* $<$ *positive/false*, *positive/false* $<$ *negative/false*, *negative/false* $<$ *negative/true* and *positive/true* $<$ *negative/true* on trace time for programmers.

Also, Boysen and Keller (67) investigated the effect on code comprehension of expressions resulting in true and false. The authors conducted a experiment where they asked 49 professionals and students to answer questions about the result of expressions that were presented (Trace) and calculated the reaction time and the accuracy of the responses. The authors of this study measured the reaction time of the correct answers and concluded that subjects have a better performance when analyzing code including `if` statements (the paper employed a repeated measures ANOVA with Duncan's multiple range post-hoc test). The authors measured the reaction time (time while looking at the program) and the correctness. They verified whether there is a significant difference between the versions in reaction time when the subjects successfully produce a correct response. They found out that expressions resulting in true are processed more quickly, but there is no significant difference in expressions involving the operators *NE* and *NOT*.

Considering the effect of boolean operators positive(*EQUAL*)/negative(*NOT EQUAL*) on code comprehension, Iselin (78) found out a statistical significance (ANOVA, p-value< 0.05) for trace time where the positive expressions were the best between the students. She did not find statistical difference for the error data variables. Boysen and Keller (67) studied the use of operators *EQUAL* and *NOT EQUAL* (they are related to positive/negative expressions). They found out a statistical difference on reaction time of correct answers where the processing time for *EQUAL* was smaller than *NOT EQUAL*.

The paper by Ajami et al. (42), previously mentioned in Section 4.2.1, investigated the use of negation on boolean expressions. They presented three versions of denial expressions: (i) *an* is a expressions with four conjunctions where two are negated; (ii) *an1* is the inversion of the *an*; and *an2* is obtained by applying De Morgan's law to *an1*. The authors found out a statistical difference for the time of correct answers where the version *an* was better than *an1* and *an2* (Wilcoxon test, $p < 0.001$). They not find a statistical significance between *an1* and *an2*. The authors claim that "some but not all uses of negation are more difficult: negations are different from one to the other".

4.3.2 Implicit Intent vs Not Implicit Intent

The paper by Gopstein et al. (25) investigated whether *Change of Literal Encoding* (e.g., `printf("%d",01E)`) instead using the literal encoding directly (e.g., `printf("%d",30)`) affects the code comprehension. The details of the experiment was explained in Section 4.2.1. They found out a statistical significance (McNemar's test, $p = 2.93e-14$, effect-size$= 0.63$) for correctness that suggest using a different encoding of result presentation decrease the understanding of the code.

Gopstein et al. (25) also investigated whether *Implicit Predicate* causes misunderstanding. It is possible when the language compiler/interpreter can infer numerical values (e.g., 1 and 0) as boolean values (true and false). Then, the statement `if(4 % 2 != 0)` is equivalent to `if(4 % 2)`. They found out a significant statistical (McNemar's test, $p = 4.27e-03$, effect-size$= 0.24$) for correctness where implicit a predicate cause confusion in code comprehension. Similarly, they also evaluated the using of *Arithmetic as Logic* (e.g., `(V1-3) * (V2-4)`) instead of logical operators (e.g., `(V1-3) && (V2-4)`). The authors did not found a statistical significance (McNemar's test, $p = 0.248$, effect-size$= 0.1$) in correctness. It means that use arithmetic as logic do not decrease readability.

Finally, two papers assessed whether the complexity of precedence rules cause confusion. In those studies, they compared the same expression where the alternative used *Operator Precedence* (e.g., parentheses) to explicit the precedence. Gopstein et al. (25) found out a statistical significance (McNemar's test, $p = 5.90e-05$, effect-size$= 0.33$) where not using of operator precedence decrease the correctness of the subject's answers. In another paper, Medeiros et al. (18) asked the subjects' opinion. The details of the methodology is in section 4.2.1. In the first study, they also found out that not using operator precedence is perceived as negative for understanding by 80.4% of the developers. In the second study, they submitted 5 pull requests to remove this idiom. Among these, two were accepted, two were rejected, and one was not answered by developers.

4.3.3 Expressions with vs. without Side-effects

This group aggregate the studies that evaluated code comprehension in expressions with side-effects and their alternatives. For example, the paper by Dolado et al. (24) assessed whether pre/pos-increment expressions cause misunderstanding. They prepared two set of

questions where each one had a version with side-effect (SE) and another side-effect-free (SEF). The authors asked students to answer the two set of questions (named Test 1 and Test 2, respectively) and programmers answer only one set of questions (named Test 3). In each Test, the subjects was divided in two groups and each one work on a version of the questions and after on the another version. How the students participated of two periods of test, they used the latin-square layout. The authors evaluated the correctness and the response time of the tests. They found out a statistical difference for correctness in Test 1, Test 2, and Test 3 (for all tests, ANOVA, $p = 0.00$) where SEF was better. Also, for time variable there was a significant difference in Test 1, Test 2, and Test 3 (for all tests, ANOVA, $p \leq 0.018$) where the SEF was the best for comprehension. They also evaluated the learning effect and found out it happens for time variable.

Gopstein et al. (25) studied two types of side effect instructions: post-increment/decrement (e.g., `i++`) and pre-increment/decrement (e.g., `++i`). The details of the experiment was explained in Section 4.2.1. They found out a statistical significance in correctness for post-increment/decrement (McNemar's test, $p = 6.98e-08$, effect-size$= 0.45$) and pre-increment/decrement (McNemar's test, $p = 6.89e-04$, effect-size$= 0.28$) where the use of those statements with side-effect cause confusion for code comprehension. Medeiros et al. (18) also evaluated the same side-effect statements. The methodology was detailed in Section 4.2.1. In the first study, differently for Gopstein et al. (25), they found out that the subjects did not perceive post-increment/decrement and pre-increment/decrement as negative for understanding. Only 31.96% and 21.65% of the developers thinks that those statements are negative for understanding. In the second study, they submitted 2 patches to remove this idiom, where one was accepted and one was rejected by developers.

Gopstein et al. (25) also investigated the pattern *assignment as value* (e.g., `V1 = V2 = 3`) and found out a statistical difference (McNemar's test, $p = 3.78e-10$, effect-size$= 0.52$) where the alternative code, separated assignment, is better to code understanding.

4.3.4 Conflated vs. Decomposed Expressions

In this group we describe studies that evaluated the code comprehension when combined series of a sequence of computations combined instead of that sequence separated. For example, the paper by Gopstein et al. (25) investigated whether the using of *comma operator* to combine a sequence of computations (e.g., `V1 += 1; V3 = V1;`) in one expression (e.g.,

`V3 = (V1 += 1, V1);`) causes confusion for developers. The methodology was explained in the Section 4.2.1. They found out a statistical significance (McNemar's test, $p = 2.46e-04$, effect-size$= 0.30$) that implies the combination of computations in one expression is confuse to developers and separated operations is the best option. Medeiros et al. (18) also assessed the same comparison. We described the methodology in Section 4.2.1. Medeiros et al. (18) also concluded that combine a sequence of computations is tough for understand, where 100% of subjects perceived this pattern negative for comprehension. They did not evaluate this pattern in the second study.

The paper by Santos et al. (16), carried out a study to assess the impact of a set of Java coding practices on the understandability perceived by developers. They studied whether frequent calls to sub-properties of class member properties should be made storing a reference to that sub-property, avoiding multiple statements containing long chains of objects and sub-properties. For this investigation, the authors applied an opinion survey with 62 students and professionals. They asked the subjects to provide an opinion on the code (Opinion). They found out a statistical difference (two-tailed test for proportion, $p = 0.000$) where store a reference to the sub-property and use it is better than using a long chain of references.

4.3.5 Pointer and Arrays

In this section we describe studies that evaluated alternatives to write instructions with arrays and pointers. For example, Gopstein et al. (25) and Medeiros et al (18) assessed an idiom that use reversed subscript operation to access elements in array (e.g., `1["abc"]`) and compared with access by index (e.g., `"abc"[1]`). The methodology of both studies was described in Section 4.2.1. Gopstein et al. (25) concluded there is a statistical significance (McNemar's test, $p = 1.52e-06$, effect-size$= 0.40$) where the using of reversed subscript operation in arrays decrease the code readability. In the first study by Medeiros et al. (18) also concluded that reversed subscript operation causes misunderstanding for developers, where 91.75% of subjects perceived this idiom as negative to readability. They did not evaluated this idiom in the second study.

The paper by Mackowiak et al. (79) evaluated the code comprehension in the using of heads instead of index to access elements of an array (or matrix). The head is a pointer for a element of array that move together with the other heads. For example, the head `H_Current` points to the second element of array and `H_Previous` points to the first element. When

`H_Current` moves to the third element, `H_Previous` moves to the second element. Therefore, the developer can using heads to transverse the array and compute referencing relative parts of array. They used twenty-eight students (13 in the control group and 15 in the experimental group). The subjects had to answers the output of a function which calculates the Levenshtein distance between two words. The authors found out a statistical difference for time (Mann–Whitney–Wilcoxon test, Cliff's $\delta = 0.456$, p$= 0.021$), for the cancellation rate (Fisher exact test, p-value< 0.005), and first attempt failed (Fisher exact test, p-value$= 0.013$) where the experimental group (who used the head version) was faster and less failed-prone. They did not find a statistical difference to number of attempts.

4.4 DECLARATIONS

In this subcategory is possible to find groups that address aspects related to different ways of writing statements.

4.4.1 Lexical Order

Geffen and Maoz (77) investigate the effect of ordering methods in understanding the code. They compared four strategies to order methods in a class: (i) StyleCop (SC) that has specific rules for ordering based hierarchically on the access of methods and non-access modifiers; (ii) Calling (CL) requires that when a method invokes another method, the invoking method should come before the invoked method; (iii) Connectivity (CT) advocate that related methods should be close to one another; and (iv) Calling+Connectivity (CLCT) mix the rules of CL and CT. The authors selected 5 classes from open-source Java projects where each one used an ordering strategy studied more random ordering. They created versions of the others strategies for each selected class. Then, they asked 60 engineers to answer questions about these classes (Trace) and carry out a modification in the provided code snippet (Trace + Implement). The authors assessed correctness and response time. The results do not show correlation between method ordering and correctness. For the Trace activity, results for the CLCT had the least percentage of wrong answers. Still, for Trace activity, CLCT and CL requests reduce the time to understand (at the 25th percentile). But, for the modification tasks, the results were inconclusive.

Table 14 – Summary of Declarations.

Factor – Study: Treatment (Programming Language) – Dependent Variables (Activities): Results
Lexical Order
Geffen and Maoz (77): Method ordering policies (Java)
Correctness (Trace and Implement): No difference was found.
Time (Trace and Implement): Calling (CL) and Calling+Connectivity (CLCT) method was better for readability. The results were inconclusive for the implement activity. No statistical test was performed.
Preprocessor Usage
Malaquias et al. (91): Disciplined vs. undisciplined preprocessor-based annotations (C/C++)
Opinion: Disciplined annotations had high acceptability as the best alternative for readability.
Correctness (Inspect + Implement, Debug): Disciplined annotations were better for readability.
Time (Inspect + Implement, Debug): Disciplined annotations were better for readability.
Schulze et al. (80): Disciplined vs. undisciplined preprocessor-based annotations (C)
Correctness (Inspect, Debug, Adapt + Inspect + Trace): No difference was found.
Time (Inspect, Debug, Adapt + Inspect + Trace): Disciplined annotations were better for the debug activity.
Gopstein et al. (25): Disciplined vs. undisciplined preprocessor-based annotations (C/C++)
Correctness (Trace): Disciplined annotations were better for readability.
Gopstein et al. (25): Macro operator precedence (C/C++)
Correctness (Trace): Define macro references separated from code statements and expressions was better for readability.
No Modularization vs Types of Modularization
Woodfield et al. (75): Monolithic vs. functional vs. super vs. abstract data type modularization (Fortran)
Correctness (Trace, Present, Inspect): Abstract data type modularization was better for readability.
Tenny (87): None vs. internal vs. external modularization (PL/I)
Correctness (Inspect, Trace, Recognize): No difference was found.
Variable Usage
Gopstein et al. (25): Repurposed variables vs. unique purposed variables (C/C++)
Correctness (Trace): Unique purpose variables were better for readability.
Maćkowiak et al. (79): Single assignment vs. re-assignments in a visual language (Board Programming)
Correctness (Trace): Single assignment was better for readability.
Time (Trace): Single assignment was better for readability.
Attempt (Trace): Single assignment was better for readability.
Avidan and Feitelson (51): Parameters vs. local variables (Java)
Correctness (Present): No statistical difference was found.
Time (Present): No statistical difference was found.
Opinion: Parameters were better for readability.
Gopstein et al. (25): Constant variables vs. literals (C/C++)
Correctness (Trace): No statistical difference was found.

4.4.2 Preprocessor Usage

In this section we present the studies related to the alternatives to use preprocessor in source code. Preprocessor is a language-independent tool used to implement variable software systems using conditional compilation. The paper by Malaquias et al. (91) evaluated the difference to code comprehension between disciplined annotations for preprocessor and undisciplined annotations. While disciplined annotations is aligned with the structure of the source code, undisciplined annotations emcompasses only parts of syntactical unit. Malaquias et al. carried out two studies. In the first, the authors selected undisciplined annotations in real projects, then submitted patches that change them in disciplined annotations. They assessed the acceptability rate of developers (Opinion). From 110 patches submitted, 99 were answered, and 62 were accepted. In the second study, the authors ask for 64 undergraduate students to change parts of the source code to fix a bug (Debug) or introduce a new feature variant (Inspect + Implement). They measured the time and the number of attempts to conclude all tasks. They found out there is a statistical difference for time (ANOVA, $p = 0.004$) and attempts (ANOVA, $p = 0.0001$) where the use of undisciplined annotations increases the time and it is more error-prone when performing the tasks in preprocessor-based systems.

Schulze et al. (80), also conducted an experiment to investigate the effect of preprocessor annotation types on program understanding. The authors asked 19 undergraduate to answer questions about the characteristics of the code (Inspect), to change a part of the source code (Adapt + Inspect + Trace) and find and correct bugs in the code (Debug). They assessed the time and correctness for the tasks. They did not find a statistical difference for correctness (Chi-squared test, $p > 0.247$) in any task. Also, for time, they did not find a statistical difference (t test, $p > 0.068$) except for one task of seven (t test, $p < 0.05$) where disciplined preprocessor were faster.

Gopstein et al. (25) evaluated the undisciplined preprocessor as an atom of confusion where its alternative is the disciplined preprocessor. They named it as *Preprocessor in Statement*. The methodology of this paper was detailed in Section 4.2.1. They found out a statistical difference (McNemar's test, $p = 8.53e - 11$, effect-size$= 0.54$) for correctness where using of undisciplined preprocessor causes misunderstanding for developers. The authors also studied the using of macro operators to define macro references (e.g., `#define M1 64-1`) in the middle of code statements and expressions instead of separate from them. The result for this element was similar to undisciplined annotations preprocessor result. They found out a

statistical difference (McNemar's test, $p = 1.77e-07$, effect-size$= 0.53$) for correctness where that using of macro operators cause confusion for code understanding.

4.4.3 No Modularization vs Types of Modularization

In this section we analyze studies that compared different alternatives to split the source code in modules. Woodfield et al. (75) compared four types of modularization with and without comments and investigated how they impact program understanding. The types of modularization were: (i) monolithic: no modularization; (ii) functional modularization: each natural logical module is in a separate physical module; (iii) super modularization: very small physical modules, five or ten lines of code each; and (iv) abstract data type modularization: that follows the functional modularization type. For the analysis in this section, we consider only the evaluation of the modularization without comments. The authors conducted an experiment with 48 undergraduate and graduate students where they asked the subjects to answer questions about the characteristics of the code (Trace and Inspect) and explain what the code does (Present). They evaluated the correctness of the questions and they did not find statistical difference between the monolithic, functional, or super modularization versions (ANOVA, $p = 0.464$). However, they found a statistical difference (ANOVA, $p = 0.056$) between abstract data type modularization and the next better, monolithic. However, the alpha that we consider significant is $p > 0.05$.

4.4.4 Variable Usage

In this section we describe studies that evaluated the effect in code comprehension for alternatives to the using of variables, constants and parameters. Gopstein et al. (25) compared adopt repurposed variable, the same variable is used in different context of the code, instead of unique purposed variable. The methodology was detailed in Section 4.2.1. They found out a statistical difference (McNemar's test, $p = 6.66e - 03$, effect-size$= 0.22$) for correctness where the using of repurposed variable causes misunderstanding.

Mackowiak et al. (79) investigated whether the using of single assignment of variable (immutability) is better for code readability than re-assignment variables. We described the methodology in Section 4.2.2. This study concluded using the Mann Whitney test that programs based on a single assignment with exemplary calculations seem to be easier to under-

stand. In this experiment, the 44 students had to answer the output of a source code that computed the value of a polynomial. The participants were divided in two groups, experimental group (single assignment) and control group(re-assignment). The authors found out a statistical difference for time (Mann–Whitney–Wilcoxon test, Cliff's $\delta = 0.528$, p$= 0.001$), for number of attempts (Mann–Whitney–Wilcoxon test, Cliff's $\delta = 0.53$, p$= 0.001$), and first attempt failed (Fisher exact test, p-value$= 0.005$) where the experimental group (who used single assignment) was faster and less failed-prone. They did not find a statistical difference to the cancellation rate.

Avidan and Feitelson (51) assessed the impact of local variables and parameter names in a method for code readability. In this direction, they selected six different methods from open-source Java project and masked their method name (e.g., replace by 'xxx'). Then, they prepared four version of each one: (i) all variables names remained intact; (ii) only parameters names were replaced by single letters; (iii) only local variables names were replaced by single letters; and (iv) both parameters and local variables names were replaces by single letters. The authors conducted an experiment with 9 developers where the subjects had to explain what the method does (Present) and provide a opinion about what makes the code readable (Subject Opinion). The subjects participated of experiment sessions for each method. They were divided in control group, that received the version (i) of method, and the experimental group, that received one of the others versions. They measured the time and correctness of answers. They found out that for three functions there was a statistical difference (Mann-Whitney test, $p < 0.05$) between control group (meaningful parameters and variables names) and experimental group (masked parameters and variables names) for time of correct answers, where the control group was faster than experimental. It means that for these methods the name of parameters and local variables help to understand the code. However, for the other three methods there was a small number of correct answers or they did not find a statistical difference. When the subjects were questioned what help more to understand the code, in 79% of the cases subjects thinks that parameters contribute more.

4.5 IDENTIFIERS AND NAMES

This subcategory aggregate all comparisons related to code elements names, such as variable names and language keywords. The summary of results is presented in Table 15. In this section, we just describe the comparisons that presented statistical differences.

Table 15 – Summary of Identifiers and Names.

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

Identifier Length

Lawrie et al. (47): Long vs. short meaningful identifier names (number of syllables is considered for length) (C, C++, and Java)

Correctness (Present, Memorize): Short meaningful identifier names were better for readability.

Time (Present, Memorize): No difference was found.

Opinion: No difference was found.

Hofmeister et al. (41): abbreviations vs. word (C#)

Correctness (Debug): Word identifiers were better for readability.

Time (Debug): Word identifiers were better for readability.

Visual Metrics (Debug): Word identifiers were better for readability.

Schankin et al. (53): Short single-word vs. long descriptive compound (Java)

Correctness (Debug): Descriptive name identifiers were better for readability.

Time (Debug): No difference was found.

Visual Metrics (Debug): Descriptive name identifiers were better for readability.

Scanniello et al. (50): Abbreviations vs. full words (C)

Debug (Correctness, Time): No difference was found.

Lawrie et al. (47): Abbreviations vs. full words(C, C++, and Java)

Correctness (Present, Memorize): No difference was found.

Time (Present, Memorize): No difference was found.

Opinion: No difference was found.

Fully-qualified Names

Santos et al. (16): Fully-qualified names vs. imports (Java)

Opinion: Import clauses were better for readability.

Binkley et al. (48): Long vs. short fully-qualified names (Java)

Correctness (Memorize, Recognize): No difference was found.

Time (Memorize, Recognize): Short fully-qualified names were better for readability.

Meaningful Names vs Obfuscated Name

Ceccato et al. (49): Clear vs. obfuscated names (Java)

Correctness (Inspect, Trace+Implement): Clear names were better for readability.

Time (Inspect, Trace+Implement): Clear names were better for readability.

Table 15 continued

Opinion: Clear names were better for readability.

Teasley et al. (45): Meaningful vs. nonsense names (Pascal)

Correctness (Trace): Meaningful names were better considering the functional level of comprehension.

Chaudhary et al. (68): Two random letter composed names vs. meaningful names (Fortran)

Correctness (Present, Present+Implement): Meaningful names were better for readability.

Avidan et al. (51): Three random letter composed names vs. original names (Java)

Correctness (Present): Original names were better for readability.

Time (Present): Original names were better for readability.

Hofmeister et al. (41): Single letter vs. full words (C#)

Correctness (Debug): Full word identifiers were better for readability.

Time (Debug): Full word identifiers were better for readability.

Visual Metrics (Debug): Full word identifiers were better for readability.

Hofmeister et al. (41): Single letter vs. abbreviations (C#)

Correctness (Debug): No difference was found.

Time (Debug): No difference was found.

Visual Metrics (Debug): Abbreviations identifiers were better for readability.

Lawrie et al. (47): Single letter vs. full words (C, C++, and Java)

Correctness (Present, Memorize): Full word identifiers were better for readability.

Time (Present, Memorize): Full word identifiers were better for readability.

Opinion: Full word identifiers were better for readability.

Beniamini et al. (52): Single letter vs. full words (Java, C)

Correctness (Debug, Present, Trace+Implement): No difference was found.

Time (Debug, Present, Trace+Implement): No difference was found.

Opinion: No difference was found.

Meaningful Name vs Another Meaningful

Santos et al. (16): Dictionary vs. invented names (Java)

Opinion: Dictionary names were better for readability.

Blinman and Cockburn (46): Full-descriptive vs. non-descriptive naming for framework interfaces (functions) (J#)

Correctness (Relate): Full-descriptive naming was better for readability.

Time (Relate): Full-descriptive naming was better for readability.

Table 15 continued

Linguistic Patterns
Wiedenbeck et al. (73): Beacons vs. non-beacons (Pascal)
Correctness (Memorize, Present): Beacons were better for readability.
Wiedenbeck et al. (74): Beacons vs. other structures with beacons (Pascal)
Correctness (Memorize, Present): Beacons were better for readability.
Chaudhary and Sahasrabuddhe (68): Purposeful vs. nonsensical names (Fortran)
Correctness (Present, Present+Implement): Purposeful names were better for readability.
Binkley et al. (48): Identifiers with memory ties vs. identifiers without memory ties (Java)
Memorize (Correctness): Identifiers with ties were better for readability.
Siegmund et al. (81): Beacons vs. non-beacons (Java)
Brain Metrics (Relate): No difference was found.

4.5.1 Identifier Length

In this section we described studies that evaluated how the length of identifiers affects code comprehension for meaningful names. For example, Lawrie et al. (47) evaluated the impact of three levels of identifier length: full words, abbreviations, and single letters. They executed an experiment with 192 students and professionals, where 80 of these completed the survey. The participants had to read twelve code snippets in C, C++, or Java. For each code snippet, they were asked to write the purpose of the code (Present), the confidence in that answer (Subject Opinion), and select identifiers that they recalled in code between a list of six candidates (Memorize). Lawrie et al. measured three response variables: (i) description rating - the correctness of the description of the purpose of code on a scale between 0 to 5, (ii) confidence - how much the participants are confident that understanding the code on a scale between 1 to 5, and (iii) PCIS (percent correct in source) - the percent of correct answers for identifiers that appeared in the code. Additionally, they measured the time for each task. The author compared the the effect of long and short meaningful identifier for code comprehension. They measured the PCIS (Memorize) and computed the number of syllables for each identifier measured. They found out a statistical significance (ANOVA, $p < 0.0001$) between PCIS and number of syllables where short meaningful identifier were the best. Furthermore, they found that for each additional syllable present in the set of identifiers, on average PCIS decreases by 1.4%.

Hofmeister et al. (41) evaluated whether the length and semantics of the identifier name affect the performance of developers during program comprehension They compared three styles of identifier name: word, abbreviation, and meaningless single letter. Similar to Scanniello and Risi (50), they asked participants to find defects (Debug) in code snippets written in C#. The authors recruited 72 developers who participated of a experiment with two steps: (i) they had to find semantic defects on three snippets, and (ii) they had to find syntax defects on three snippets. Each snippet had three versions, in which the identifier names were changed to either words, abbreviations, or meaningless single letters. Also, the snippets had one version with a semantic defect and another with a syntax error. The authors measured the time and correctness to find a defect and the visual attention during the experiment. For visual attention, they used an implementation of the restricted focus viewer (114), here named letterbox, where the code snippet is divided into frames (or area of interest), and the participants could shift up and down using the arrow keys. It was considered three AOI: comment, pre-defect, and defect. They measured three metrics. First, dwell time is the total time in each AOI (area of interest). Second, first pass reading time (FPRT) is the time it took participants to navigate the letterbox from entering the actual code to the position where the defect first appeared in the letterbox. And third, AOI visits is the frequency that one AOI was visited. In this section, we are interested in the results of abbreviated names and words versions. They found out a statistical difference in finding defects/time (ANOVA, $p = 0.02$), where semantic defects were found faster when using word identifiers than abbreviations, but not to syntax errors (ANOVA, $p = 0.8$). They analyzed the visual metrics for the different styles and tasks using factorial repeated-measures ANOVA. There is no statistical difference for dwell time (semantic, $p = 0.26$; syntax, $p = 0.08$). They found a statistical difference in task of finding semantic errors for FPRT (semantic, $p < 0.001$; syntax, $p = 0.79$). For AOI visits, there was a statistical difference in finding semantic errors for the AOI comment (semantic, $p < 0.001$) where word style was the best in both.

Schankin et al. (53) investigated whether more descriptive name (e.g., *singleWordParts*) is easier to read than single-word identifier (e.g., *parts*). They executed an experiment with 187 students and professionals, where 88 of them completed the survey. The subjects had to find and fix bugs in the code (Debug). The experiment design is similar to experiment of Hofmeister et al., it had code snippets with versions of descriptive name/single-word name and semantic/syntax defects. The authors measured correctness of answers, task completion time and the visual attention. They also used the letterbox (114) for measure the focus

time in an area of interest (AOI), which were tagged in the following areas: commentary, pre-defect, defect, and post-defect. For task completion time of correct answers, they found out a statistical difference for semantic defects (paired t-test, $p = 0.048$, $d_z = 0.27$) where participants were faster in finding semantic defects in descriptive name version. But, they did not find a statistical difference for syntax defects (paired t-test, $p = 0.515$, $d_z = 0.09$). For time spent in AOI, they did not find a statistical difference for any AOI between finding semantic or syntax defects, except for pre-defect (paired t-test, $p = 0.012$, $d_z = 0.39$) and post-defect (paired t-test, $p = 0.025$, $d_z = 0.32$) in semantic defects where the participants spent more time in descriptive name and single-word name, respectively.

4.5.2 Fully-qualified Names

In this section, we describe studies related to qualified names (e.g., `package.class.method()`. Santos et al. (16) compared the using of fully-qualified names instead of using import clauses. We detailed the methodology in Section 4.2.1. They found out a statistical difference (two-tailed test for the proportion, $p = 0.00$) where the participants preferred to avoid fully-qualified names and add import clauses.

Binkley et al. (48) presents a study about the interplay between programmer short-term memory limitations and entity names found within programs. They compared the effect of long and short fully-qualified names for code understanding. The authors asked 158 subjects to recall the Java code snippet (Memorize) and describe the kind of application where this line of code could be found (Recognize). They measured the time and correctness of answers. The authors found out a statistical difference for the time to answer (linear mixed models, $p < 0.0001$) where long names take more time to process than short names. However, they did not find a statistical difference for correctness. They also investigated counting syllables and parts (counted by separating the name based on Java's dot-operator) in place of length (size in letters). The size of syllables was significant for time (linear mixed models, $p < 0.0001$) and correctness (linear mixed models, $p = 0.0019$), but the numbers of parts was significant only for correctness (linear mixed models, $p < 0.0174$) where the long names need more time to process and cause more misunderstanding.

4.5.3 Meaningful Names vs Obfuscated Name

In this section we describe studies that evaluated the effect of techniques to mask identifiers in code readability. Ceccato et al. (49) compared identifiers obfuscated by their technique with a meaningful name. The technique of obfuscating code aims to remove relevant information from the code and add meaningless identifiers. The authors realized two experiments where asked 32 students to perform two types of tasks: (i) comprehension tasks, e.g., where is the part of code that does an operation (Inspect); and (ii) change tasks, e.g., modify the program to have a different behaviour (Trace+Implement). After the experiment, the subjects had to fill a survey (Opinion). The first experiment was performed with 10 Master students and the second experiment with 22 PhD students. The authors assessed correctness, time, and opinion. They found out a statistical difference for comprehension tasks (Mann-Whitney test, exp1: $p < 0.01$, effect-size$= 1.66$; exp2: $p < 0.01$, effect-size$= 1.02$) and change tasks (Mann-Whitney test, exp1: $p = 0.05$, effect-size$= 0.95$; exp2: $p < 0.01$, effect-size$= 1.29$) for both experiments where the meaningful names had more correct answers per time. In the survey they found out subjects in experiment 1 felt that the obfuscation makes feature location more difficult (Mann-Whitney test, $p = 0.04$), while there is no difference for code comprehension tasks (Mann-Whitney test, $p = 0.52$) and change tasks (Mann-Whitney test, $p = 0.20$). For the experiment 2, subjects felt that obfuscation makes all three activities —comprehension, feature location and change— more difficult (Mann-Whitney test, $p < 0.01$ in all cases).

Teasley (45) studied the effect of name styles for code comprehension. In this study, the author compared program of versions where one had meaningful identifiers and another had procedures and variables names masked. Teasley realized two experiments. The first was conducted with 43 computer science students and the second with 18 students of a database course. The author asked, in both experiments, to answer questions about the value of the code element (Trace). The questions evaluated four types of knowledge in the code: (i) operation: actions the program performs at the level of source code; (ii) control flow: execution sequence of program; (iii) state: connections between execution of an action and the state of all aspects of the program that are necessarily true at that point in time; (iv) functional: relations concerning the main goals and subgoals of the program. She assessed the score of the correctness of the answers. In the first experiment, the author found out a statistical difference (ANOVA test, $p < 0.005$) for correctness only in questions about functional aspects of the program where the version with meaningful names were the best. In the second experiment,

the author did not find any statistical difference.

Similarly, the paper by Chaudhary and Sahasrabuddne (68) conducted an experiment with 105 students to investigate the effect of factors of complexity in programs (e.g. names of variables). The authors asked the subjects to explain what the code does (Present) and remember and reconstruct the entire code snippet (Present + Implement). In this experiment, 5 different programs were used (identified by letters A-E), each with a different variation of the factors of the study: (i) complexity, where the program had matrix and vector structures (programs D-E); (ii) purposeful, where the program had a purpose in any problem domain (programs B-E) ; and (iii) meaningful names, where the identifiers names represented any meaning (programs C and E) instead of random letters. It is important to mention that programs B-C and D-E are counterparts. For this section scope, we evaluate only the comparison for the meaningful name factor in the counterparts programs. The authors found out a statistical difference for the correctness of present activity between B-C (F-test, $p < 0.05$, f-ratio$= 4.67$) and D-E (F-test, $p < 0.05$, f-ratio$= 7.15$) where programs with meaningful names were the best. For the reconstruct task, they found out a statistical difference for correctness only between D-E (F-test, $p < 0.05$, f-ratio$= 4.42$) where programs with meaningful names were the best. It worth mention that for that last task the program A (without complexity, purpose, and meaning names) were significantly better than all other programs (F-test, $p < 0.05$).

Avidan and Feitelson (51) performed an experiment to investigate the effect of understand the code in the using of original parameters and local variables names and names with random letters. We detailed the methodology of the study in Section 4.4.4. In this section, we focused in the evaluation about the meaning of names. In six methods evaluated, they found out a statistical difference for three methods (Mann-Whitney U test, $p < 0.05$) where the control group (meaningful names) were more faster than experimental group (variable names were replaced by a, b, c, and so on). For the others three methods, either it is not possible to compare the time because both groups answered the questions incorrectly either there was a significant difference.

Hofmeister et al. (41) investigated the difference for code comprehension in three styles of identifier name: word, abbreviation, and meaningless single letter. The methodology of the study was explained in Section 4.5.1. In the scope of this section, we evaluated the results that compared the letter style with other styles. The authors assessed the defects/min in finding a errors. They found out a statistical difference only in finding semantic errors (ANOVA, $p = 0.02$, $d_z = 0.31$) between letter style and word where the meaningful name style (word)

was the best. They did not find statistical difference between letter and abbreviation style in semantic errors, and between letter and other styles in syntax errors. The authors analyzed the visual metrics for the different styles and tasks using factorial repeated-measures ANOVA. There is no statistical difference for dwell time (semantic, $p = 0.26$; syntax, $p = 0.08$). They found a statistical difference in task of finding semantic errors for FPRT (semantic, $p < 0.001$; syntax, $p = 0.79$) where subjects spent more time with letter style than other styles. For AOI visits, there was a statistical difference in finding semantic errors for the AOI comment (semantic, $p < 0.001$) where letter style visited more times this area than other styles.

Lawrie et al. (47) also investigated the effect of identifier name styles (word, abbreviation, single-letter) in code readability. The methodology of this study is detailed in Section 4.5.1. Considering the scope of this section, we will analyse only the comparisons between single-letter and other styles. The authors used the Bonferroni's correction, then adopted $\alpha = 0.0001$. They found out statistical difference (ANOVA, $p < 0.0001$) in two of twelve questions for description rating, four of twelve questions for confidence, and three of twelve questions for PCIS, where single-letter was the worst for code comprehension.

4.5.4 Meaningful Name vs Another Meaningful

In this section, we described the studies that investigate the effect of the meaning of identifier names in code readability. For example, Santos and Gerosa (16) compared the using of name of dictionary and the using of invented names. We detailed the methodology of this study in the Section 4.2.1. They found out that a statistical difference in opinion of subjects (two-tailed test for the proportion, $p = 0.0001$) where the names of dictionary was considered the best option for readability.

Similarly, the paper of Blinman and Cockburn (46) investigated the effects of the descriptive identifier names and the availability of interface documentation. The authors realized an experiment with 11 students where they asked subjects to read a source code of a function and choose the description that explain what the functions does in a list of options (Relate).The study used a 2x2 within-participants factorial design. The factors were: naming style (descriptive name and non-descriptive name) and documentation (provided and not provided). They assessed the time and the correctness of the questions. Considering that documentation is out of the scope, we only evaluate the results related to the naming style. The authors

found out a statistical difference in time (ANOVA, $p < 0.01$) where subjects spent less time with descriptive names (81.9 seconds) than non-descriptive (177.2 seconds). Also, they found out statistical difference for correctness (ANOVA, $p < 0.001$) where subjects were better in descriptive names (mean: 86.4 percent) than non-descriptive (mean: 36.4 percent).

The paper by Stefik and Gellenbeck (95) carried out two studies aiming to explore how the programming environment and the syntax of language programming affect code comprehension. They performed two experiments. The first sought to assess the effectiveness of the development environment developed by the authors. The second investigated the impact of a set of programming language words and symbols on code comprehension. Considering the scope of our paper, we evaluated only the second experiment. In this experiment, the authors asked 106 students (novices and experienced) to classify words that could represent a programming concept (Opinion). The authors conducted the test-t with Bonferroni correction between the words for each programming concept. Results were presented through a classification based on statistical difference. One of the results of this study shows that the word *for*, as in a for loop, was reliably ranked worse than the word *repeat*. The full results of this study are presented in Table 16.

Table 16 – Summary of Meaningful name vs Another Meaningful name. The symbol * is p<0.05, ** is p<0.001.

Meaningful	**Results**
Stefik and Gellenbeck (95). *Best word for...*	
"doing something zero or more times"	**Novice**: `repeat`* **Experienced**: `loop`**
"stopping doing an action"	**Novice**:`[a<10]`, `(a<10)`, and `|a<10|` **Experienced**:`(a<10)`, `[a<10]`, `a<10`, and `a<10`
"take a behavior"	**Novice**: action **Experienced**: operation, action, method*, function and task
"take a behavior only when 2 conditions are true"	**Novice**: `AND` and `&` **Experienced**: and
"take a behavior when at least one condition is true"	**Novice**: `OR`** **Experienced**: `OR`**
"take a behavior when one condition is true, but not both conditions"	**Novice**: `OR`* **Experienced**: `XOR`**

Table 16 continued

"assigning a value to a computer's memory"	**Novice**: `a=1024**` **Experienced**: `a=1024**`
"doing something only when a condition is true"	**Novice**: `with the condition that A<10*`, `if A<10**`, `only if A<10` and `while A<10`. **Experienced**: `if A<10**`
Stefik and Gellenbeck (95). *best approach to...*	
"tell a noun to perform a behavior"	**Novice**: `Square:move_up*` and `Square=move_up*` **Experienced**: `Square.move_up**` and `Square:move_up*`
"enclose a list of characters"	**Novice**: `"my words"` ($p < 0.05$) and `{my words}` **Experienced**: `"my words"*` and `'my words'*`
"put together two separate lists of characters into one larger word"	**Novice**: `fire+fox` **Experienced**: `fire+fox`
Stefik and Siebert (92). *Best word for...*	
Variable types	**First study**: integer, float, boolean, string, and delimiting text are better. **Second study**: string constructs in Java, Quorum, PHP, and Python are also good.
Variable manipulation	**First study**: the words string concatenation, assignment operator, cast, modulus, increment had the best averages.
Boolean comparison operators	**First study**: and, `boolean equals`, `not equal to`, or, `xor` had the best averages.
Arrays	**First study**: the words array, index operator had the best averages.
Generics	**First study**: the words multiple types generics and generics had the best averages.
Classes and Inheritance	**First study**: the words parent, this, class, is a, abstract had the best averages.
Function Handling	**First study**: the words return, method, return type had the best averages. The results of the second study showed that the function constructs in Quorum, Smalltalk, Go, C++/Java presented a statistically significant result.
Properties	**First study**: the words private, protected, constant, public, static had the best averages.
Other Concepts	**First study**: the words use, dot operator, null had the best averages.
Error Handling	**First study**: the words catch, throw, finally, try had the best averages.

Table 16 continued

I/O	**First study**: the words say, input, output had the best averages.
Comments	**First study**: the words single line comment, multi-line comment had the best averages.
Aspect Oriented Programming	**First study**: the words pointcut and aspect had the best averages.
Method Modifiers	**First study**: the words after, around, and before had the best averages.
What programming language has the best words to program constructs?	**Second study**: general constructs in Quorum, Java, Ruby, C ++ were best for understanding.
Constructor	**Second study**: the constructions in Quorum, PHP, Perl, C ++, Java were the best for understanding.
Inheritance	**Second study**: inheritance constructs in Ruby, Quorum, PHP, C++, Java were the best for understanding.
Stefik and Siebert (92). *Best terms for...*	
Control structures	**First study**: `if`, `loop`. **Second study**: Quorum (`repeat times`), Smalltalk (`timesRepeat`), C ++ / Java (`while`, `do while`, `for`).

In a similar paper, Stefik and Sibert (92) executed four studies aiming to investigate the effect of syntax of programming languages in code comprehension. The first and second studies aimed to identify which words and symbols beginners consider intuitive or non-intuitive in a general-purpose programming language. While the first is a replication of the previous paper (95), the second study compared the syntax of the languages directly. These two studies asked 196 students for their opinion through a fee for symbols and language constructions (Opinion). The third and fourth studies aimed to study whether novices using programming languages for the first time can simply write computer programs more accurately using alternative programming languages. According to the scope of our paper, we considered only the results of the first and second studies. The authors did not applied a statistical test but they ranked the best syntax words based on the mean of the answers.

4.5.5 Linguistic Antipatterns

In this section we discuss about studies that evaluated poor practices in naming variables and other identifiers. The paper by Arnaoudova et al. (10) aimed to identify inconsistencies

between naming, documentation, and implementation of an entity, called by them linguistic anti-patterns. In this study, the authors mined software repositories to search examples of such inconsistencies, which resulted in a set of 17 anti-patterns. They asked 44 participants to provide their opinions about the name of an entity in snippet code (Opinion) where some of them were linguistic anti-patterns and others were their consistent alternative. The opinion was expressed in a 5-Likert scale (from "Very poor" to "Very good"). Those 44 participants, 30 were developers and graduated students external to the projects mined in the first step; and 14 were developers internal to the projects. The authors found out a statistical difference in the median of opinions of subjects (Mann-Whitney test, $p < 0.0001$) where 69% of external and 51% of internal subjects perceived linguistic anti-patterns poor or very poor practices. Also, they analysed each linguistic anti-pattern separately and found out $12/17$ are perceived as poor practice by internal and/or external subjects. For the external subjects, the authors considered as particularly poor when they are perceived as "Poor" or "Very poor" by at least 75% of external subjects. For internal subjects, the authors considered as particularly poor when there was a full agreement among internal subjects or when the internal subjects took an action to resolve them. The Table 17 detail the result.

Fakhoury et al. (17) extended the work of Arnaoudova et al. (10). Differently of the previous paper, they used functional near-infrared spectroscopy (fNIRS) and eye-tracking devices as an active measurement object of program comprehension. The authors asked 70 undergraduate and graduate students to describe the snippet code functionality (Present), find semantic bugs in the code snippet (Debug), and classify the difficulty of the task (Opinion, present, debug). They evaluated the time, correctness, opinion, brain metrics (Oxy- total oxygenation concentration changes-), and visual metrics (fixations). They selected code snippets from open-source projects, modified them to include a semantic bug, and make the following four versions of these snippets: (i) Control, version without linguistic anti-pattern and without structural defects; (ii) LA, version with linguistic anti-pattern; (iii) Structural, version with structural defect; and (iv) LA & Structural, with linguistic anti-pattern and structural defect. In this section we are focusing in the effect of linguistic anti-pattern in code comprehension. When they compared the Oxy between (i) and (ii), they found out a statistical difference (Kruskal-Wallis test, $p = 0.005$, effect-size, $d = -0.992$) which indicates that the presence of linguistic anti-patterns in the source code significantly increases the average Oxy a participant experiences. Then, they compared the difference in the Oxy in fixations of parts of version (ii) that contains linguistic anti-patterns and not contains, they found out a statistical difference

Table 17 – Linguistic Anti-patterns considered as a poor practice by Arnaoudova et al. (10).

Linguistic Anti-pattern	Poor by external	Poor by internal
GET for performs actions other		
"Is" returns more than a Boolean		✓
"Set" method returns	✓	✓
Expecting but not getting single instance		
Comments suggest conditional behavior not implemented		
Validation method does not confirm		✓
"Get" method does not return	✓	
Not answered question	✓	✓
Transform method does not return		
Expecting but not getting a collection	✓	
Method name and return type are opposite		
Method signature and comment are opposite	✓	
Name attribute suggests a single instance but contains many		
Name suggests Boolean but type does not	✓	
Says many but contains one	✓	
Attribute name and type are opposite	✓	✓
Attribute signature and comment are opposite	✓	✓

(Mann-Whitney U tests, $p = 0.00011$, effect-size, $d = 1$) where the linguistic anti-patterns parts increased the Oxy of the subjects. When evaluated the rate of correct answers and the time to answer, the subjects found 76.4% of the bugs in a average time of 3:21 (min:sec) in version (i); and they found 65% of the bugs in 5:13 in version (ii). The authors also analysed the fixations in isolation, they found out a statistical difference (Mann Whitney U test, $p = 0.026$) only in one snippet code of three considered in the study. Finally, they analysed opinion of task difficult, a rate from 1 (less difficult) to 5 (more difficult), and found out that the median were 2.0 and 3.0 for versions (i) and (ii) respectively.

4.5.6 Linguistic Patterns

In this section we describe studies that evaluated the effect in code readability of identifier names pattern that indicates a specific structure or operation in a particular domain. For example, the paper by Wiedenbeck (73) aimed to evaluate the effect in code readability of lines

of code which serve as typical indicators of a structure or operation, named beacons. The author performed two experiments in which she compared the use of beacons with non-beacons. In first experiment, the author asked 12 novice programmers, 12 intermediate programmers, and 12 experienced programmers in the Pascal language to memorize and recall the entire program presented (Memorize), and identify the function of the program (Present). In second experiment, the author asked 12 experienced programmers to study the program and then recall the next two lines of code from a given line of code (Memorize). For both experiments, she assessed the correctness of recalled lines of code. For the first experiment, she did not find a statistical difference in the present activity between the level of subject expertise (Chi square test, $p > 0.05$), and also in memorize activity between beacons and not beacons lines of code (ANOVA, $p > 0.05$). However, they found out a statistical difference between beacons and not beacons when considered the level of expertise (ANOVA, $p < 0.05$) where experienced programmers recalled beacons significantly better than non-beacons (Newman-Keul test, $p < 0.05$). Also, experienced programmers recalled beacons better than did intermediates or novices (Newman-Keul test, $p < 0.05$). For the second experiment, she found out a statistical difference in memorize between different lines of code (ANOVA, $p < 0.05$) where the line with beacon had the best resulted in correctness (Turkey's test,$p < 0.05$).

Similarly, the study by Wiedenbeck (74) carried out four experiments aiming to study the role of beacons in the initial formation of programmers' knowledge about the function of the program. The first experiment evaluated whether a beacon (here, swap lines) can help to recognize the program functionality. They compared two sort functions versions, one with a beacon (no-disguise) and another without beacons (disguise). A total of 80 subjects (novice and experienced) participated in this experiment. The authors asked subjects to perform three tasks: (i) describe the function of the program (Present); (ii) assess the confidence of the answers to know if they had correctly understood the function (Opinion+Present); (iii) remember the program (Memorize). The authors found that the no-disguise function was correctly understood significantly more often than the disguise function (Chi-squared test, $p < 0.05$). Also, they found that experienced programmers were correctly understood significantly more often than novice programmers (Chi-squared test, $p < 0.05$). There is no statistical difference between subject expertise and program version. Again, they did not find a statistical difference (ANOVA, $p > 0.05$) in recalling the programs with and without beacons.

The second experiment evaluated the influence of beacons on functions known to be unfamiliar to the programmers. They compared the effect of code readability in two unfamiliar

sort functions, one with swap (prototypical) and another without beacons (non-prototypical). This experiment had 38 novices and 38 experienced programmers. They assessed the same variables as the previous experiment. They also found a statistical difference (Chi-squared test, $p < 0.05$) where the subjects answered correctly more often the prototypical than the non-prototypical function. Also, they found a statistical difference (Chi-squared test, $p < 0.05$) between the level of expertise where the experienced programmers were better than novices. When they evaluated the recall, there was a statistical difference (ANOVA, $p = 0.003$) where the subjects remembered an average of 8.8 lines of the prototypical function and 6.1 lines of the non-prototypical function. Furthermore, there was a statistical difference (ANOVA, $p = 0.011$) in the level of expertise, where experienced remembered an average of 8.6 lines of code and novices an average of 6.3.

The third experiment investigated whether such an inappropriate beacon leads to false recognition of a program function. The authors compared the effect on the code readability in two binary searching functions, one with a fake swap (intrusion) and another without alterations (no-intrusion). In this study, 38 novices and 38 experienced programmers participated. They measured the same variables as the previous experiment. The authors found a weak statistical difference in the program version (Chi-squared test, $p < 0.10$) where subjects comprehended the no-intrusion version correctly as a binary searching function more often than the intrusion version. Moreover, the subjects answered that the intrusion version was a sort function more than no-intrusion (Chi-squared test, $p < 0.05$). When evaluating the recalled lines, they found that subjects remembered the beacon-like lines in the intrusion version more than the corresponding control lines in the no-intrusion version (ANOVA, $p = 0.006$). Still, the experienced programmers recalled lines more than novices (Chi-squared test, $p = 0.001$).

The fourth experiment evaluated how beacons affect program understanding without other supporting features. They compared four versions of a sort function: (i) correct and without alterations (Correct), (ii) incorrect without one loop (OneLoop), (iii) incorrect without any loops (NoLoop), and (iv) incorrect without swap (NoSwap). This experiment had the participation of 80 novices and 80 experienced programmers. The authors asked subjects to describe the function of the program (Present) and assess the confidence of the answers to know if they had correctly understood the function (Opinion+Present). They found a statistical difference between the program versions (Chi-squared test, $p < 0.05$) where subjects identified the version NoSwap as a sort function less than other versions. Also, the experienced programmers were more likely to answer that the programs were a sort function than novices (Chi-squared

test, $p = 0.019$).

The paper by Chaudhary and Sahasrabuddhe (68), as presented in Section 4.5.3, which aimed to propose a new model to measure understanding the code also compared the program related any domain problem with program with valid statements without any relation with a problem. This study concluded that names of significant variables and familiarity with the problem domain facilitate understanding (F-test, $p < 0.05$), that is, program related any domain problem is positive for understanding. However, it is not possible to generalize these results because one of the programs with a low level of significance also presented a significant score.

The paper by Binkley et al. (48) evaluated the effect of identifiers with *ties* causes in the code readability. According to the authors, ties represent the expectation that the selected part has links with the persistent memory of all subjects. Thus, identifiers with ties are names that have a link with the subject's persistent memory (e.g. `name.substring()`, `numbers.min()`). The methodology was detailed in Section 4.5.2. For the variable ties, only the correctness was assessed. The authors found out that the presence of ties increases correctness (linear mixed models, $p = 0.0009$) in recalling the code snippet.

4.6 MISCELLANEOUS

In this section, we describe studies that do not fit into any of the subcategories presented above. The summary of results is presented in Table 18. In this section, we just describe the comparisons that presented statistical differences.

4.6.1 Typing

This group gathers comparisons about system types and data type conversions. For example, the paper by Kleinschmager et al. (90) investigated whether static-type systems improve the code readability. The authors compared programs with static system type (Java) and dynamic system type (Groovy). For this comparison, they performed an experiment with 36 subjects, but only 33 subjects completed the experiment. These subjects were divided into two groups. The experiment was carried out in two rounds where each group started with a different language, and in the second round, the language was switched. The subjects had to perform three types of tasks: (CIT) identify classes and fill a method stub (Inspect+Implement);

Table 18 – Summary of Miscellaneous.

Factor – Study: Treatment (Programming Language) – Dependent Variables (Activities): Results
Typing
Kleinschmager et al. (90): Static vs. dynamic type systems (Groovy, Java)
Correctness (Inspect+Implement, Debug): Static type systems was better for readability.
Time (Inspect+Implement, Debug): Static type systems was better for readability.
Gopstein et al. (25): Numeric type coercion vs. value transformation (C)
Correctness (Trace): Value transformation was better for readability.
Verbose or Unnecessary Code vs. Concise or Required Code
Wiese et al. (19): Special cases with general solution vs. only general solution (Java)
Correctness (Implement, Trace): Special cases with general solution pattern was better for readability.
Opinion: Special cases with general solution pattern was better for readability.
Wiese et al. (84): Multiple cases vs. general solution (Java)
Correctness (Implement, Trace): Multiple cases was better for readability.
Opinion: Multiple cases was better for readability.
Wiese et al. (19): Use directly an expression to boolean returns vs. use an if construct to test and return true/false (Java)
Correctness (Implement, Trace): Use an if construct to test and return true/false was better for readability.
Opinion: Use an if construct to test and return true/false was better for readability.
Wiese et al. (84): Use directly an expression to boolean returns vs. use an if construct to test and return true/false (Java)
Correctness (Implement, Trace): Use an if construct to test and return true/false was better for readability.
Opinion: Use an if construct to test and return true/false was better for readability.
Wiese et al. (19): Repeating code within if and its else vs. code outside the if/else (Java)
Correctness (Implement, Trace): No difference was found.
Opinion: No difference was found.
Gopstein et al. (25): Code with unnecessary (dead, unreachable, repeated) statements vs. only necessary code (C/C++)
Correctness (Trace): No difference was found.
Others
Yeh et al.(85): Code atoms vs. non code atoms (C/C++)
Correctness (Trace): Non code atoms was better for readability.
Brain Metrics (Trace): Non code atoms was better for readability.
Jbara and Feitelson (69): Regular vs. non-regular (C)
Correctness (Present, Debug, Trace+Implement): Regular program version was better for readability.
Time (Present, Debug, Trace+Implement): Regular program version was better for readability.

(TEFT) fix type errors (Debug); and (SEFT) fix semantic bugs (Debug). They assessed the time to the subject finish each task correctly. The authors analyzed each group's results separately, and considering all data. For the group starting with Groovy, in all tasks, the subjects spent less time in the Java version than in the Groovy version (Wilcoxon test, $p < 0.028$). For the group starting with Java, only 6/8 of the tasks had a statistical difference (Wilcoxon test, $p < 0.034$), and the best result were part for Java and part for Groovy. When all data were analyzed, they found a statistical difference in 6/8 of the tasks (Wilcoxon test, $p < 0.05$), which the subject spent less time in the Java version.

The paper by Gopstein et al. (25), as presented in Section 4.2.1, compare the numeric type coercion with value transformation. In some cases, type coercion could lead to a misunderstanding. For example, one can expect that the expression `(double)(3/2)` results in `1.5`, however the result is `1.0`. An alternative is explicit the expected result in the division, for example, the expression `trunc(3.0/2.0)`. In the experiment, they found out a statistical difference (McNemar's test, $p = 5.17e - 07$, effect size of 0.42), in which the numeric type coercion is considered negative for the understanding of the code.

4.6.2 Verbose or Unnecessary Code vs. Concise or Required Code

In this section, we describe studies that evaluated code alternatives with more or less than the necessary for the functionality. For example, the papers by Wiese et al. (19, 84) compared the effect on code readability of the code pattern in which a general solution and the special cases are explicit (novice version) instead of only a general solution (expert version). The methodology for both studies was described in the Section 4.2.1. In the first study (19), Wiese et al. found that 71% of the subjects considered the expert version (only a general version) most readable, while 94% considered the same version the best style. The results for comprehension accuracy show that 72% of the subjects consider only general solution a good style, and 81% consider it a poor style. They also found that at least 20% of subjects consider a non-expert (general solution with special cases) code snippet to be more readable than the expert version (z-test corrected by Bonferroni, $p = 0.0008$). Finally, the authors analyzed the students' agreement with the experts' choices regarding style and readability. They found a significant result (McNemar test corrected by Bonferroni, $p < 0.0001$) where more students chose the expert code version for the style than for readability.

In the second paper (84), Wiese et al. replicated the previous study focusing on students at

the beginning of the Computer Science course. This paper used the same dependent variables and tasks as the previous study. In this study, the name of the pattern changed to multiple cases with general solutions. The authors found that 61% and 50% of subjects agreed expert version (only general solution) was the most readable and best styled, respectively. They also found out that more than 20% of subjects believe that the novice code (general solution and the special cases) is more readable than the expert version (z-test, $p = 0.055$).

In the same two studies (19, 84), Wiese et al. also compared the difference for code readability of the code pattern that use directly an expression to boolean returns (expert version) and use an if construct to test and return true/false (novice version). In both paper (19, 84), the authors found that more than 20% of subjects believe that the novice version is more readable than the expert version (z-test corrected by Bonferroni, $p < 0.0001$). In a specific analyse in the first paper (19), they found that more students chose the expert code version for the style than for readability (McNemar test corrected by Bonferroni, $p < 0.0001$).

4.6.3 Others

In this section, we grouped studies that are not related to any other. The paper by Yeh et al. (85) investigated whether programmers' brains react differently to a set of 6 confusing code patterns (atoms of confusion). They compared the atoms of confusion with their alternatives. For this comparison, the authors carried out an experiment with eight students where they asked them to answer questions about the characteristics of the code (Trace), assess the difficulty of the task (Opinion + Trace), and provide the confidence level of their answers (Opinion + Trace). The authors assessed correctness, brain metrics, opinion, and confidence. Yeh et al. used an electroencephalogram (EEG) device To capture brain activity that collected 8-channel signals related to the subjects' cognitive load. The authors analyzed whether there is a significant difference in the EEG magnitude between the confounded and non-confused questions. They found out that the alpha and theta magnitude of the confounded questions was significantly higher in most channels (t-test, $p < 0.006$). The authors also investigated whether the questions within the group caused this effect. The result showed no significant differences in alpha or theta magnitude for questions with confused and non-confused patterns. Additionally, the authors performed an analysis to determine whether the absolute power in alpha broadband can predict the performance of subjects. For this, the individuals' performance was measured through the total number of correct answers. They found out a correlation

(Pearson correlation, $r = 0.72, p < 0.05$) between the alpha broadband and the subjects' performance. According to the authors, this correlation remained the same when calculated with alpha power when solving confusing questions (Pearson correlation, $r = 0.70$) and with alpha power when solving non-confused questions (Pearson correlation, $r = 0.73, p < 0.05$).

In the paper by Jbara and Feitelson (69), they evaluated the effect of regularity on code readability. The authors consider regularity as a style of code in which the same structures are used repeatedly. Then, in this study, they compared code regular with code non-regular. The authors performed two experiments with 110 students. In the first, they asked students to explain the program functionality (Present), fix bugs in the code (Debug), add new features (Trace + Implement), and provide the difficulty of the task (Opinion). There were three programs with different levels of regularity: regular program (89.7% regular), sort program (55.6% regular), and array program (36.5% regular). The authors measured the time, correctness, and opinion. They found out a statistical difference between the correctness of the programs (Welch ANOVA, $p = 0.000$). The subjects were better in the regular program and array program (Games-Howell post-hoc test, $p = 0.000$ and $p = 0.002$, respectively) than in the sort program. There is no difference between regular and array programs. Jbara and Feitelson argue that the regular program, despite its high MCC (McCabe's cyclomatic complexity) and LOC (lines of code), is not necessarily worse than the array style, which has very low MCC and LOC. When evaluating the time, they found a statistical difference (One-way ANOVA, $p = 0.013$). The array program was better than the sort program (Tukey post-hoc test, $p < 0.05$). However, there was no statistical difference between regular program and others. Despite this, the regular program was better than others considering the average time per line of code (Tukey post-hoc test, $p < 0.05$).

In the second experiment, they only asked the subjects to explain what the program does (Present), provide their assessment of the difficulty of the programs, and specify the characteristics that made them difficult (Opinion). The subjects received two programs with two versions each: median regular (79.3% regular), median non-regular (60.7% regular), diamond regular (82.8% regular), and diamond non-regular (43.8% regular). The authors also measured the time, correctness, and opinion. In this experiment, the authors found a statistical difference in correctness (Mixed ANOVA, $p = 0.000$) between the regular and non-regular versions. The correctness score of regular versions was better than the non-regular versions. They did not find a statistical difference for time. Furthermore, they did not apply a statistical test for the opinion results.

4.7 DISCUSSION

This section describes aspects of this study related to the multiple tasks reviewed. It also describes the gaps identified in the literature and possible directions for future work.

4.7.1 Addressing the two research questions.

Section 4.1 presents the two research questions that this work aims to answer:

RQ1 What code elements have been investigated in human-centric studies aiming to compare the readability of these elements?

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

The result of our study, for **RQ1** we identified 40 scientific papers that compared structural and semantic elements. We found a total of 196 elements, which can be classified into 5 subcategories and 20 groups. These subcategories are about flow of control (e.g. conditional constructs and styles), expressions (e.g. boolean expressions as conditions), declarations (e.g. preprocessor usage), identifiers and names (e.g. identifier length), and miscellaneous (e.g. typing).

The 196 structural and semantic elements found, we found 113 comparisons. A comparison is equivalent the one structural/semantic element vs. other structural/semantic element. For **RQ2**, out of the 113 comparisons, 85 had exclusively statistically significant results. For these comparisons, the best alternatives were control flow (e.g. If construct, using && and control flow construct), the expressions (e.g. non-negation, expressions greater than and boolean operators equal), the declarations (e.g. method order calling+connectivity, prepossess disciplined and macro definition at the statement level), the identifiers and names (e.g. long descriptive compound, fully-qualified name and meaningful name) and miscellaneous (e.g. static type systems, returning boolean values with operators and regular program version). 28 of the comparisons did not present statistically significant differences (e.g. if vs case and for vs while). In addition, two comparisons showed divergent results between different studies, e.g., typographic and structural organization of nested-if.

Most readability papers, did not perform a power analysis. The study by Wiese et al. (19) reports that the lack of this analysis is a threat to the validity of the study because it can determine the minimum sample size. In total, only 8 readability works performed a power analysis (54), (25), (24), (79), (53), (50), (41), (84). This reveals that most studies on readability involving human beings are still not concerned with the minimum size that a sample must contain for the results to be more statistically reliable.

4.7.2 Contrasting empirical results with existing style guides

In this section, we summarize the main results. As our goal is examine which structural and semantic elements were investigated and which were considered more readable in human-centered studies.

Our results of the **readability** category reveals that, unlike the legibility category, the papers can be divided into 5 subcategories and 20 groups.

Control flow. Most of the papers on control flow carried out studies on conditional constructs and styles. A smaller body of work has investigated repetition constructs and styles. Our results show that in conditional constructs and styles the use of ternary operator is considered a confusing pattern (25). However, Medeiros et al. (18) shows that this pattern is commonly used by developers. Another relevant point is that the results of these two works show that logic as control flow is also negative for understanding the code and is little used by developers. The work of Love (72), collaborates with previous results showing that simple flow control is best understood by students. The work of Ajami et al. (42) shows that using if statements are more understandable compared to using for statements. Additionally, the works by Wiese et al. (19), (84) show that the use of && is more readable than nested if statements. The results also reveal a divergence about the typographic and structural organization of nested-if. The work of Oman and Cook (88) shows that sequential IFs are the most comprehensive. On the other hand, the work of Oman and Cook (89) did not reveal any significant difference.

The body of work that investigated Repetition Constructs and Styles reveals that the Read-/process loop style resulted in less trace time and less error for (78) students. In collaboration with this result, the work of Ajami et al. (42) shows that looping counting up is better understood compared to looping counting down. The work by Benander et al. (54) also shows that recursion is a good factor in understanding the code. Finally, the work by Maćkowiak et al. (79) shows that data-driven matrices seem to be easier to understand.

Expressions. Other works on code readability have studied expression elements. 5 types of comparisons are found in this subcategory. The results of the comparisons on Boolean Expressions as Conditions reveal that the resulting expressions are positive for understanding the code (67). As well, the boolean non-negation pattern is also positive (42). The results also show that the expression "Greater than" is one of the most understandable expressions (67) and that Boolean Operators also help in understanding the code (78).

The results of the comparisons on Implicit intent vs Not implicit intent reveal that the Change of Literal Encoding pattern is negative for code comprehension (25), as well as the Implicit Predicate (25) and Operator Precedence (25) patterns, (18), although the latter is rarely used by developers (18).

Comparisons about expressions with vs. without side-effects show that there is an agreement between the works that study Pre/Pos-Increment vs Separated Operations. The work by Dolado et al. (24) reveals that side-effect statements reduce effective performance in code comprehension tasks. The work by (18) states that pre/post-increment is little used by developers and has a high negative perception on the part of these individuals. And the work by Gopstein et al. (25) supports these claims by showing that separate operations are better for understanding code. Another interesting result found in this set of comparisons refers to the use of assignment as value. The work by Gopstein et al. (25) concluded that this pattern is negative for understanding the code. The work by Medeiros et al. (18) on the other hand, showed that this pattern is commonly used by developers and has a low negative perception.

On comparisons regarding Conflicted vs. decomposed expressions, the results show that the use of comma operator is negative for understanding the code (25). The work by Medeiros et al. (18) showed that this pattern is not commonly used and developers have a negative perception about the use of this pattern. Another significant result found in this set of comparisons refers to store a reference to that subproperty, which proved to be a good pattern in understanding the code (16).

Comparisons over Pointer and arrays show that the use of pointer arithmetic has no negative impact on code understanding (25). It is also little used and has a negative perception by the developers (18). Another significant result found in these works is that Reversed subscripts are also a negative pattern for understanding the code (25) and that it is not a commonly used pattern, but which has a negative perception on the part of (18) developers. Another interesting result presented in the work by Maćkowiak et al. (79) shows that programs with board programming, color selection, and heads are easier to understand.

Declaration. Another set of works on readability focused on studying declarations. 4 types of comparisons were found in this subcategory. The first one is comparisons about lexical order. For example, the work of Geffen and Maoz (77) reveals that calling+connectivity reduces code comprehension time. The second type of comparison is preprocessor usage. The work of Malaquias et al. (91), for example, showed that undisciplined annotations increase the execution time of maintenance tasks and are more error-prone. The work by Gopstein et al. (25) reveals that macro definition and macro operator precedence are negative for code understanding. The third type of comparison brings together studies on types of modularization. The work by Woodfield et al. (75) reveals that modularization is a good pattern for understanding code. The last type of comparison found in this subcategory is about variable usage. The results show that repurposed variables negatively influence the understanding of the code (25). On the other hand, using parameters is better for code understanding (51). In the same way, programs based on a single assignment with exemplary calculations seem to be easier to understand (79).

Identifiers and names. A considerable number of readability works have investigated identifiers and names. 6 types of comparisons were found in this subcategory. The first of these are comparisons over identifier length. These comparisons reveal that more descriptive identifiers were better at finding a semantic defect (53). As well, long and meaningful identifiers are also good for code understanding (47). Another type of comparison found is over fully-Qualified names. These comparisons reveal that avoiding fully qualified import clauses is a good standard for code readability (16). Another important result found in this type of comparison is that fully qualified short names are better for understanding (48).

The third type of comparison found addresses the aspect of meaningful names and obfuscated names. The results of comparisons of this type show that clear names are better for code understanding (49). Meaningful names are also better given the functional level of understanding (45) (68). As well, the original names are also good for understanding (51). Another interesting result is that abbreviation and full-world are better than variables named with a single letter (41).

The fourth type of comparison found in this subcategory studied the use of a meaningful name versus the use of another meaningful name. The results of these comparisons show that the most influential programming languages use words, symbols, and semantics that can be quite intuitive for beginning programmers, but some cases are really not intuitive, for example, the use of control structures for loop (95) . In addition, the results of these

comparisons also reveal that fully descriptive nomenclature is an excellent choice to maintain code understanding. The fifth type of comparison found in this subcategory gathers patterns about linguistic antipatterns. The work by Arnaoudova et al. (10) reveals that developers perceive linguistic antipatterns as negative practices that should be avoided. The sixth type of comparison found in this subcategory brings together studies on linguistic patterns. These comparisons show that beacon code is easily remembered (73) (74). Also, the program with valid statements unrelated to a problem (68) and identifiers with ties are good standards for understanding (48) code.

Miscellaneous. In this subcategory, 3 types of comparisons are found. Comparisons on typing show that static-type systems are better for maintenance tasks, but do not have a great effect on semantic error correction activities (90). On the other hand, using numeric type coercion is negative for understanding the (25) code. The second type of comparison is about verbose or unnecessary code and concise or required code. These comparisons reveal that code with general solution patterns is more readable (19). Some students believe that unnecessary code is also more readable (84). Returning boolean values with operators and returning boolean values without operators also positively influence (19) readability. And finally, repeating the code inside if and else is also positive for understanding according to students (19). The third type of comparison is named after others. Their results show that confusion atoms are negative for readability (85). Just as non-regular programs are also negative (69).

The software engineering scenario has evolved considerably in recent years, which makes it necessary to carry out new investigations on previously studied concepts. For example, some included studies on linguistic patterns were carried out between the years 1980 and 2009. Although the results are quite relevant, the languages used in these studies are little used nowadays. Wiedenbeck (73), (74) I use PASCAL, the study by Chaudhary and Sahasrabuddne (68) used FORTRAN. On the other hand, the study by (48) used JAVA. Therefore, the results obtained in these studies cannot be generalized to other languages. This reveals another important point, which is the lack of replication of studies. There are few works included in this review that carry out this type of study. In this sense, replications can be made to investigate readability aspects in other languages such as Python and JavaScript.

4.8 THREATS TO VALIDITY

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

Construct validity. Our study was built on the selected primary studies, which stem from the search and selection processes. The search for studies relies on a search string, which was defined based on the seed papers. We chose only conferences to search for seed papers. A paper published in a journal is often an extension of a conference paper and, in Computer Science, the latest research is published in conferences. Furthermore, we focused on conferences only as a means to build our search string. Journal papers were considered in the actual review. Additionally, we only used three search engines for our automatic search.

Other engines, such as Springer and Google Scholar, could return different results. However, the majority of our seed studies were indexed by ACM and IEEE, and then we used Scopus to expand our search. Moreover, while searching on the ACM digital library, we used the *Guide to the Computing Literature*, which retrieves resources from other publishers, such as Springer. Finally, we avoided Google Scholar because it returned more than 17 thousand documents for our search string and we did not have the resources to analyze so many papers.

Internal validity. This study was conducted by multiple researchers. We understand that this could pose a threat to its internal validity since each researcher has a certain knowledge and way of conducting her research activities. However, each researcher conducted her activities according to the established protocol, and periodic discussions were conducted between all researchers. Another threat to validity is the value of Cohen's Kappa in the study inclusion step ($k = 0.323$), which is considered fair. This value stems from the use of three possible evaluations ("acceptable", "not acceptable", and "maybe") in that step.

However, we employed "maybe" to avoid having to choose between "acceptable" and "not acceptable" when we had doubts about the inclusion of a paper—all papers marked with at least one "maybe" were discussed between all authors. Moreover, a few primary studies do not report in detail the tasks the subjects perform. Because of that, we might have misclassified these studies when mapping studies to task types. Also, another threat is the period when primary study searches were conducted. Relevant new work may have been published recently and its results may impact our conclusions. To mitigate this threat, we intend to perform a search with the same search string considering only the subsequent years of the first one.

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

4.9 CONCLUSION

We present in this paper a study in which we investigated which code elements were investigated and which were considered more readable in human-centered studies. Through a systematic review we synthesized the results of 38 previous studies on code readability. We present an extensive categorization with all the elements found. Our analysis took into account the response variables, the tasks and the subjects who participated in the analyzed studies.

We show that the 38 studies can be categorized into 20 groups that can be divided into five readability subcategories: control flow (conditional constructs and styles, repetition constructs and styles), expressions(boolean expressions as conditions, implicit intent vs not implicit intent, expressions with vs. without side-effects, conflated vs. decomposed expressions, pointer and arrays), declarations(lexical order, preprocessor usage, no modularization vs types of modularization, variable usage), identifiers and names(identifier length, fully-qualified names, meaningful names vs obfuscated name, meaningful name vs another meaningful name, linguistic anti-patterns, linguistic patterns), miscellaneous(typing, verbose or unnecessary code vs concise or required code and code atoms vs non code atoms).

Our results show that were found 196 structural and semantic elements. For control flow subcategory elements like If construct, simple control flow construct and looping couting up has positive results in the studies investigated. For expression subcategory elements like non-negation, explicit predicate and statement separated has positive results. For declarations subcategory elements like preprocessor disciplined, modularization and unique purposed variables has positive results in the studies. For identifiers and names subcategory elements like identifiers more descriptive, avoid full qualified names and meaningful name has positive results. For miscellaneous subcategory elements like regular program and not use code atoms has positive results.

Our study focused on structural and semantic elements of the code. Even so, other studies on code comprehension have investigated aspects related to metrics and tools for readability. Thus, future studies can investigate how these metrics and tools are built and propose improvements through our results. In addition, other studies may also investigate which main learning activities are being used in readability studies.

5 UNDERSTANDING CODE UNDERSTANDABILITY IMPROVEMENTS IN CODE REVIEWS

To accomplish this, we manually examine 2,401 code review comments from Java open-source projects on GitHub and categorize them based on their focus on improving code understandability. We then investigate a subset of 300 comments that pertain to code understandability and identify eight categories of code readability issues and the solutions that were applied to fix them. This work aim to investigate how developers improve code understandability during software development regarding what code alternatives are the best for code understandability (goal #3).

5.1 CODE REVIEW

Code review is an assessment of source code to identify a variety of quality issues, for example, defects, security vulnerabilities, poor understandability, and bad design, among others. In collaborative development platforms, code review takes place when source code changes are proposed to a project by a developer, the *submitter*. Then, the proposed changes are reviewed by other developers, the *reviewers*, who provide comments with, for instance, suggestions or requests for improvements that the submitter should apply to the code. Submitters and reviewers interact until the reviewers consider that the proposed code changes are good enough to be integrated into the codebase (else the proposed changes are rejected).

GitHub, the most popular collaborative software development platform, allows code review in the context of pull requests. Pull requests' reviewers can provide general code review comments in the body of the pull request timeline or *inline code review comments* that are specific to source code lines of the changed files. Inline comments are about fine-grained, localized reviewers' concerns, because they explicitly point to specific source code lines where the concerns lie. Figure 12 shows an example of an inline code review comment. The source code lines highlighted in green represent a change proposed to a GitHub project through a pull request. A reviewer named "eshanholtz" wrote an inline comment associated to line 190, suggesting the replacement of the `if` block by a ternary expression. Here, the reviewer's concern is explicitly about readability.

In this work, we leverage code review comments to investigate the reviewers' concerns related to code understandability and how the submitters address these concerns. Because we

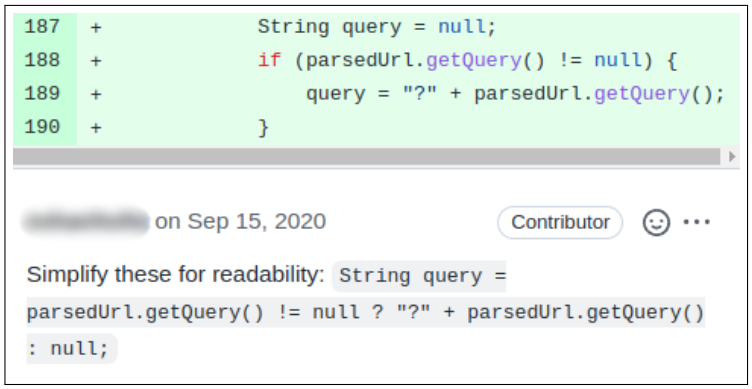


Figure 12 – Inline code review suggestion: replace an `if` statement by a ternary expression. (2)

want to examine the solutions developers adopt at a fine level of granularity, e.g., the replacement of an `if` statement by a ternary expression (Figure 12), we focus on inline code review comments (we use the term code review comment for simplification). Since code reviewers are quality gatekeepers in software development, we hypothesize that, if code understandability is an important concern in practice, there are code review comments about it.

5.2 CODE UNDERSTANDABILITY

Legibility and readability are both important factors for making code easy to read, but understandability is the ultimate goal. Legibility refers to how easily individual code elements can be distinguished from each other, while readability refers to how easily developers can comprehend the code as a whole (56). Ultimately, understandability encompasses both legibility and readability, and refers to how easily developers can extract relevant information from the source code, including non-functional parts, such as documentation, to perform software development or maintenance tasks (86). In this work, we are interested in understandability.

In the literature, many studies have evaluated how to improve code legibility and readability (75, 87, 19, 18, 25, 41). All these studies are based on an informal notion of *understandability improvement*. A code understandability improvement is a refactoring that, when applied to a program, makes its source code better at communicating to the developer what it does, according to to one of more quality criteria (56).

A basic example of this type of change is renaming. Renaming a program element, in general, is not a functional correction and does not change the program functionality. However, a change to the name of a program element may make it clearer for other humans, by better communicating purpose or rationale. The main motivation for renaming is to improve code

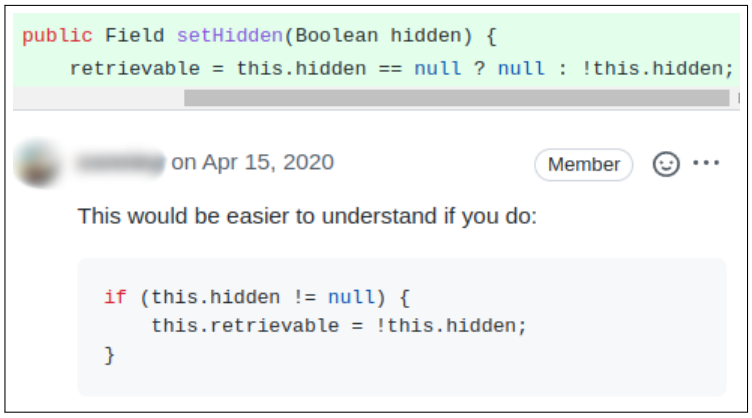


Figure 13 – Inline code review suggestion: replace a ternary expression by an `if` statement (3).

understandability, although there are some exceptions where renaming is not motivated by understandability, such as when it matches a pattern of reflection. Moreover, certain aspects of source code are pertinent to understandability by nature, specifically documentation and formatting. Although disregarded by compilers, these elements are deliberately integrated into the source code to assist developers in comprehending the code.

Many factors in the source code can affect code understandability, such as the number of identifiers, size of a line of code, and nesting (115, 30). However, the definition of whether a change is an understandability improvement depends on the developer's background (e.g., skill and experience) (86). Also, we could consider this can be affected by the developer's goals (e.g., implementing a new feature, writing tests, and fixing a bug). In the literature, for example, Gopstein et al. (25) found that subjects misunderstand the output of a code snippet when it contains ternary/conditional expression (e.g., `a?b:c`) instead of an `if` statement. In a similar study, Langhout and Aniche (20) did not find that the ternary operator causes misunderstanding. In a study by Medeiros et al. (18), the practitioners answered that ternary/conditional expression does not influence code understandability positively or negatively.

The perception of what understandability improvement is depends on the context in the real world. During code review, reviewers suggest source code changes to improve code understandability, but the development team can influence what is considered an improvement. For instance, in Figure 12, a reviewer suggested replacing an `if` statement by a ternary expression to improve code understandability. In Figure 13, which refers to another open-source project, a reviewer suggested replacing a ternary expression by an `if` statement to improve understandability. These two code reviewers suggest the opposite operations, both concerning understandability. This means they perceive understandability improvements (in the context of the usage of `if` statements vs. ternary expression) in a different way.

Style guides (a.k.a. code conventions) also are adopted to improve code understandability in the real world. Smith et al. (26) defined that *"code conventions are a body of advice on lexical and syntactic aspects of code, aiming to standardize low-level code design under the assumption that such a systematic approach will make code easier to read, understand, and maintain"*. However, the recommendations proposed by different style guides can be divergent. For example, while the Google Java Style Guide[1] suggests the use of braces even when the block is empty or contains only a single statement (e.g., `if (a) b+c`, the Linux Kernel Coding Style[2] disagrees with it for single statements (e.g., `if (a) b+c`). Understandability improvements are, to some extent, in the eye of the beholder, i.e., they depend on the experience and context of the developer. There is insufficient empirical evidence to support the creation of a style guide. This study could contribute to it.

5.3 STUDY DESIGN

This section presents the research questions we aim to answer (Section 5.3.1), the data collected for the study (Section 5.3.2), and the process for identifying code review comments that suggest understandability improvements (Section 5.3.3).

5.3.1 Research Questions

RQ1. *How often do reviewers ask for code understandability improvements in code review comments?*

Motivation: While code reviews are commonly employed to enhance code quality (27, 28), there is a lack of information regarding the prevalence of code review comments focusing on code understandability. If a substantial portion of code review comments involves understandability improvements, it may be advantageous to prioritize automation strategies tailored to address this specific area. Furthermore, not every improvement targets executable source code and it may be useful to quantify the extent to which other scopes, such as code comments and string literals, are targeted. Such targeted approaches could lead to significant time and effort savings.

Method: We analyze a representative sample of code review comments to determine the rate

[1] <https://google.github.io/styleguide/javaguide.html>
[2] <https://www.kernel.org/doc/html/v4.10/process/coding-style.html>

at which reviewers suggest code understandability improvements. In this direction, we manually classify 2,401 code review comments selected as explained in Section 5.3.2, assessing whether each comment pertains to improving code understandability. Furthermore, we categorize the associated scopes of these comments (e.g., documentation, executable source code, string literals). Each code review comment was classified by two different authors of this work. We calculated the Kappa coefficient (63) for assessing the agreement between the two analyzes. We found $k = 0.59$, which is considered a moderate agreement strength (58). For every case where there was a disagreement, three or more authors engaged in discussion to reach a final classification. Besides this classification, we also verify the scope to which each comment refers, i.e., where in the source file the reviewer has identified an opportunity to improve understandability (e.g., source code and Javadoc). Additionally, we randomly selected and classified 300 code review comments related to code understandability. We distinguished between explicit comments, which indicate that the improvement suggestion is motivated by code understandability, and implicit comments, where the intention of code understandability improvement is inferred from the comment text, code snippet, and the code improvement itself. We also provided a description of why we considered each code review comment to be related to code understandability. In the sample, we found a total of 32 explicit comments and 268 implicit comments related to code understandability.

Novelty: Previous studies (116, 117, 118) explored the projects' files histories to identify code understandability improvements based on state-of-the-art metrics or commit messages that reported an understandability improvement explicitly. We evaluate 2,401 code review comments and identify suggestions by code quality specialists (reviewers) to improve source code understandability. Unlike previous work (119), we also examine when the intention to improve code understandability is implicit.

RQ2. *What are the main issues pertaining to code understandability in code review?*

Motivation: Identifying the typical understandability issues reported by reviewers can guide researchers and tool builders when designing and training machine learning-based tools to improve understandability. Additionally, this knowledge can contribute to the development of code review checklists (120) to make the process more systematic and efficient to address code understandability concerns in a structured manner (121).

Method: We have selected 300 code review comments randomly from the 1,012 reviews that were related to code understandability improvements and were classified in Section 5.4.1.

This sample size of code review comments is larger than a representative sample (279) with a confidence level of 95% and a margin of error of 2% of the 1,012 instances, according to Cochran's formula (122). We then identify the issues that triggered the suggestions for change (the what) and group them into categories that we called *understandability smells*. For example, in one code review [key=821[3]], the reviewer asked to remove three code elements because they were constants that were not being used, i.e., UNNECESSARY CODE (the "what"). Also, we identify the type of code element targeted for improvement (the "where") using Spoon (123), a well-established open-source library for analyzing Java source code. Three different authors examined the sample of 300 code code review comments. They described the issue presented in each comment and identified where the issue occurred in the code. The same authors then discussed the issues and grouped similar comments together. Afterwards, the groups were refined, and categories were established during meetings involving at least five authors. We use the nodes of Spoon's AST to group constructs related to each issue pointed out by code reviewers into categories. For example, we group the nodes `CtIf` and `CtSwitch` into the Conditional category[4] We consider the most specific AST node to which an understandability smell is associated to determine where it happens. For example, in a code review [key=123], a reviewer requested using an alternative method invocation to evaluate an expression in an `if` statement, and we categorize the most specific node identified by Spoon, `CtInvocation`, as a Call. However, in other scenarios, the understandability smell consists of a set of nodes, and we consider the innermost parent node that includes them. For instance, in another code review [key=1878], the reviewer asked to rewrite a method that included a declaration, an invocation, and an `if` statement. In this case, the innermost parent node is the Method itself, `CtMethod` in Spoon. We also categorize it as Method when the reviewer requested renaming the method, adding JavaDoc to it, or moving it. In two code review comments where the reviewers pointed out multiple issues in different parts of the source code, we classify the "what" and "where" based exclusively on the location where the code review comment was marked. For example, in a code review comment in the class declaration where the reviewer asked to *"remove the Db [from class name] and name attributes from the annotation"* [key=1772], the code review comment is at the line of the class declaration and we consider that this (Class) is the place where the issue was pointed out.

[3] Hereafter, we will use the reference [key=####] to represent a link [https://codeupcrc.github.io?key=####], pointing to the code review comments in our dataset.

[4] The `CtIf` node refers to an 'if' statement in the AST, while the `CtSwitch` node refers to a 'switch' statement. Both are conditional statements in the Java language.

Novelty: Previous studies (116, 117, 118) classified the issues related to code understandability in terms of software maintenance types and rules of static analysis tools. However, their classification is limited to these terms, which do not capture the actual code understandability issues. In this work we examine code review comments, as in the work of Dantas et al. (119), but take two complementary perspectives: coarser-grained, by eliciting the high-level issues that trigger improvements, and finer-grained, by examining which source code elements are connected to these issues.

RQ3. *How likely are understandability improvement comments to be accepted?*

Motivation: Analyzing the odds of developers accepting comments when they suggest understandability improvements highlights the importance or lack thereof of this kind of comment. It may provide arguments for recommendations in cases where automatically addressing the identified problems is not possible.

Method: We aim to investigate if recommendations for improving code understandability are more likely to be accepted than code reviews that do not address code understandability. To achieve this, we took the pool of 300 code reviews that were analyzed for RQ2 and added another random sample of 300 code reviews that suggest changes unrelated to code understandability. The sample was divided among the three first authors who manually searched for changes during the pull request in the source code commented by the reviewer. They then verified whether the change made was an improvement suggested by the reviewer. We classify improvement suggestions in reviews as either accepted or not accepted based on whether or not we find the code changes as suggested by the reviewer. We calculate the odds ratio to determine the likelihood of accepting a suggestion, given it is a code understandability improvement.

Novelty: Brown and Parnin (124) investigated the acceptability of *GitHub suggested changes*, a special kind of code review comment where reviewers recommend a specific change to the code line or lines. However, as far as we know, no study evaluated how developers reacted to the reviewer's suggestions for code understandability improvements.

RQ4. *What code changes are found in understandability improvements?*

Motivation: There are multiple studies (75, 87, 19, 18, 25, 41, 125) that empirically evaluated the impact of different ways of writing code on understandability in a lab setting. However, understandability strongly depends on the context where the code must be understood, e.g.,

due to the experience of the readers. This RQ aims to help developers and researchers to know what makes the code more understandable in practice. Information about typical real-world understandability improvements can help us establish what kinds of tools can address each scenario (linters, more in-depth static analysis, LLM-based, etc.).

Method: We take as a starting point the set of code reviews examined for RQ2 and RQ3. For this question, we focus on the code review comments where the recommendation made by the reviewer was accepted, and a patch was applied as a consequence. We analyze the code change applied in each patch to resolve the problem raised by the reviewer. The sample was divided among the three primary authors, who described the improvements made in response to the reviewer's comments. We then organize these different types of changes into categories based on the understandability smells identified in RQ2. Finally, we discussed how the improvements could be organized in meetings with at least four of the authors.

Novelty: As with RQ2, we adopt a bottom-up approach where we identify code understandability improvements in terms of the fine-grained refactorings that developers perform. Previous studies (116, 117) employed a top-down approach, classifying the understandability smells in terms of software maintenance types and rules of static analysis tools.

RQ5. *To what extent are accepted code understandability improvements reverted?*

Motivation: Understanding the frequency at which code understandability improvement patches are reverted after being applied during the code review process is essential for assessing the subjectivity of code understandability improvements. Frequent reversions of such patches may indicate a high level of ambiguity and variability in the perception of code understandability, which can lead to challenges when utilizing this data to train machine learning systems, potentially resulting in suboptimal performance (126).

Method: For this RQ, we employ the same set of code review comments that were analyzed for RQ4, i.e., comments about understandability where the recommendation made by the reviewer was accepted and a patch was applied. We manually search subsequent versions of the files modified in that patch to determine whether the patch has been reverted. We consider that a reversion happens when (i) a reviewer makes a suggestion to improve code understandability, (ii) that suggestion is accepted and a patch is submitted as a consequence, incorporating said suggestion, and (iii) subsequently, a new patch is submitted, undoing the understandability improvement. In this process, we account for renamed files and cases where portions that were

affected by the patch were moved to other parts of the system, e.g., a different file. If the latest version of the file still includes the patch that was introduced as a response to the suggestion for improvement, we consider that it was not reverted. If the file as a whole or the element (method, class, etc.) where the modification was performed is deleted, we also consider that the change was not reverted, since we cannot ascertain whether the deletion is connected to said change.

Novelty: Kalliamvakou et al. (127) explored the integration rate of proposed changes in GitHub pull requests into the codebase, while Shimagaki et al. (128) assessed the frequency of commits that were subsequently reverted in project histories. However, a gap remains in the existing literature, as there is no study that specifically examines the adoption and reversion of code understandability improvements within the context of code reviews and the history of the codebase.

RQ6. *Do linters contain rules to detect the identified code understandability smells?*

Motivation: Linters are widely used in real-world software development (129, 130, 131). Furthermore, they are capable of detecting a number of typical problems in source code. It is then worth investigating the extent to which they capture the code understandability issues that code reviewers point out. Ideally, reviewers should not have to worry about problems that an automated tool can capture.

Method: We investigate the coverage of the code understandability smells identified in our study by rules implemented in Spotbugs (132), PMD (133), SonarQube (134), and Checkstyle (135). These four tools have the highest number of rules (131), are widely used by practitioners (130), and have been studied by researchers (136). Together, they have a total of 1,315 rules. We manually checked each observation of an understandability smell against each tool to determine if it could detect it.

We utilize the set of code review comments that were previously analyzed for RQ4, which are about understandability smell instances. For each understandability smell, we manually inspect the descriptions of the rules implemented in linters for finding one that could detect the smell. In the end, we calculate the coverage of understandability smells by computing the number of instances that could be detected by at least one of the linters.

Novelty: Previous work (119) investigated the extent to which SonarQube can detect code understandability issues in code reviews by running this tool. In contrast, in this work we

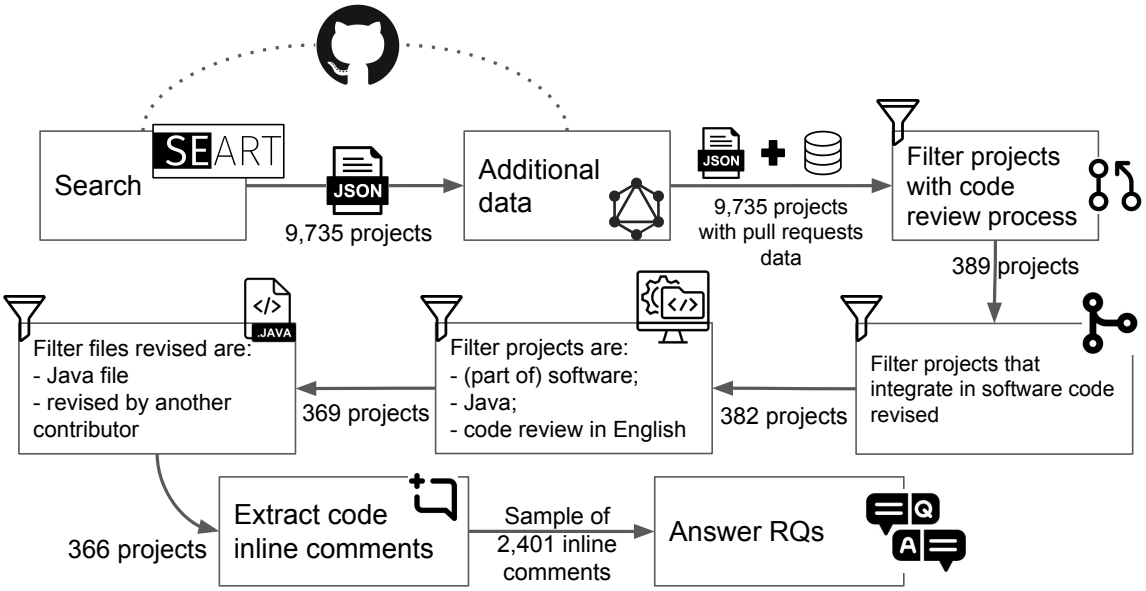


Figure 14 – Selection of projects and extraction of inline comments.

manually analyze the applicability of every rule implemented by four different linters. The issues reported by linters depend on how they are configured (137), as linters sometimes have contradicting rules, e.g., SonarQube includes a rule *Close curly brace and the next "else" [..] should be on two different lines* but also one that ends with *...on the same line*. A manual analysis can capture these inconsistencies.

5.3.2 Data Selection

To answer our research questions, we select open-source projects with an active code review process in Github, i.e., projects where new pull requests are created frequently. Our focus is on the code review comments made by project contributors regarding the code changes submitted through these pull requests. Furthermore, those projects must be non-trivial software systems, e.g., applications, infrastructure components, libraries, frameworks, etc., with many contributors. We start out by using SEART (138) to search for project candidates because it offers appropriate criteria to search for projects. In SEART, we consider the following criteria: (i) Java projects, (ii) at least ten stars (fixed by SEART), (iii) at least one pull request, (iv) at least ten collaborators, (v) not a fork repository.

We select projects with ten or more contributors because we need a number of contributors that justifies the code review process. These criteria contribute to filter out personal projects (127). SEART returns a list of 9,735 projects.

Next, we filter the projects to select only those where developers frequently submit their

Table 19 – Descriptive Statistics for the code review comments, discussions, and source code lines associated to these comments and discussions, for the 363 selected projects.

	Mean	Std.	Min.	25%	50%	75%	Max.
# chars/comment	120.2	216.9	0	30	72	148	44,043
# chars/line of code	53.0	95.9	0	30	49	72	22,382
# chars/discuss.	224.2	435.7	0	50	115	247	45,101
# comments/discuss.	1.9	1.2	1	1	2	2	43

code changes for review by another project contributor. Thus, we select projects with at least one pull request per month in 2020, that includes one or more inline code review comments. These requirements ensure that code reviews are commonplace and happen at the level of source code lines. Given that our search was conducted in October 2021, we focus on pull requests submitted during the year 2020. With this filter, we keep 389 projects.

We further refine this set by removing projects where we can not identify the pull requests that were merged, since merging a pull request is a clear criterion to determine whether said pull request was accepted. Merging can be done manually through *git* commands on the local repository and then pushing to the remote. Alternatively, it can be done by using the merge button directly on the GitHub platform. Pull requests with automated merging have a clear indication of this event on the pull request's timeline, with its status updated to "merged." However, identifying projects with manual merging can be challenging as it requires the use of heuristics to determine the commit that merged the pull request, as explained by Kalliamvakou et al. (127). We identify eight projects (e.g., apache/gobblin and openjdk/skara) where the pull requests are not merged automatically and decided to remove them. After performing these filtering steps, we have 381 projects with an active code review process that integrates their code changes in their main branch through Github.

Then, we perform a fine-grained filtering of these 381 projects. We manually analyze each one and evaluate whether it is mainly composed of code. As a result, we exclude eight projects that do not fit our software criteria, such as those classified as documentation. Also, we constrain our investigation to projects where most code review comments are written in English. We remove seven projects where most code review comments are not in English. In addition, one project (returntocorp/semgrep) didn't primarily use Java, and another (apollographql/apollo-android) was transitioning from Java to Kotlin. As a result, we remove both. Finally, in one project (fisco-bcos/web3sdk) the code review comments are all made by bots, such as *"sonarcloud"*. This project is also removed, leaving us with a final total of

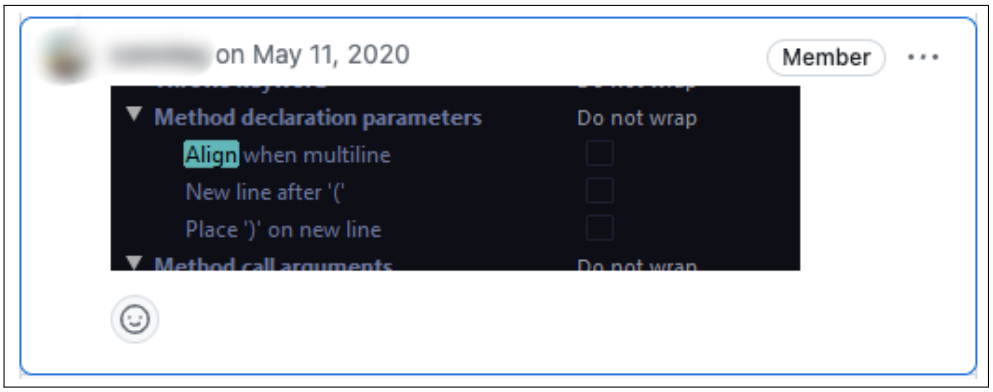


Figure 15 – Example of comment comprised solely of images (4).

363 projects. The list can be found at <https://codeupcrc.github.io/projects.html>. The selected 363 projects comprise more than 349,000 code review comments in total made in 2020. Table 19 presents descriptive statistics of these code review comments and their associated discussions. The table indicates that in some cases, code review comments can be zero characters. These comments comprise only images (Figure 15 presents an example), bullet points without accompanying text, emojis, or are simply empty.

We download these code review comments using the GitHub GraphQL API [5]. Since we need to manually evaluate and classify the code review comments, we extract a statistically representative sample. We chose a confidence level of 95% and a margin of error of 2%, resulting in a sample size of 2,401 code review comments. We employ a stratified random sampling (139) strategy, as the resulting sample can be more representative of the investigated population of code review comments than if we just employed simple random sampling, where projects with few code review comments might not be represented. We randomly select code review comments from the projects in proportion to the total number of code review comments each project has. This means that we select more code review comments from projects that have a higher number of code review comments. As a result, the minimum number of code review comments per project is 1, the maximum is 15, and the median is 7. The complete list of analyzed comments can be browsed at <https://codeupcrc.github.io>.

5.3.3 Identifying Understandability Improvements

To answer the research questions of this study, we need to classify whether reviewer comments suggest understandability improvements. In this classification process, we need to iden-

[5] <https://docs.github.com/en/graphql>

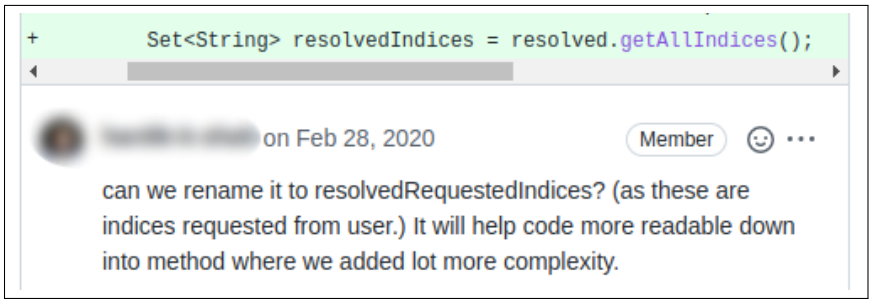


Figure 16 – The reviewer explicitly asks for understandability improvement (5).

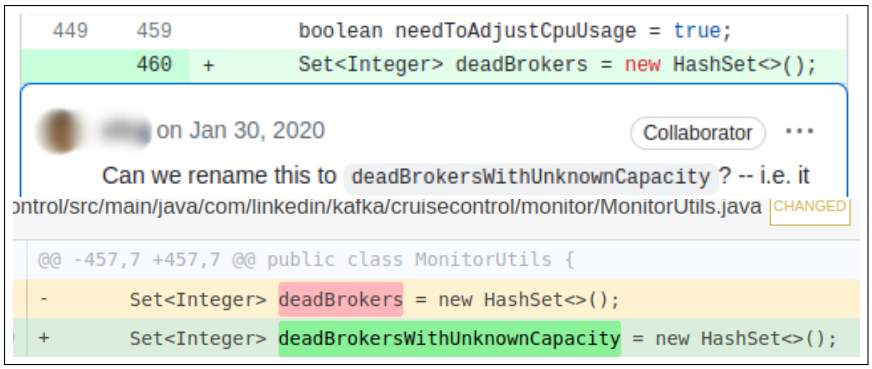


Figure 17 – The reviewer implicitly asks for understandability improvement (6).

tify the intention of the reviewer based on their comment. Sometimes, this intention is clear and explicit in the comment text. For example, in Figure 16, the reviewer asked to rename a local variable because *"it will help code more readable down into method [...]"*. However, other comments are terse, and the intent is not explicitly stated. For example, in Figure 17, the reviewer asks to rename *deadBrokers* to *deadBrokersWithUnknownCapacity* but the intention is not explicit. Although the reviewer does not write it, the most likely motivation is that the new name helps developers better understand the purpose of this variable.

When the inline comment is not enough, it is necessary to consider more information in the context, i.e., the replies to the comment, the source code, and the patch (when available). We first check whether the suggestion does not change the code's functionality. For example, in Figure 18, the reviewer asks the author of the pull request to pass another object to function `contains` instead of the variable `eol`. We need to evaluate whether the suggestion changes the function of the code snippet to which the suggestion refers because it could indicate that the reviewer is suggesting a correction or different functionality. When we look at the patch, we confirm that the suggestion is behavior-preserving, i.e., the old and the new versions produce the same result given the same input. Thus we consider it as an understandability improvement. However, we have found some comments where the logic of the source code changed, but the function is the same. For instance, in Figure 19, the reviewer suggests changing how a list is processed in the stream, but the expected result of the stream is preserved. We also consider

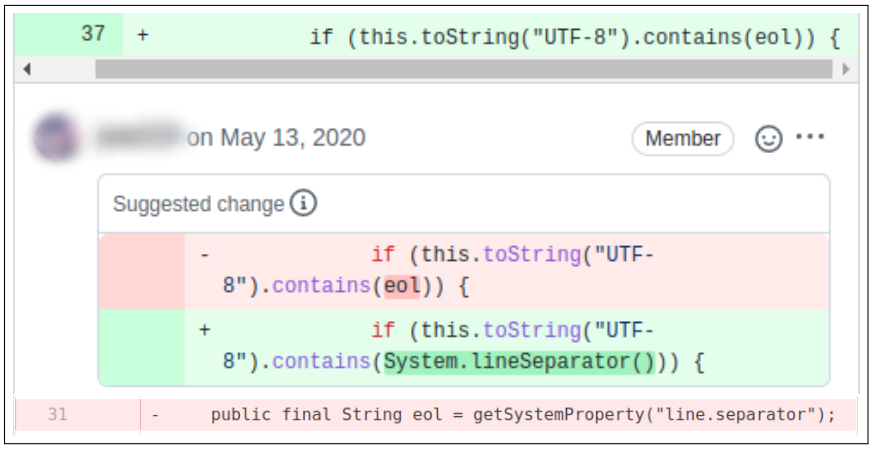


Figure 18 – The intention of the comment is ambiguous (7).

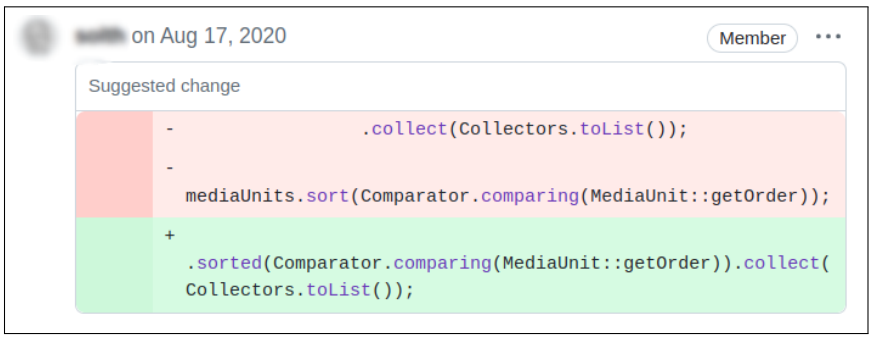


Figure 19 – Example of suggested change to preserve the function but change the logic (8)

.

these cases to be potential understandability improvements.

In summary, we classify a code review comment as suggesting an understandability improvement if it fits within one of the following cases: (i) the intent to improve understandability is explicit in the text; (ii) it is a comment about aspects that do not affect the compilation or the execution of the program, i.e., code comments or code formatting, (iii) the refactoring that it suggests is motivated mainly by improving understandability, as discussed by Fowler (140), and (iv) it does not explicitly state its intent, but it makes a suggestion that is very similar to a comment where the intent is explicit, i.e., the suggestions for improvement require the same change, e.g., in an explicit code review comment, the type `Boolean` is replaced by `boolean` to ensure consistency [key=1201], similarly, in an implicit code review comment, the type `LinkedList` is replaced by `ArrayList` [key=1089] for the same reason. In a code review comment, the reviewer could suggest an improvement that impacts multiple factors, such as performance and understandability. We also consider those types of improvements that impact understandability. Each comment, including those where the reviewer explicitly mentioned that it pertains to understandability or those that are naturally about understandability (as

described in Section 5.2), was classified by at least two authors. Conflicts between answers were resolved through meetings involving at least five authors. In the remainder of this section we explain the types of comments that required additional analysis.

a) Ambiguous intention comments. There are scenarios where it is impossible to discern the intention of the reviewer. For example, in a comment (141), the reviewer suggested "*better to use putString*". This suggestion can be a related to improving code understandability or improving performance. There is no evidence of any intention, neither in replies nor patches. If we cannot establish intention with some confidence, we discard the comment.

b) Repetition comments. Similarly, in other code review comments, we find tiny comments, e.g., *"ditto"* (142) and *"same here"* (143), that do not bring any information about the reviewer's intention. It would be necessary to look at another code review comment to get the context of the comment, and looking for the original comment could be time-consuming. Since these comments account for approximately 1% of our sample, we have decided to disregard them. However, we have retained them in the dataset labeled as "Discard" for future analysis.

c) Question comments. Sometimes reviewers use the code review comments to ask questions to the PR author. On the one hand, there are clarifying questions where reviewers attempt to understand the purpose of the code snippet under review (e.g., "*Why is this needed?*" (144), "*Why protected?*" (145)). In those scenarios, we considered that the comment does not suggest an understandability improvement. On the other hand, we found scenarios where reviewers suggested understandability improvements through questions. For example, in a code review comment, the reviewer asked "*wrong indent?*" (146). This suggestion, phrased as a question, indicates a problem with how the code is formatted. This kind of code review question is considered related to understandability improvements. Previous work (147) has shown that comments with different purposes are often phrased as questions.

d) Multi-purpose comments. In other code reviews, the reviewers suggested changes with multiple purposes. In one example (148), the reviewer commented: "*[...] Suggested change* `if (path.contains("b/")) {` → `if (path.startsWith("b/")) {` *for better performance [...] and preferably change the name of the urlString variable back to just url*" [key=2151]. Although the reviewer suggested a change to improve performance, they also asked to rename an identifier intending to improve the code understandability. This kind of code review comment was considered related to understandability improvements. In our understanding, it is a single comment suggesting multiple different changes.

e) Bot comments. Finally, we found code review comments introduced by bots. Some

of them are suggestions based on linters to improve understandability. For example, the bot named *codeclimate* created the following code review comment (149): "*Method buildMenu has 59 lines of code (exceeds 30 allowed). Consider refactoring*". Although the suggestion of the code review comment is related to understandability improvement, we discarded these code review comments because there was not an evaluation by a human specialist (i.e., a reviewer). Previous work (129) has shown that many recommendations made by linter bots are ignored by reviewers.

5.4 RESULTS

In this section, we present the results of the study, organized in terms of the six research questions.

5.4.1 How often do reviewers ask for code understandability improvements in code review comments? (RQ1)

Table 20 presents the results of this two level classification. Each row indicates the number of comments per scope. The scope of a code review comment can be related to (i) executable source code (CODE), i.e., excluding comments and blank lines and focusing on the instructions that will actually be compiled and executed, (ii) high level API documentation (JAVADOC), (iii) regular source code comments (COMMENT), (iv) a string value or expression with strings (STRING LITERAL), or (v) OTHERS (e.g., annotations). We separate the comments to generate documentation and regular comments because they have different characteristics. JAVADOC is well structured, has special annotations to reference specific elements in the documentation, and is used to explain classes, attributes, and methods, often for readers who are not looking at the code. COMMENT has no prescribed structure and is used to explain the code or the rationale behind it in a more direct manner for readers who are directly looking at a specific part of the code. Also, we analyze STRING LITERALs separately because these literals may give developers hints about program behavior, e.g., in log messages, but they are not comments. Finally, annotations, log levels, and other special cases are analyzed separately. We do not consider annotations to be part of the executable source code because, according to the Java Language Specification[6], they have *"no effect at run time"*.

[6] <https://docs.oracle.com/javase/specs/jls/se12/html/index.html>

Table 20 – Classification of the 2,401 code review comments in terms of whether they pertain to code understandability (Yes) or not (No). The two groups are further divided based on the scopes to which the code review comments refer (code, JavaDoc, string literals, etc.). The percentages for each row add up to 100%.

	Is it about understandability?		
Scope	Yes	No	Discarded
CODE (1,974)	33.49% (661)	63.93% (1,262)	2.58% (51)
JAVADOC (182)	86.26% (157)	12.09% (22)	1.65% (3)
COMMENTS (110)	78.18% (86)	20.91% (23)	0.91% (1)
STRING LITERAL (81)	80.25% (65)	18.52% (15)	1.23% (1)
OTHER (54)	79.63% (43)	20.37% (11)	0% (0)
Total (2,401)	42.15% (1,012)	55.52% (1,333)	2.33% (56)

In the analyzed sample, we found out that 1,012 (42.15%) of the code review comments include suggestions, requests, or discussions about code understandability improvements. This amounts to more than two out of every five code reviews. This result emphasizes that reviewers are worried not only about the correctness and performance of the code but whether the developers are following the best practices of programming and code understandability. We check whether comments suggesting code understandability improvements differ from other comments by comparing their sizes. If one kind of comment tends to be significantly shorter, this may suggest that developers are willing to dedicate less time and effort to it. The 1,012 comments suggesting code understandability improvements tend to be shorter (99.21 characters) than those not related to understandability (117.70 characters). The difference is statistically significant (Mann-Whitney U, $p = 0.000001$) but the effect size is negligible (Cliff's Delta, -0.12). As for the associated source code lines, i.e., the last line of the diff hunk associated, the mean character counts are 50.52 and 54.15 and the difference is not statistically significant (Mann-Whitney U, $p = 0.07$). Overall, we can say that the difference between comments and source lines of code associated to understandability improvements and those that are not is inconsequential.

Considering all code review comments analyzed (2,401), the majority of them (82.2% (1,974 out of 2,401)) are in the scope of CODE. Among the 1,974 code review comments whose scope is CODE, 33.49% (661 out of 1,974) are related to code understandability improvements. The reviewers in these code review comments have asked code submitters to rename identifiers, remove unnecessary code elements, use alternative code constructs, and others. We discuss the content of those code review comments in Section 5.4.2. The results

support other studies (27, 28) that indicate that improving code quality is one of the main goals of code reviews.

When we look only at the code review comments in the scope of JavaDoc, 86.26% (157 out of 182) of them are about understandability improvements. Generally, reviewers ask code change submitters to add JavaDoc documentation (e.g., *"Missing javadoc"* [key=747]), improve it (e.g., *"@link on this and the next line please"* [key=802]), or fix the description (e.g., *"appear → appears"* [key=819]). We found examples of code review comments related to JavaDoc that do not suggest understandability improvements. For example, the code review comment *"what if these objects don't have a common ID or don't have a common event time?"* [key=26] discusses the design solution described on the JavaDoc. In this case, the documentation is not the target of the discussion but the code design related to this documentation.

Similarly to JavaDoc comments, 78.18% (86 out of 110) of the code review comments in the scope of Comments are related to understandability improvements. The reviewers suggest the addition (e.g., *"Please add comments where you see fit."* [key=1287]), improvement (e.g., *"Nit: Typo shake → sake"* [key=182]), or removal (e.g., *"we should remove this comment"* [key=1062]) of comments in the source code. Conversely, some of the code review comments that impact comments do not relate to understandability improvements. They are related to comments with tags such as *TODO* and *FIXME*.

In the analyzed code review comments related to String literal scope, 80.25% (65 out of 81) can be considered code understandability improvements. In those code reviews, the reviewers suggest changes to string literals to improve exception messages (e.g., *"Can you add what the expected and actual partition path was in the exception message ?"* [key=177]), assertion error messages (e.g., *"Can you please change this to be cookieMaxAge cannot be zero?"* [key=2096]), logging messages (e.g., *"I suggest that you can change to* `logger.error("unknown error", e);` *"* [key=2294]), and other method arguments (e.g., *"The number of threads...ingest → This property is deprecated since 2.1.0. The number of threads..."* [key=60]). In contrast, in the code review comments unrelated to understandability improvements, the string literals are arguments used to send messages to the end user (e.g., *"Let's change the description to "List of transports the GAPIC can use..."* [key=1008]), the paths of a resource on disk (e.g., *"I think you have a typo in the filename: the file is: ClangAnalzer.txt the fileName is: ClangAnalyzer.txt"* [key=1259]), and tags used on the header of a protocol (e.g., *"TFB → ServiceTalk..."* in `.addHeader("Server", "TFB")` [key=2157]).

In the OTHER scope, the code review comments are related to annotations and logging (excluding comments that directly refer to string literals). From those, 79.63% (43 out of 54) are related to understandability improvements. The reviewers ask code change submitters to change (including adding and removing) logs and their configuration (e.g., *"Do we need this log here or was it just for debugging?"* [key=2372], *"Use* `log.info`*?"* [key=1896]) and add documentation annotations (e.g., *"*`@PublicAPI` *instead of* `@beta`*"* [key=839]). We also found code review comments that not support understanding the source code where reviewers ask for modifications in annotations that may be related to bugs (*"Does the lazy inject actually work since* `AuditEventLoggingFacade` *is not a lazy bean?"* [key=1545]) or modifications that are related to release notes (*"An entry in CHANGES_NEXT_RELEASE describing the fix...should be included"* [key=2161]), for example.

Answer to RQ1. We found that 42.15% (1,012/2,401) of the selected discussions are related to code understandability improvements. Among them, 65.31% (661/1,012) are about executable source code, almost 24,01% (157+86 out of 1,012) are related to comments, and more than 10.67% (65+43 out of 1,012) are related to literal strings, annotations, and others. Approximately one out of every three code review comments that refers to executable source code is about understandability. Furthermore, four out of five code review comments that refer to elements other than executable code in a program source file pertain to understandability improvements.

Implications. In previous research (117, 118), the researchers identified ways to improve code readability at the level of files by analyzing commit messages. Our findings suggest that researchers and developers can also learn from code reviews to improve code understandability at the level of source code snippets. Moreover, our results demonstrate that, automated tools that enhance code understandability can have significant impact in reducing developers' effort during code reviews. Additionally, our work makes it clear that improving code understandability requires tools that work with both programming language and natural language elements.

Table 21 – The frequency of understandability smells (**what**) found by reviewers and **where** they are in the source code. The understandability smells were found in declaration, access, or implementation of kind of code elements with an asterisks (*).

Understandability smells	Method*	Call	Literal	Class	Attribute*	Conditional	Variable*	Parameter*	Others	Total
Incomplete or inadequate code documentation	38	2	1	11	2	6	2	0	7	69
Bad identifier	26	0	0	8	9	0	6	9	0	58
Complex, long, or inadequate logic	9	18	0	2	3	7	4	1	10	54
Unnecessary Code	10	8	1	2	8	1	2	0	7	39
Inconsistent or disrupted formatting	12	2	1	7	1	5	0	0	6	34
Wrong, missing, or inadequate string expression or literal	0	0	26	0	0	0	0	0	1	27
Inadequate logging and monitoring	2	8	0	0	0	0	1	0	2	13
Missing constant usage	0	0	6	0	0	0	0	0	0	6
Total	97	38	35	30	23	19	15	10	33	300

5.4.2 What are the main issues pertaining to code understandability in code review? (RQ2)

Table 21 shows the frequency of the understandability smell (rows) found in our sample and where they are in the source code (columns), with each cell matching the understandability smell in a specific place in the source code. We try to keep the names of the understandability smells as self-explanatory as possible. Therefore, we sometimes employ longish names such as *"Incomplete or inadequate code documentation"*, instead of more typical smell names such as *"Shotgun surgery"* and *"Comments"* (140). We categorize eight kinds of understandability smells found in 16 places in the source code. More specifically, we encounter the following places (with the number of occurrences in parentheses): Method (97), Call (38), Literal (35), Class (30), Attribute (23), Conditional (19), Variable (15), Parameter (10), Operation (7), Try-Catch (7), Loop (6), Annotation (5), Import (5), Interface (1), Object (1), Package (1). The categories with less than 10 instances of understandability smells were grouped under Others in the table.

The most prevalent understandability smell is INCOMPLETE OR INADEQUATE CODE DOCUMENTATION, which appears in 23% (69 out of 300) of the code review comments. It groups the cases where the concerns are in documentation and code comments. There are comments where reviewers point out concerns related to JavaDoc, inline code comments, and Java annotations. In general, the reviewers indicated missing documentation (e.g., *"Add a comment indicating whether start/end are inclusive or exclusive"* [key=2020]); incorrect documentation (e.g., *"Minor: which is available..."* [key=1115]); incorrect placement of documentation (e.g., *"Think you may have intended to have the below comment here too, or only here since it's the first occurrence"* [key=1724]); and unnecessary comments (e.g., *"What does the comment mean? Or can it be removed?"* [key=321]). Furthermore, the reviewers indicated incorrect or missing documentation about the status of methods using the Java annotation `@Deprecated`. More than half ($38/69 = 55.1\%$) of the occurrences of this understandability smell are related to Method (35 in declaration clause and three in body block) and about 15.9% (11/69) are related to Class. We find more understandability smells about code documentation in these places because, in Java, code documentation is frequently written in methods and classes.

The second most popular understandability smell is BAD IDENTIFIER, which appears in 19.3% (58 out of 300) of the code review comments. The reviewers pointed out concerns related to the content of the identifier name and its style. A good identifier can help better communicate the content or its functionality, e.g., the variable n that represents the person's name is better represented by identifier name. We highlight code review comments indicating typos in the identifier names (e.g., *"Typo in variable name"* [key=811]); parameter names in upper case (e.g., *"Would ... making ... the MBI parameter in createPatientResource() lowercase? Might get mistaken for constants..."* [key=579]); method names that do not explain functionality (e.g., *"This method actually returns the* `Requirements` *class, not the annotation, so better call it* `getArtifactRequirements()` *"* [key=535]); and variable names incompatible with their type (e.g., *"Rename the field as well? Since it's now an executor and not a handler..."* [key=340]). Most of the occurrences of this kind of understandability smell are in Method (24 in the declaration clause and two in the body block), Attribute (9), Parameter (9), and Class (8).

Another understandability smell is COMPLEX, LONG, OR INADEQUATE LOGIC. The reviewers pointed out concerns about complex and long alternatives to writing expressions, code constructs, and other statements. For instance, the boolean expression `map.keySet().size() > 0` could be replaced by an unique method invocation, `!map.isEmpty()`. We find this category

in 18% (54/300) of the observations in our sample. The reviewers have identified redundant method calls (e.g., *"Looks like it could be simplified"* [key=83]); and long method implementation (e.g., *"Maybe you can refactor this logic into a different method, it makes this method quite large..."* [key=1413]). We also found comments suggesting that anonymous inner classes are too verbose (e.g., *"With a lambda, this would be slightly shorter"* [key=58]); and complex `if-else` implementation with repeated code (*"I think a better way to write this method is: [duplicate code within if-else out of if-else statement]"* [key=1878]). Moreover, reviewers indicated concerns related to using API classes, such as using a class as type instead of its primitive type (e.g., *"Why* `Boolean` *and not* `boolean`*?"* [key=1201]). This smell is more frequent in Call (18), Methods (8 in the body and one in the declaration clause) and Conditionals (7).

UNNECESSARY CODE occurs in 13% (39 out of 300) of the code review comments in our sample. This understandability smell involves snippets of source code that can be safely removed without altering the functionality of the code. This goes beyond simpler scenarios, such as textually duplicated or dead code. For example, in the code snippet `V1=2; V1+=1; V1=1`, the two first statements do not influence the final value of `V1` but may still impose a cognitive load on the reader and, in the worst case, may cause her to misjudge the final value of `V1`. Among other elements, the reviewers identified unused imports, classes, methods, variables, and constants (e.g., *"unused constant"* [key=816]); and commented out source code (e.g., *"commented code"* [key=1074]). Other concerns are related to redundant code, such as unnecessary use of `this` (e.g., *"nit: this. is redundant here"* [key=1871]); unnecessary intermediate variable (e.g., *"Could be better to have the full* `clientSession.newRequest()...execute()` *to avoid having* `loadCsv` *variable"* [key=1561]); and semantically duplicate method (e.g., *"Is this necessary given that super.getDependencies does the same?"* [key=1741]). We encounter this smell more often in Method (10 – six in the declaration clause and four in the body block), Call (8), and Attribute (8). These three comprise more than half of the instances in this category.

In 11% (33/300) of the code review comments the reviewers identified INCONSISTENT OR DISRUPTED FORMATTING. This understandability smell is linked to spaces, braces, parentheses, and formatting styles. The formatting elements can help the reader to identify and delimiter the elements in source code, e.g., the use of curly braces (`{}`) to define the scope of a block. This includes comments that identified missing vertical and horizontal space (e.g., *"more spaces please"* [key=683]); missing parentheses to make expression evaluation order explicit (e.g., *"Suggested change* `...requiresMaintenance() && null != u.getTech()`

→ `...requiresMaintenance() && (null != u.getTech())`" [key=1453]); and missing brackets and line breaks. The reviewers also pointed out extraneous vertical and horizontal spaces and parentheses (e.g., *"A little too many parens here"* [key=1446]). Additionally, we run into comments that indicate code formatting that does not match formatting guidelines, e.g., in `if` statements, *"IF statement should be on it's own line"* [key=2024]). These cases of inadequate formatting are most frequently 54.5% (18/33) found in Method (11 – sixin the declaration clause and five in the body) and Class (7).

We find in 9% (27/300) of our sample code review comments related to WRONG, MISSING, OR INADEQUATE STRING EXPRESSION OR LITERAL. This understandability smell pertains to concerns regarding natural language string messages such as those used as arguments in exceptions or logs, as well as scenarios involving the style and typographical correctness of string values. In these comments, the reviewers indicated incorrect words in string messages (e.g., *"... iterable, not iterator..."* [key=2365]); different styles of string message (e.g., *"not supported sounds a bit better than "unsupported""* [key=2287]); and missing string messages (e.g., *"Maybe an explanation here would be nice?"* [key=1345]). Moreover, the reviewers pointed out the string value in a parameter that does not match the code conventions (e.g., *"Suggested change* `@Column(name = "END_TEXT")` → `@Column(name = "end_text")`" [key=1418]); and a typo in a string value (*"Suggested change* `operationId = "aproveInboxItemById"` → `operationId = "approveInboxItemById"`" [key=1642]). By definition, all comments where this smell manifests pertain to concerns found in existing literals during code review. The sole exception to this trend, constituting one instance, relates to the absence of a literal within a throw invocation. In this specific case, we consider that the place of the smell is a Try-Catch node.

The instances of the INADEQUATE LOGGING AND MONITORING smell account for 4.3% (13/300) of the code review comments. This comprises scenarios related to the logs and exceptions statements. These statements communicate to developers information about system behavior at run time, e.g., for debugging and monitoring purposes. This information can also help developers to make sense of the code functionality around these statements. In these comments, reviewers indicated concerns about the usage of logs and exceptions, such as missing log (*"should we log this? just to make investigation easier"* [key=1135]); unnecessary log (*"In my opinion, we do not need this log. WDYT?"* [key=2337]); incoherent log levels (e.g., *"...this is currently logging every second in test. Can this be changed to debug?"* [key=574]); and throwing exceptions of generic types (e.g., *"This is not a very good exception, let's tell*

the developer what is wrong in each case and how they could fix it" [key=919]). Most of the instances of this understandability smell are related to Call (8/13) statements. This smell does not cover issues with text literals used as log messages, as they are instances of the WRONG, MISSING, OR INADEQUATE STRING EXPRESSION OR LITERAL smell.

Finally, 2% (6/300) of the code review comments are about MISSING CONSTANT USAGE. Constants are often used to indicate the purpose of literals with a specific meaning in a program, e.g., using a constant named `PI` instead of its actual value. Therefore, one of their goals is to improve understandability. In these code reviews, the reviewers pointed out the direct use of literal values (e.g., *"It would be better to declare this Strings as constants"* [key=2163]), even when there are constants available (e.g., *"...You should also be able to reuse the constants declared at the beginning of* `ConflictsWithScrutinizer`*"* [key=1687]). By definition, reviewers found these understandability smells in Literals.

Answer to RQ2. In our representative sample of 300 code reviews, we found eight categories of understandability smells. MISSING OR INADEQUATE CODE DOCUMENTATION (69), BAD IDENTIFIER (58), and COMPLEX, LONG, OR INADEQUATE LOGIC (54) comprise more than 60% of the understandability smells in our sample. They are related to 16 code element types in source code. The three most frequent places, i.e., Method (97), Call (38), and Literal (35), account for more than 56% of the code reviews.

Implications. We have discovered a variety of issues affecting code understandability and grouped them into understandability smells. Automated solutions to improve code understandability must deal with these different kinds of issues, with varying levels of context dependence, and tackling multiple modalities: source code as well as unstructured and structured natural language. The results suggest that tools based on LLMs have to be trained on these different languages, as some smells manifest only in Java source code (e.g., UNNECESSARY CODE) whereas others occur in natural language text or identifiers, e.g., BAD IDENTIFIER and INCOMPLETE OR INADEQUATE CODE DOCUMENTATION.

5.4.3 How likely are understandability improvement comments to be accepted? (RQ3)

To answer this question, we analyzed a set of 300 code review comments related to code understandability improvements (the same set analyzed in Section 5.4.2) and added another

Table 22 – What are the odds of the code review suggestion being accepted, given that it is a code understandability improvement?

Contingency table (600)		Understandability Improvement	
		Presence	Absence
Acceptability	Accepted	253	193
	Not accepted	47	107

random sample of 300 code review comments not related to code understandability. The code review comments were classified based on two properties: (i) whether they suggested an improvement for understandability, and (ii) whether the suggested changes were accepted or not. The results of this classification are presented in Table 22. This contingency table allows us to investigate the odds of accepting a code review comment, given that it is an understandability improvement.

We verified whether code review comments that include suggestions for improving code understandability are more likely to be accepted by developers than comments that do not include such suggestions. Our analysis revealed that code review comments that suggest understandability improvements are more likely to be accepted than comments that do not address understandability; the odds ratio is 2.98, with a confidence interval (1.79, 4.99) for a 99% confidence level. We also performed a Chi-Square test to evaluate the relationship between these two variables and found a significant correlation between them. In fact, suggestions for improving code understandability were found to be significantly more likely to be accepted than comments unrelated to understandability ($\chi^2(1, N = 600) = 31.4, p < 0.01$).

Additionally, we conducted a detailed analysis of the code reviews that suggested improvements in understandability. We observed that in three instances where understandability improvements were suggested, the developers accepted that the code needed improvement, but they did not agree with the reviewer's suggestion. Instead, they implemented what they believed was the best alternative without any discussion. For instance, in one code review, the reviewer suggested *"rename* `module_uuid` *to* `assembly_uuid`*"* [key=1935], but the developer renamed it to `assemblyId`. Also, in eight of the analyzed code reviews developers answered that they accepted the suggestion, but we cannot find any change, even when the developer answered *"changed"* [key=1601]. In 13 instances of suggestions of code understandability improvements that were not accepted, the developers argued in favor of their alternative. For example, during a code review, a reviewer suggests initializing a boolean attribute with the

value `false` [key=854], but the developer argues that it is redundant because the default value of a boolean variable is already `false`.

Developers have given a similar level of attention to the understandability smells. For example, there were 51 patches aiming to address BAD IDENTIFIER (Table 23 provides the data for the remaining smells) and 58 instances of that smell (the rightmost column of Table 21). This means that 88% of the instances of this smell resulted in an accepted patch with an improvement, the highest percentage among the understandability smells. Overall, 84.3% (253/300) of the code review comments triggered code understandability improvements. For INADEQUATE LOGGING AND MONITORING, the percentage was 77% (10/13), the lowest among the smells and the most distant from the mean of 84.3%. Calculation of the odds ratio considering the groups where this latter smell was fixed vs. not fixed and where any other smell was fixed vs. not fixed suggests that the difference is not statistically significant (OR=0.60, p=0.46).

Answer to RQ3. Code review comments that suggest understandability improvements are significantly more likely to be accepted than comments that do not address understandability improvements. The majority (83.3%) of the suggestions for code understandability improvements in code review comments were accepted. Only 1% acknowledged the need for improvement but applied another solution. However, in 13% of the cases, the pull request authors did not accept the suggestion. In addition, the smells received similar levels of attention, in terms of the percentage of instances for which an understandability improvement was accepted.

Implications. Developers typically accept suggestions for improving code understandability during the code review process. These suggestions and the changes related to them can be a valuable and trustworthy data for devising automated approaches to improve code understandability.

5.4.4 What code changes are found in understandability improvements? (RQ4)

We examine 253 code review comments from the results of our analysis in Section 5.4.3 to assess improvements in code understandability. We sort and categorize the improvements made to address the understandability smells in these reviews, and the results are presented in Table 23. The patches are organized based on the understandability smells presented in

Section 5.4.2, e.g., UNNECESSARY CODE and BAD IDENTIFIER. The similar patches in each category were organized in groups, e.g., ↪ Unused code. In Table 23, an arrow (↪) represents a group and its name is underlined. We also combine the patches based on the action taken by the developer and we created a second level of indentation on the table to avoid repetition of text, e.g., explain the functionality of a method. Patches appearing three or more times are marked with the clubsuit symbol (♣) to indicate their higher frequency. In the remainder of this section, we discuss the understandability improvements that were applied for each understandability smell.

Table 23 – Understandability smell categories and the improvement patches found in the 253 accepted code reviews. The patches with three or more instances are marked with the symbol ♣. In the PDF version, each category links to an example.

Category ↪ Group or type of improvement patches (# found)

Incomplete or inadequate code documentation (57)

↪ Changes related to JavaDoc: (42)

- Add new JavaDoc block to describe the functionality of a code element (7) ♣
- Add new tag and hyperlink to existing JavaDoc (2)
- Add text to existing JavaDoc to
  - explain the functionality of a method (5) ♣
  - explain internal implementation details of a method (2)
  - improve explanation about code element (2)
  - explain reason for deprecation (1)
- Change text in existing JavaDoc to
  - improve explanation about functionality of a method (6) ♣
  - improve grammar or fix a typo (11) ♣
  - update copyright (2)
  - better explain the purpose of an annotation (1)
  - fix the name of a code element to which it refers (1)
  - make it more concise (1)
- Move a tag of documentation to the conventional place (1)

↪ Changes related to code comments: (12)

- Add code comment to
  - explain an expression (1)

Table 23 continued

- explain the type of an attribute (1)
- explain each part of a test (2)

– Remove TODO code comment for tasks that have already been performed (2)

– Remove a useless code comment (2)

– Change code comment to better explain the associated code element (2)

– Change text in code comment to fix typo (1)

– Transform a code comment into JavaDoc documentation (1)

↪ Add or remove `Deprecated` annotation to update the method status (3) ♣

Bad identifier (51)

↪ Changes related to content: (46)

– Modify an identifier to express the meaning or type of an element (35) ♣

– Modify an identifier to fix a typo (8) ♣

– Modify an identifier to be consistent with a convention (3) ♣

↪ Changes related to style: (5)

– Change the style of an identifier to be camelCase, capitalized, or lowercase (5) ♣

Complex, long, or inadequate logic (44)

↪ Extract method (5) ♣

↪ Extract variable (1)

↪ Rewrite the code to avoid semantic duplicate method call (2)

↪ Move the code element to consistent place (1)

↪ Changes related to usage of APIs: (15)

– Replace method calling chain by existing API (6) ♣

– Use a different type to be consistent with other parts of the code (4) ♣

– Replace usage of an external API by an internal one (2)

– Replace a set of statements by direct method in existing API (2)

– Replace Lists.newArrayList by Arrays.asList (1)

↪ Changes related to expressions: (11)

– Replace call to static method by instance method call (3) ♣

– Replace boolean expression by an alternative direct method (2)

– Replace an expression by an alternative expression (2)

– Replace the use of a parameter by an expression with another parameter (1)

Table 23 continued

- Replace indirect expression in parameter by direct value (1)
- Replace an expression that creates object by existing attribute (1)
- Invert the boolean expression to simplify it (1)

↪ Changes related to constructs: (6)

- Replace anonymous inner class by lambda (2)
- Replace an imperative logic by functional (2)
- Replace the while and switch structure by an alternative switch with default (1)
- Replace short-circuit in stream by an alternative reducing operation (1)

↪ Changes related to imports: (3)

- Expand the imports (2)
- Remove the library name reference in fully qualifier name (1)

Unnecessary code (33)

↪ Unused code: (17)

- Remove non-referred constant, import, class, method, or variable (12) ♣
- Remove commented out code statements (5) ♣

↪ Redundant code: (16)

- Remove duplicated code or processing (10) ♣
- Remove null constants (1)
- Remove type parameter for inferrable type (1)
- Remove `this` (1)
- Remove imported type from fully qualified names (1)
- Inline temporary (2)

Inconsistent or disrupted formatting (29)

↪ Space usage: (18)

- Add or remove horizontal spacing (9) ♣
- Add or remove vertical spacing (9) ♣

↪ Formatting element visibility: (3)

- Add braces in a single statement block (2)
- Add parentheses to compound logical expressions (1)

↪ Formatting style: (8)

- Add line break to a long statement line (1)

Table 23 continued

– Remove parentheses in logical expression operands and move them to separated lines (1)
– Change order of boolean expression operands to make `null` the second one (1)
– Move code elements to their own line (3) ♣
– Move `catch` declaration to the same line of its associated `try`'s closing block brace (1)
– Move methods to be in call ordering (1)
Wrong, missing, or inadequate string expression or literal (24)
↪ Add a message to an exception (1)
↪ Changes related to a natural language string literal: (18)
– Fix an incorrect string literal (6) ♣
– Replace a string literal by a different meaning or a synonymous (6) ♣
– Extend a string literal (3) ♣
– Change literal to adhere to project standards (3) ♣
↪ Changes related to a string magic value: (5)
– Fix an incorrect value (2)
– Change magic value to adhere to project standards (3) ♣
Inadequate logging and monitoring (10)
↪ Add logging code (2)
↪ Remove log (5) ♣
↪ Change log settings (2)
↪ Change generic exception to a specific (1)
Missing constant usage (5)
↪ Replace hardcoded value by a new constant (3) ♣
↪ Use a constant from a library (2)

Incomplete or inadequate code documentation. We find 57 patches that add, remove, enhance, or fix JavaDoc and code comments in the source code. The majority of them (11) are patches that improved the grammar or fixed a typo in existing JavaDoc (e.g., *"memory which available"* → *"memory which is available'* [key=1115]). Also seven patches added JavaDoc documentation to a method or class where it was absent (e.g., methods [key=756] and classes [key=2244]). In other patches related to JavaDoc, the developers enhanced the description of code elements (e.g., *"Time spent for waiting a task to be completed."* → *"This time spent*

while waiting for the task to be completed is being recorded in this counter." [key=151]) and added tags and hyperlinks (e.g., *"@link JobModel"* [key=294]). In terms of changes applied to code comments, there are patches that added code comments to explain the choice of an attribute type [key=2180], removed TODO code comments for completed tasks [key=2387], and transformed a code comment into JavaDoc [key=1625]. Finally, three patches added or removed the `Deprecated` annotation to document the status of a method [key=2263].

Bad identifier. 51 patches changed an identifier to better communicate what it means in the source code. These patches involved changes regarding both the content and style of the identifiers. To address concerns related to the content of the identifiers, the majority (35) of the patches focused on clarifying their meaning (e.g., `String getDates` → `String getDateRange` [key=1042]) and expressing the identifier's type (e.g., `RequestExecutor requestHandler` → `RequestExecutor requestExecutor` [key=340]). Additionally, eight patches fixed typos (e.g., `verifyEntityTypeMigrationInValidEntities` → `verifyEntityTypeMigrationInvalidEntiti` [key=1107]) and three adjusted the identifiers to adhere to the conventions adopted by the project (e.g., `boolean isActive` → `boolean active` [key=453]). Furthermore, we observed five patches that aimed to improve the style of the identifiers, such as converting them to lowercase [key=579] or camelcase [key=1753].

Complex, long, or inadequate logic. 44 patches extracted, moved, or replaced complex, lengthy, or inconsistent code in an attempt to simplify it. Among these, five patches extracted parts of a method to reduce its length (e.g., extracting a method). Also, developers applied 15 patches that replaced an usage of an external API or adopted an external API to simplify complex code, such as replacing a chain of method invocations by an unique API method invocation [key=1700]. In eleven other scenarios, developers replaced expressions in source code by an alternative that better communicates the functionality of the code, e.g., replacing boolean expression by a call to method `isEmpty()` [key=626]. Additionally, developers applied patches that replace the use of a construct in six cases, such as replacing an anonymous inner class by a lambda [key=1971] or replacing a lambda with conditional by an `if` statement [key=1892]. Finally, the developers applied three patches to organize the use of APIs in a source code file, such as expanding imports instead of using wildcards [key=2203], and adding an `import` statement to avoid the use of the fully-qualified name of a class (`TileSourceManager.TileSourceTemplate template` → `import ...TileSourceManager; private TileSourceTemplate template` [key=1779]).

Unnecessary Code. A total of 33 patches eliminate code that has no impact on the execution.

Patches that remove unused code include twelve instances of code elements that are not being referred to (e.g., *"unused constant."*) and five instances of commented out code [key=444]. Also, the developers applied patches that removed duplicated code (e.g., equivalent methods [key=325]) or processing (e.g., iteration happening twice on the same list [key=536]) ten times, and different constants with value `null` [key=949]. There are also two patches that inline temporary variables [key=1561].

Inconsistent or disrupted formatting. 28 patches organize and standardize code formatting. These patches manage the use of formatting characters such as space and braces, as well as formatting styles. In 17 of the patches, the developers added space between different code elements (e.g., *horizontal space* [key=493] and *vertical space* [key=2050]); and removed space between related code statements (e.g., *for declaration and first statement of the block* [key=1660]) or extra space between code elements in the same expression (e.g., *between the point and the invoked method in an expression* [key=1754]). Additionally, developers added other structuring characters (e.g., braces and parentheses) to highlight the region of a code element (e.g., *block* [key=1078] or *operand* [key=1453]). Finally, in eight cases developers changed the order or moved code elements aiming to apply a formatting style, such as keeping the code elements in their own lines (e.g., *body of* `if` *statement in its own line* [key=2024]) and arranging method declarations [key=1860].

Wrong, missing, or inadequate string expression or literal. 24 patches address concerns related to incorrect, missing, or inadequate string expressions or literals. These patches involve adding, fixing, extending, or changing strings to better suit the code context and improve code understandability. In six of these patches, the developers replaced a natural language string literal by an alternative with a different meaning (e.g., *assertEquals("not_equals"... → assertEquals("Expected and actual values should be the same!"...* [key=786]); or a synonym (*...are unsupported → ...not supported* [key=2287]). Furthermore, in six other patches developers fixed incorrect words in string expressions (*Can not set min... → Can't set min...* [key=1149]) and string value (*aproveInboxItemById → approveInboxItemById* [key=1642]). Additionally, developers changed five direct string values to adhere to project standards [key=2318] or fix values [key=39] to alternatives that adhere to project standards.

Inadequate logging and monitoring. Developers have fixed concerns related to logging and exceptions in ten patches. In these patches, the developers added new logging instructions in two instances to facilitate the understanding of the execution of a method or statement (e.g., adding logging after processing a list [key=1135]), and removed unnecessary logging in five in-

stances (e.g., removing log for local debugging with `System.out.println` [key=1021]). Moreover, developers changed logging levels (e.g., from *fine* to *finest* [key=1751]) and simplified parameters of formatted text when logging (use direct value instead of variable [key=1173]). Additionally, in one case [key=919], a patch replaced `RuntimeException` by a more specific type, `EncodingException`.

Missing constants. In five patches developers created new constants or used available ones to replace literals. They created new constants to replace literal arguments in three instances (e.g., `bodyParams.put("username", user)` → `bodyParams.put(PARAM_USERNAME, user)` [key=2163]) and replaced literals by library constants in two [key=590].

Answer to RQ4. We found 253 patches where developers added, removed, changed, replaced, and moved code elements to improve code understandability. The most common type of understandability improvement was modifying an identifier to express the meaning of a code element (35). This was followed by removing unused constants, imports, classes, methods, or variables (12), changing text in existing JavaDoc to improve grammar or fix a typo (11), and removing duplicate code or processing (10).

Implications. These results show that there is considerable variation in understandability improvements; some can be easily supported by linters (e.g., removing unused elements) whereas others require approaches that may be challenging even considering recent advances in machine learning (e.g., removing functionally equivalent code that is not textually similar). They emphasize how diverse tools can be employed to address the different understandability smells.

5.4.5 To what extent are accepted code understandability improvements reverted? (RQ5)

We analyze whether or not developers kept the 253 patches analyzed in Section 5.4.4 throughout the history of the patched files. The results are shown in Table 24. We observe that none of the patches implementing code understandability improvements were reverted in the pull requests where they were applied. However, we come across seven instances where there are commits with reversions that were undone in subsequent commits of improvement patches in the same pull request. For example, in one case [key=1471] the improvement patch followed a suggestion to remove `@Deprecated`, but in the following commit this annotation

Table 24 – Out of the 253 accepted patches addressing understandability smells, the frequencies of patch merging, reversion within the pull request where it was accepted, permanence in the code based until the last version of the file, and reversion at any point in the history of the project.

Patch status (# analysed patches)	Yes	No
Patch merged (253)	244 (96.4%)	9 (3.6%)
Patch reverted in the pull request (9)	0 (0%)	9 (100%)
Patch in the last version of the file (244)	199 (81.6%)	45 (18.4%)
Patch reverted in the codebase history (45)	2 (4.4%)	43 (95.6%)

was added again and, later, in subsequent commit, it was removed as suggested initially by reviewer. Also, we discover that 96.4% of the improvement patches were successfully merged. The remaining patches (nine) involved code snippets that were superseded by subsequent commits within the same pull request. For example, a patch added JavaDoc to a method and later both the method and the associated JavaDoc were removed [key=1338], a method named `inAppDisplayhold` was renamed to `displayPaused` in a patch and later renamed to `autoDisplayPaused` [key=1203], or the patched file was removed entirely [key=416].

We also evaluate whether the (244) patches implementing suggested understandability improvements and integrated into codebase were reverted at some point in the history of the codebase, after the original pull request and the associated code review were closed. Out of the total 244 patches, 81.6% (199/244) are present in the latest version of the codebase, although in 20 cases, the developers renamed or moved the patched files. However, we cannot find the patch in the last version of the codebase in 45 patches. Among these, the developers replaced the patch with an alternative solution in nine observations. For instance, a subsequent patch changed a snippet with functional implementation (commit a0c1cbf9746ed9486f08ce4aa2376828e846c786) by an alternative imperative implementation [key=1453]. In 20 patches, the developers removed the code snippet that contained the patch, e.g., the patch included the deprecated annotation and deprecated methods were deleted [key=2263]. Still, in 14 patches, the developers deleted the entire patched file, e.g., AssemblyJsonServlet.java deleted [key=1935]. Finally, the patches were reverted and kept reverted until the lattest version of the codebase in two of these 45 observations. One such case involved the removal of an unnecessary class [key=2073], but the file was added back into the codebase and later renamed.

As a corollary result of this analysis, we observe that most of the analyzed patches implementing code understandability improvements (96.4%) were integrated into the codebase,

whereas Kalliamvakou et al. (127) reported only 44% of pull requests in GitHub (in general) being merged. This stark difference may highlight the high acceptance rate of the reviewers' suggestions for code understandability improvements. It must be taken with a grain of salt, though, as the study of Kalliamvakou et al. was published in 2014. In addition, only two out of 253 patches were reverted and maintained their reversion in the latest codebase version.

Answer to RQ5. After analyzing the history of 253 patches, we discovered that 78.7% (199 out 253) were present in the latest version of the codebase, while the remaining ones were either removed or replaced by alternative solutions. Additionally, we observed that none of the patches were reverted during the pull request. Finally, we found 2 cases that were removed later during the history of the codebase.

Implications. The results in this section suggest that understandability improvements tend to be stable, i.e., it is unlikely that an improvement will later disappear within a project. Thus, training machine learning models to identify code understandability smells and improving their understandability, one of the goals of code reviews, may yield better results than automating code review activities in general (150, 151).

5.4.6 Do linters contain rules to detect the identified code understandability smells?(RQ6)

We investigate the coverage of the code understandability smells identified in our study by rules implemented in Spotbugs, PMD, SonarQube, and Checkstyle. To investigate if any of these linters' rules can flag the 253 understandability smells presented in Section 5.4.4, we manually checked each observation of an understandability smell against each tool to determine if it could detect it. For example, for an observation of UNNECESSARY CODE related to the use of `import` clauses, we inspect the rules *UnnecessaryImport*, *Unnecessary imports should be removed*, and *UnusedImports* in PMD, SonarQube, and Checkstyle respectively, which could detect the *unused import* pointed out by the reviewer. There is no rule to detect this particular instance of UNUSED CODE in Spotbugs. We also considered occurrences where it may be necessary to configure a rule for the linter to detect it. For instance, the rule *MissingJavadocMethod* of Checkstyle needs configuring the scope to warn about missing JavaDoc in methods with "package" visibility (otherwise, it only reports issues with "public" ones). Table 25 presents the number of occurrences of each understandability smell that each linter could detect. The rows represent each type of understandability smell, while the columns

Table 25 – The frequency of understandability smells that are covered by linters.

Understandability smells (# occurrences)	Spotbugs	PMD	SonarQube	Checkstyle	Coverage
Incomplete or inadequate code documentation (57)	0 (0.0%)	8 (14.0%)	5 (8.8%)	12 (21.1%)	12 (21.1%)
Bad identifier (51)	1 (2.0%)	5 (9.8%)	2 (3.9%)	5 (9.8%)	5 (9.8%)
Complex, long or inadequate logic (44)	0 (0.0%)	7 (15.9%)	9 (20.5%)	6 (13.6%)	14 (31.8%)
Unnecessary Code (33)	6 (18.2%)	10 (30.3%)	15 (45.5%)	6 (18.2%)	19 (57.6%)
Inconsistent or disrupted formatting (29)	0 (0.0%)	3 (10.3%)	12 (41.4%)	20 (69.0%)	20 (69.0%)
Wrong, missing, or inadequate string expression or literal (24)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)
Inadequate logging and monitoring (10)	0 (0.0%)	2 (20.0%)	2 (20.0%)	0 (0.0%)	2 (20.0%)
Missing constant usage (5)	0 (0.0%)	0 (0.0%)	2 (40.0%)	2 (40.0%)	2 (40.0%)
Total (253)	7 (2.8%)	35 (13.8%)	47 (18.6%)	51 (20.2%)	74 (29.2%)

correspond to each linter. The column *Coverage* presents the number of instances of each smell that can be identified by at least one linter.

Our analysis indicates that 74 out of 253 occurrences of understandability smells can be detected by any of the linters. This highlights wasted human effort, as automated tools could be pointing these issues out. On the one hand, Checkstyle and Sonarqube are responsible for detecting more than half of detectable occurrences. On the other hand, Spotbugs detects only seven occurrences overall, where six are related to unnecessary code. Furthermore, we did not find rules that could to detect the occurrences related to Wrong, missing, or inadequate string expression or literal, either because the tools do not inspect string literals or because they are natural language text.

The understandability smell that can be more often detected in our dataset by the linters is Inconsistent or disrupted formatting. Rules such as *NeedBraces* (Checkstyle), *Lines should not be too long* (SonarQube), and *UselessParentheses* (PMD) can detect the majority (69%) of the occurrences of this smell. Additionally, these tools acknowledge that different projects have different styles. For example, in a code review comment, the reviewer suggested that *"bracket [left curly brace] should be on the next line"* [key=1487], i.e., the left curly brace should be in beginning a new line instead of the end of current statement line. Accounting for the possibility of some projects using left curly braces at the end of the current line, SonarQube contains rules for both cases, *An open curly brace should be located at the beginning of a line* and *An open curly brace should be located at the end of a line*.

The two most popular understandability smells, BAD IDENTIFIER and INCOMPLETE OR INADEQUATE CODE DOCUMENTATION, are the second and fourth less detectable by linters, respectively. The majority of the occurrences of these understandability smells are associated to natural language text (e.g., typo in identifier and issue in content of Javadoc). The cases of BAD IDENTIFIER detected by the linters are those related to style (e.g., parameters starting with capital letter [key=579]). In these cases, rules such as *FormalParameterNamingConventions* (PMD), Nm: *Field names should start with a lower case letter (NM_FIELD_NAMING_CONVENTION)* (Spotbugs), and *LocalVariableName* (Checkstyle) can be used to detect them. In the case of INCOMPLETE OR INADEQUATE CODE DOCUMENTATION, there are only rules for missing documentation, e.g., *CommentRequired* (PMD); missing or incorrect order of tags in Javadoc, e.g., *WriteTag* (Checkstyle); and TODO code comments, e.g., *Track uses of "TODO" tags* (SonarQube). Many of the instances of these smells require in-depth understanding of the associated elements, e.g., to produce an identifier that better represents the purpose of a method or to improve an explanation about an element. At the same time, the most common change applied to JavaDoc documentation in our study is fixing typos (Table 23), which could be automated.

Among the remaining 179 out of 253 instances of understandability smells, we identify occurrences that could be covered by the investigated linters by performing small adaptations to existing rules. For example, the linters could cover the *missing parentheses* (e.g., key=1453) by creating an inverse rule to the *UselessParentheses* rule of PMD or "*UnnecessaryParentheses*" rule of Checkstyle. This new rule could suggest the inclusion of additional parentheses in boolean expressions to highlight evaluation order. Additionally, the linters could identify opportunities to use primitive types instead of the corresponding boxed types (e.g., key=1201). As another example, the linters could detect when a exception is created without a message (e.g., key=1345). However, other instances of understandability smells would require more sophisticated analyses. For example, to detect the occurrences of incomplete documentation, unnecessary logging messages, confusing string messages, inadequate use of method call chains, and missing logging, it would be necessary to understand the context of the source code, the culture of the project, the intention of the developers, predict the system behavior at run time, or to work with both Java and natural languages.

Answer to RQ6. We found 74 out of 253 occurrences of understandability smells that could be detect by the linters. Some of the remaining 179 occurrences of understandability smells could be detected by creating new rules in these linters, while other occurrences require a deeper and context-dependent analysis that goes beyond what linters typically do.

Implications. The four linters combined have rules to cover less than 30% of the instances of understandability smells from our dataset. This suggests that developers waste effort during code reviews by alluding to issues that existing automated tools could detect. Among the cases that the linters cannot detect, many of them require an in-depth "understanding" of the program under analysis or the context in which it is inserted. In particular, current linters are unable to cope with code understandability issues related to natural language. Some of these cases present opportunities to devise new tools, e.g., to better summarize the intent of a code element or to give it a more illustrative name. Others could be addressed by simple improvements to current linters, e.g., pointing out typos in documentation, identifiers, and string literals. This also applies to rules related to other aspects, such as messages in exceptions and the use (or not) of boxed types.

5.5 THREATS TO VALIDITY

In this section, we discuss the threats to the validity of our study.

Selection of projects. We constructed our dataset by mining software repositories from GitHub. However, researchers must be cautious about potential pitfalls (127). To ensure data quality, we applied filters to exclude personal projects, repositories with small contributor groups, and forked repositories. Moreover, we utilized a heuristic to select projects with an active code review process during a specific period. Also, we manually analyzed each remaining repository to guarantee the relevance of the dataset, excluding non-software repositories. Additionally, it is possible that some projects may have configured an ASAT (Automated Static Analysis Tool) to prevent certain issues related to code understandability from being introduced into the codebase. However, since we did not assess whether any project adopted this approach, it is possible that some understandability smells may not have been identified by our study.

Size of the sample of code review comments. Before analyzing our sample of 2,401 code review comments to determine whether they are about code understandability improve-

ments, we explored various natural language processing techniques, such as LDA, GSDMM, LSA, and LDA Guided, to extract topics and create clusters for both understandability and non-understandability improvements. Unfortunately, we encountered challenges in obtaining satisfactory results due to two main reasons: (i) our training sample was relatively small, both in terms of the number of comment instances and the length of the text; and (ii) there were significant similarities between comments related and unrelated to code understandability. This problem has been reported by previous work as well (152, 150).

Manual analysis and subjectivity. We conducted a study where we manually evaluated 2,401 code review comments. Manual categorization may result in bias due to different interpretations by human coders. Some comments that were analyzed lack precision and clarity. This could affect the validity of any analysis or decision-making based on the text. Additionally, it is possible that there is overlap between different smells. In these cases, we analyze the reviewers' comments to understand the intent of the proposed change. Each comment was independently classified by at least two authors, and whenever a disagreement was encountered, up to five discussed the classification. We also held sessions to discuss dubious comments and to align the classification process. Furthermore, we analyzed the source code to which each comment refers and subsequent changes to it that may have been triggered by the comment.

We also conducted a manual analysis of potential rules in four linters that could detect issues highlighted by reviewers. For each instance of an understandability smell that we analyzed, we searched through a set of 1,315 rules to find matching ones that could identify the issue. Some of these rules can be configured to detect the problem, and we considered all possible configurations. However, we decided not to include rules that relied on regex patterns, due to their specificity. It's important to note that there may be some rules that we missed during this process, which could result in some issues being classified as not covered by linters.

Code review bots. During the code review process for pull requests, we came across bots that inspect the source code, i.e., bots that work as linters. These bots, like `vaadin-bot` and `codeclimate`, identify potential issues based on pre-configured rules. For instance, `vaadin-bot`[7] found a deprecated method annotation and suggested removing it in the future, while `codeclimate`[8] commented on a method that exceeded the allowed number of lines and suggested refactoring. `Codacy`, `Lint-staged`, and `Prettier` are examples of popular bots that review the source code searching for violations of their pre-configured rules. We decided not to investigate these

[7] <https://github.com/vaadin/flow/pull/8409#discussion_r429865662>
[8] <https://github.com/triplea-game/triplea/pull/7241#discussion_r460364607>

bots specifically, considering that their rules are similar to the linters studied previously. For example, codeclimate leverages SonarJava[9], also used by SonarQube, to verify Java programs.

External Validity. This study focuses exclusively on open-source projects, and as such, the findings may not be fully representative of all software development contexts. To address this concern, we made efforts to include a diverse range of software projects in our analysis. Nonetheless, it is essential to acknowledge that the identified code understandability smells and corresponding understandability improvements might not be universally applicable across all programming languages, frameworks, development environments, or development teams. Furthermore, a limitation inherent to using Github is that many engineered projects in the platform are mostly developer-centric, e.g., libraries, frameworks, etc. This means that many sub-domains, such as games and mobile apps, are under-represented. Additionally, our study focused on code reviews conducted in 2020, which may have been impacted by COVID-related issues. Therefore, it is unclear how much we can generalize the findings. In our analysis, we did not take into account the experience level of the developers and reviewers. This means that we may have factored in the opinions of inexperienced individuals. However, we carefully selected a diverse sample group that included many developers and reviewers. This helped to minimize the impact of this issue.

5.6 RELATED WORKS

In this section we examine related work. We start by contrasting the understandability smells and improvements we have identified in practice with recommendations and experimental results reported in the literature (Section 5.6.1). This helps us identify mismatches and accordance between research and practice, as well as research gaps. We then review previous work discussing models to assess code understandability (Section 5.6.2), following up with a few papers that have investigated manifestations of confusion and reactions to lack of necessary information during code reviews (Section 5.6.3).

[9] <https://docs.codeclimate.com/docs/sonar-java>

5.6.1 Coding alternatives for understandability: literature vs. practice

We found in the literature many studies about code understandability. Among these studies, the researchers investigated the best code alternatives for code understandability. Oliveira et al. (57) conducted a systematic literature review that categorized and organized studies that used empirical evaluation to compare code alternatives for formatting elements. For instance, Gopstein et al. (25), Medeiros et al. (18), and Langhout and Aniche (20) examined the impact of explicit block delimiters (i.e., curly braces in Java) compared to the omission of block delimiters when possible. Furthermore, we found studies that compared other code elements, such as monolithic expressions compared to multiple shorter expressions (16). We aim to compare our findings from practical experience with what we gathered from these studies.

Our focus in this work is on understandability improvements, i.e., situations where existing code exhibits an understandability smell and is replaced by an alternative solution that is more understandable. Thus, in this section we focus on the five understandability smells where, to the best of our knowledge, alternative solutions were compared in existing studies from a human-centric perspective, e.g., based on the preference or performance of study participants: Bad identifier, Unnecessary code, Inconsistent or disrupted formatting, Complex, long, or inadequate logic, and Missing constant usage.

Bad identifier. Chaudhary and Sahasrabuddhe (68) evaluated the impact on code understandability of using identifier names unrelated to the program domain. This study concluded that the identifier names with meaning related to the problem domain facilitate understanding. In this direction, the practitioners improve the identifier names to better match their intent and type. Santos and Gerosa (16) compared using fully-qualified names and `import` clauses with shorter type names. They found out that participants preferred to avoid fully-qualified names and add `import` clauses, which is what the practitioners in our study prefer. Schankin et al. (53) investigated whether a more descriptive name (e.g., *singleWordParts*) is easier to read than a single-word one (e.g., *parts*). They found that a descriptive name was the best in one scenario, and there was no difference in the other. The practitioners also considered improving understandability with descriptive names. Sharif and Maletic (102) and Binkley et al. (66) investigated the effect of the style of the identifier (underscore and camel case) on code understandability. Sharif and Maletic found that the use of underscore to separate parts of an identifier improves understandability, whereas Binkley et al. found out that camel case

works better. For practitioners in Java, using the camel case style for non-constant identifiers improves code understandability.

Unnecessary code. Gopstein et al. (25) and Torres et al. (125) investigated whether the presence of dead, unreachable, or repeated code (e.g., `v1 = 1; v1 = 2`), leads to misunderstanding in C and JavaScript programs, respectively. Both studies found out that the presence or absence of this pattern did not significantly impact code understandability. In contrast, practitioners seem to believe this is an important issue in practice: 13% of all the 253 analyzed code review comments pertain to UNNECESSARY CODE. The practitioners removed the unnecessary code, commented out source code, and duplicated code to improve code understandability.

Inconsistent or disrupted formatting. Santos and Gerosa (16), and Sampaio and Barbosa (40) evaluated whether different conventions in terms of vertical and horizontal space between the code elements impact code understandability and did not find significant differences. Conversely, practitioners consider that spacing is an issue worth raising during code review. Out of the 253 analyzed comments, 17 (6.7%) pertain to spacing. Sometimes, they remove space, and others add space. Sykes et al. (82), Sampaio and Barbosa (40), Gopstein et al. (25), Langhout and Aniche (20), Torres et al. (125), and Medeiros et al. (18) investigated the presence or omission of the block delimiter (e.g., curly braces in Java) in blocks with one statement. Gopstein et al., Torres et al., Langhout and Aniche, and Medeiros et al. found that the presence of the block delimiter improves code understandability (in C, JavaScript, and Java). The other researchers did not find any difference. The practitioners preferred to use block delimiters even in a one-statement block. Geffen and Maoz (77) investigated the effect of ordering methods in understanding the code. For some cases ordering considering the order of calling and the connectivity between the methods is the best for code understandability. The practitioners consider that the calling order improves code understandability.

Complex, long, or inadequate logic. Gopstein et al. (25), Medeiros et al. (18), Torres et al. (125), Costa et al. (153), and Langhout and Aniche (20) investigated which ones promotes better understandability: the conditional operator or `if` statements, in C, JavaScript, Python, and Java small programs. Gopstein et al. and Medeiros et al. found that `if` statements improve code understandability (in C programs) whereas Langhout and Aniche, Costa et al., and Torres et al. did not find a statistically significant difference. The practitioners considered both `if` statement and ternary operator to improve code understandability in distinct scenarios. Lucas et al. (154) compared the use of lambdas and anonymous inner classes based on the opinions

of 28 experienced Java developers. Their results are aligned with our findings when examining code review comments: practitioners prefer lambda expressions. Most of the instances of the COMPLEX, LONG, OR INADEQUATE understandability smell are either too specific to be studied in a more general context, e.g., object creation that can be avoided by using an existing attribute, or, to the best of our knowledge, have not been studied, e.g., using `try` statements that manage resources, instead of `try` with resources.

Missing constant usage. Gopstein et al. (25), Langhout and Aniche (20), and Torres et al. (125) evaluated the effect of using constant variables instead of direct values on code understandability. They did not find a statistically significant difference. The practitioners created and used constants instead of direct values to improve code understandability.

5.6.2 Estimating code understandability

In this section, we present approaches aiming to provide a quantitative summary of the level of understandability of a code snippet. In addition, some of these studies have classified the different types of code understandability improvements.

Piantadosi et al. (116) investigated the frequency, methods, and reasons for changes in code understandability during the evolution of open-source software. Based on a code understandability state-of-the-art metric, they built a model to define states (non-existing, other-name, readable, and unreadable) in which a file can be in at a given time. Using this model, they analyzed the transitions related to code understandability in the commit history of 25 open-source software projects. By analyzing more than 340,000 understandability transitions, they discovered that code understandability rarely changes during the evolution of the project. In a manual analysis of 57 understandability transitions, they found that improving or decreasing code understandability is mostly unintentional. Additionally, they identified the type of software maintenance (adaptive, perfective, or corrective) that caused changes in code understandability.

Fakhoury et al. (117) investigated whether state-of-the-art understandability models are able to capture readability improvements as explicitly tagged by open source developers in commit messages. The authors analyzed 548 commits from 63 engineered Java projects that explicitly mention understandability improvements in the commit message. They then used three state-of-the-art readability models to assess the change brought by each of those commits

to the readability of the code. The authors found that the understandability models were not able to capture readability improvements in most cases. Additionally, they used static analysis tools to investigate what type of understandability changes were applied in the commits related to code understandability and non-understandability. They discovered that understandability commits frequently fix concerns related to imports, white spaces, and braces, whereas non-understandability commits tend to introduce more of such concerns.

Roy et al. (118) proposed a model to detect code understandability improvements made by developers in real-world scenarios using commit histories from open-source software projects. To build their model, the authors extended the dataset of commits related to code understandability improvements created by Fakhoury et al. (117). They also included commits unrelated to code understandability improvements to create an oracle. The authors employed seven source code analysis tools to collect source code metrics before and after each commit at the file level. By comparing these metrics, they identified key differences that served as the features for their model. The model achieved a precision of 79.2% and a recall of 67% on the test set.

Buse and Weimer (30) proposed a method for measuring code understandability based on human notions of understandability. Their approach involved collecting feedback from 120 human annotators on their understanding of 100 code snippets and correlating this feedback with specific features of the source code. This data was then used to train a machine learning algorithm to classify code understandability. The results showed that the model was able to accurately predict understandability judgments with an 80% success rate. Posnett et al. (93) proposed a simplification of Buse and Weimer's (30) model. They leverage source code size metrics and Halstead metrics (155). Using the dataset of Buse and Weimer (30) to train and evaluate their model, they obtained better results using just three features in small code snippets: volume, entropy and number of lines.

Scalabrino et al. (86) analysed the correlation between 121 metrics related to code, documentation, and developers and six different proxies of code understandability, based on 444 human evaluations. The study revealed that none of the individual metrics strongly correlated with the proxies for code understandability. However, combining multiple state-of-the-art metrics resulted in models that showed a slight improvement in the regression performance of two proxies, Timed Actual Understandability (Correlation: +0.07; Mean absolute error: -0.02) and Actual Understandability (Correlation: +0.02; Mean absolute error: +0.00).

Lavazza et al. (156) evaluated whether code measures are correlated with code under-

standability. They conducted an empirical study with students who performed maintenance tasks. The results showed that a code comprehensibility model cannot rely solely on measures of code structure. The obtained models were not very accurate, with the average prediction error being around 30%.

These studies (30, 93, 86) have introduced models to quantitatively estimate the degree of code understandability or to detect code understandability improvements (116, 118). In contrast, our work aims to delve into the code constructs and practices that enhance code understandability, specifically exploring code alternatives suggested by code quality specialists during software development. Additionally, we manually identified code understandability changes using code review comments rather than relying on an automatic approach based on metrics or keywords (116, 117, 118). Furthermore, we categorized code understandability smells based on the associated issues and solutions instead of categorizing them in terms of software maintenance types (116) or rules of static analysis tools (117). Finally, our work acknowledges that different projects have different rules, priorities, and development cultures and these differences have an impact on what constitutes readable code.

Some other studies (157, 158, 159) have proposed techniques for automating assessment through advanced methods, including convolutional neural networks. Mi et al. (157) introduced IncepCRM, an innovative model employing Inception architecture, which autonomously extracts multi-scale features from code, aided by human annotations to enhance accuracy. In a distinct approach (158), an 83.8% accuracy is achieved, surpassing prior machine-learning models. This is realized by transforming source code into matrices for convolutional neural networks, culminating in DeepCRM, an architecture of three networks trained on diverse preprocessed data. Similarly, in another study (159), a graph-based method is adopted, parsing source code into a graph containing abstract syntax tree (AST), along with control and data flow edges, preserving semantic structural insights. Subsequent conversion of graph node source code and type details into vectors enables the extraction of program graph features. This approach attains 72.5% and 88% accuracy in three-class and two-class classifications, respectively, thereby surpassing prevailing machine-learning models tailored for code readability. Our analysis of inline code reviews, focusing on developers' perspectives regarding code understandability, provides valuable insights for the referenced works. The research conducted by Mi et al. (157) could further benefit by incorporating insights from our study to refine their IncepCRM model. Our findings might aid in identifying specific aspects of code readability highlighted during reviews, enabling more effective feature extraction for multi-scale represen-

tation. Similarly, the strategy proposed by the author in (158) could gain from our insights by considering the nuances of developer concerns about code understandability in the design of novel representation strategies. Additionally, our study could complement the graph-based approach explored in (159) by offering a real-world perspective on how developers' concerns impact code readability. Integrating our unique dataset and analytical framework could enhance these approaches, contributing to advancements in code readability evaluation.

Vitale et al. (160) advocate a comprehensive methodology aimed at enhancing code readability through automation. Their study reveals that the approach for discerning readability-enhancing commits achieves an accuracy of approximately 86%. However, the model's precision in suggesting actions to ameliorate code readability is comparatively modest, fluctuating between 21% and 28%. Our findings lay the groundwork for the development of such predictive models.

More generic LLM models could leverage our insights to enhance their predictive capabilities. These models could refine their feature extraction processes by considering the specific factors that developers prioritize for code understandability. Incorporating our findings into their frameworks could result in more accurate assessments of code readability, aligning more closely with real-world developer concerns. Thus, our work offers a valuable resource for improving the effectiveness of broader LLM-based approaches to code evaluation. CodeBERT, introduced in (161) by Zhangyin Feng et al., is designed for tasks like code summarization and code-to-text generation. Analyzing inline reviews for readability can offer valuable insights into the patterns and priorities developers have regarding code readability. By integrating these insights, CodeBERT's performance can be enhanced, especially in generating human-readable code summaries and comments. Understanding the readability aspects developers value most can enable models like CodeBERT to align more closely with human preferences in code generation and summarization.

5.6.3 Understandability and Code Reviews

In this section, we present studies that examine code review comments and how developers express confusion in these comments while reading code. These manifestations hint at code that may be hard to understand, either for the reviewers themselves or for other readers, or code whose understandability can be improved.

The work by Dantas et al. (119) is arguably the one that is the most related to our study.

They evaluated code readability improvements in the context of pull requests (PRs) and compared them with the rules of SonarQube. Dantas and colleagues utilized keywords such as "readability" and "understandability" and analyzed the context of proposed code changes to identify PRs that enhance code readability. The researchers identified and categorized 284 PRs and found 26 different types of code readability improvements. Among the 284 code readability improvements classified, they found that 26 of them are detected by SonarQube. Differently, our study examined 2,401 code review comments that suggested fine-grained source code improvements. We manually identified 1,012 code review comments that suggested code understandability improvements based on the reviewers' intent. We classified the understandability smells and their solutions, as well as investigated the prevalence of the patches applied to improve code understandability. We have also analyzed whether understandability improvements are accepted or not and whether they are reversed. Finally, we analyzed the potential of four different linters to detect the understandability smells.

The work of Pascarella et al. (162) aims to understand the information necessary for reviewers to conduct a proper code review. It suggests ways in which research and tool support can enhance the effectiveness and efficiency of developers in their role as reviewers. The study analyzed 900 code review comments from three large open-source projects and identified seven primary information needs of reviewers, such as understanding the uses of methods and variables declared or modified within the code. The paper concludes with recommendations on how future code review tools can better support collaborative efforts and improve the reviewing process.

Ebert et al. (147) investigates the causes and consequences of confusion in code reviews and how developers typically handle such issues. The study involved a survey of 54 valid responses and a manual evaluation of 307 code review comments (both inline and general). The authors identified 30 leading sources of confusion, 14 different ways that confusion impacts code review, and 13 coping strategies to deal with confusion in code reviews. The research revealed that developers often experience confusion during code reviews when dealing with long or complex code changes, followed by changes that address multiple issues.

Our work differs from previous studies because we aim to evaluate how developers improve understandability through code review comments. Furthermore, the findings of our study can be used as recommendations to developers to improve their source code. This can lead to code that causes less confusion when being reviewed.

Other studies have explored approaches to automate code review, in the generation of

patches (163, 164) and of code review comments (165). The former approach leverages LLMs and the latter employs information retrieval. Neither of them attempts to distinguish the purposes of the code review comments. Likewise, the study by Fregnan et al. (166) evaluated what extent a machine learning-based technique can automatically classify review changes. The researchers classified 1,504 review changes manually and then evaluated three different machine learning algorithms to see if they could classify review changes automatically at two different levels of detail. Their findings indicate that machine learning can be effectively used to classify review-induced changes. This work is complementary to these approaches in that it narrows down the scope of investigation to code understandability improvements, looking at the comments and how they affect the system under review. It shows relevance of code review comments aiming to improve understandability, highlights the high level problems that these comments address, shows how many of the corresponding improvements materialize, and indicates that they are often accepted and rarely reversed. It also provides evidence that automating the application of improvements involves a variety of different issues, targets different languages, natural and artificial, and in some cases can be solved very easily whereas in others requires a deeper understanding of how systems work.

5.7 CONCLUSION

Code understandability is crucial for efficient software development and maintenance, as it reduces the time and cost required for these processes. Despite its importance, improving code understandability is not straightforward, as the factors influencing it are not fully understood. Previous studies comparing different coding approaches have yielded inconclusive or controversial results. Additionally, style guides often do not align with experimental findings, leading to differing perceptions among developers.

To address this research gap, this study focused on code review comments in open-source projects, where developers actively strive to improve code understandability. The analysis of 2,401 code review comments identified 1,012 comments related to code understandability improvements. These findings highlight the significance of code review in enhancing code understandability. Moreover, a deeper analysis of 300 code review comments identified eight categories of code understandability smells, such as INCOMPLETE OR INADEQUATED CODE DOCUMENTATION and BAD IDENTIFIERS. Developers often accepted the suggested improvements and rarely reverted them. The study also explored the types of patches applied to ad-

dress code understandability smells, where 96.4% of applied patches were integrated into the project's codebase. Additionally, while some of the understandability issues can be readily detected and fixed by linters, others demand more intricate analysis, considering natural language components and project context.

This study contributes to the understanding of developers' concerns and actions regarding code understandability during code reviews. The dataset of code review comments serves as a valuable resource for researchers and developers, enabling the creation of automated tools to detect and repair code understandability smells. Such tools would allow developers to focus on other critical aspects of code review, promoting code correctness and identifying security vulnerabilities.

6 CONCLUSION

Code understandability is dependent on the ability to read code, and the ease of this process can vary depending on code *legibility* (how readily program elements can be identified) and *readability* (how easily a program can be understood). These aspects are influenced by factors such as code constructs, whitespace formatting, and identifier naming conventions. In this thesis, we investigate what code alternatives of writing code that improve its legibility and readability based on empirical studies and practical evidence through four studies.

6.1 CONDUCTED STUDIES

In the first study, we conducted a systematic literature review to investigate how code legibility and readability are evaluated in human-centric studies that compare different ways of writing equivalent code. We reviewed 54 primary studies and identified three categories of tasks (e.g., providing information about the code) used in these studies, along with five categories of response variables (e.g., correctness) that were used to evaluate code understandability. Additionally, we adapted a learning taxonomy to program comprehension, mapping the tasks identified in the literature review to the cognitive skills that comprise the taxonomy.

In our second study, we conducted another literature review to identify the formatting elements that have been studied and the levels of formatting elements found to positively impact code legibility when compared to functionally equivalent ones in human-centric studies. We analyzed 15 studies that compared alternatives of formatting elements, and we identified 13 factors of formatting elements (e.g., indentation) categorized into five groups (e.g., spacing). Similarly, we performed a third systematic literature review to evaluate which code elements were investigated and which were considered more readable in human-centered studies. From a list of 39 studies, we categorized 20 factors of code elements (e.g., conditional construct and styles) that were aggregated into five groups (e.g., control flow).

Finally, in the fourth study, we analyzed 2,401 code review comments from Java open-source projects on GitHub to investigate how developers improve code understandability during software development. We found that over 42% of the comments focused on improving code understandability. We identified eight categories of code understandability concerns, such as complex or long logic, from a subset of 300 comments related to code understandability. We

found that 84.3% of the suggestions to improve code understandability were accepted and integrated into the codebase. We also evaluated the ability of four well-known linters to flag the identified code understandability issues and found that they cover less than 30% of these issues.

6.2 IMPLICATIONS

The conducted studies highlighted limitations in the literature. Many comparisons were evaluated by only one primary study, such as the Blank space around operators and parameters. Additionally, some primary studies are outdated, e.g., the comparisons in the group of No Modularization versus Types of Modularization. Hence, their results might not be applicable in the current context of software development. These limitations indicate that there is a need for more replications and new studies with different configurations for the same comparison. Furthermore, many primary studies that compared code alternatives in terms of code legibility and readability did not perform a power analysis. This reveals that many of these studies are still not concerned with the minimum size that a sample must contain for the results to be more statistically reliable. As a result, the findings, even inconclusive, can be different from studies with other subjects, and their results cannot be generalized. Although we have limitations in the literature, we found some results that indicate code alternatives that improve code understandability. However, some of our findings contradict existing guidelines, emphasizing the need to update them. In this direction, we could use the findings in practice in some cases where the studies show no significant difference.

When comparing actual coding practices to existing literature on improving code understandability, we found that developers tend to adopt the code alternatives that studies suggest are the best for improving code understandability. This highlights that despite limitations in the literature, there is practical evidence that specific findings contribute to the adoption of better code alternatives. Our investigation also revealed limitations in the use of linters. While some scenarios can be addressed by implementing a new rule, such as flagging missing parentheses, other issues require a semantic evaluation and cannot be flagged by linters, such as issues related to natural language. Additionally, we discovered contradictions in the adoption of code alternatives in practice. Although studies and guidelines may suggest a specific code alternative, practitioners may choose another based on their experience and the context of the project, team, and programming language.

6.3 CONTRIBUTIONS

This thesis provides a catalog of code alternatives that can improve code understandability based on evidence from the literature and practice. Such knowledge can aid developers in creating guidelines to write code based on evidence and automated aids, such as linters and recommendation systems, to improve code legibility and readability. We also adapted an existing learning taxonomy to the context of program comprehension, making it easier for researchers to design new studies and enhance our understanding of existing ones.

Additionally, we created a dataset of code review comments that suggest code understandability improvements, along with the source code associated with these comments and patches for code understandability improvements. This dataset was made publicly available and can prove to be a valuable resource for researchers and developers, enabling the creation of automated tools to detect and repair code understandability issues. Furthermore, researchers can use this dataset to investigate other characteristics of code review comments and their context. We also present an analysis of four well-known linters for the Java language and their ability to apply identified understandability improvement issues. Developers can use this analysis to identify gaps in these tools and propose new rules when possible.

The data of the four studies are available at
<https://github.com/delanohelio/understanding-code-understandability>

Finally, our studies are reported in four novel papers. Two papers have already been published, one is under review, and another one is to be submitted.

6.4 AVENUES FOR FUTURE WORKS

In the following paragraphs, we suggest future works in both short and long-term contexts.

Short-term. We found that many comparisons were evaluated based on only one study. Replications or new studies with different configurations of these studies could enhance the reliability of their findings. Additionally, other studies could investigate the main learning activities used in readability studies. In our study with code reviews, we discovered 1,012 code review comments that suggest code understandability improvements in our dataset. However, for some applications, such as machine learning training, this number of data could be small. Therefore,

a new work that expands the dataset could enable it to be utilized as training data. Moreover, we only explored a subset of 300 code review comments related to code understandability improvements. A more extensive exploration of these data could provide further insight into the practice of code understandability improvement. Also, we manually identified whether the linters could flag the understandability smells in a small set of data. In future work, we could expand this investigation to a more extensive set of data by executing linters.

Long-term. In our study with code review comments, we found contradictions in suggestions for code understandability improvements. A future study could investigate how different organizations consider the findings discovered in our study. This thesis focused on structural and semantic elements of the code. However, other studies on code comprehension have investigated aspects related to metrics and tools for understandability. Therefore, future studies can explore how these metrics and tools are built and propose improvements through our results. Furthermore, a machine learning-based tool could be developed to recommend code understandability improvements by using an expanded version of our dataset as training data. Finally, recent advancements in Large Language Models (LLMs) have introduced a new approach to software development. In this approach, developers use LLMs to automatically generate code. It would be interesting to conduct a study on the legibility and readability of the generated code based on our findings.

REFERENCES

1 BELCHIOR. *Alucinação*. Rio de Janeiro: Universal Music Ltda, 1976. Available in: <https://open.spotify.com/track/1IkJzT9nFbQS98tJxKQUlU?autoplay=true>. Accessed: 2023-08-05.

2 PULL Request #10182 - Azure/azure-sdk-for-java. Accessed: 2023–08–05. Available in: <https://github.com/Azure/azure-sdk-for-java/pull/10182/#discussion_r409002709>.

3 PULL Request #571 - Twilio/twilio-java. Accessed: 2023–08–05. Available in: <https://github.com/twilio/twilio-java/pull/571#discussion_r489083957>.

4 PULL Request #10940 - Azure/azure-sdk-for-java. Accessed: 2023–08–25. Available in: <https://github.com/Azure/azure-sdk-for-java/pull/10940#discussion_r423093729>.

5 PULL Request #198 - Opensearch-project/security. Accessed: 2023–08–05. Available in: <https://github.com/opensearch-project/security/pull/198#discussion_r385993973>.

6 PULL Request #1066 - linkedin/cruise-control. Accessed: 2023–08–25. Available in: <https://github.com/linkedin/cruise-control/pull/1066#discussion_r373266026>.

7 PULL Request #12004 - OpenLiberty/open-liberty. Accessed: 2023–08–05. Available in: <https://github.com/OpenLiberty/open-liberty/pull/12004#discussion_r424729743>.

8 PULL Request #3921 - Kitodo/kitodo-production. Accessed: 2023–08–05. Available in: <https://github.com/kitodo/kitodo-production/pull/3921#discussion_r471343349>.

9 FULLER, U.; JOHNSON, C. G.; AHONIEMI, T.; CUKIERMAN, D.; HERNáN-LOSADA, I.; JACKOVA, J.; LAHTINEN, E.; LEWIS, T. L.; THOMPSON, D. M.; RIEDESEL, C.; THOMPSON, E. Developing a Computer Science-specific Learning Taxonomy. *ACM SIGCSE Bulletin*, Association for Computing Machinery, New York, NY, USA, v. 39, n. 4, p. 152–170, dez. 2007. ISSN 0097-8418. Disponível em: <https://doi.org/10.1145/1345375.1345438>.

10 ARNAOUDOVA, V.; PENTA, M. D.; ANTONIOL, G. Linguistic Antipatterns: What They Are and How Developers Perceive Them. *Empirical Software Engineering*, Kluwer Academic Publishers, Hingham, MA, USA, v. 21, n. 1, p. 104–158, fev. 2016. ISSN 1382-3256. Disponível em: <http://dx.doi.org/10.1007/s10664-014-9350-8>.

11 SIEGMUND, J. Program comprehension: Past, present, and future. In: *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. [S.l.: s.n.], 2016. v. 5, p. 13–20.

12 XIA, X.; BAO, L.; LO, D.; XING, Z.; HASSAN, A. E.; LI, S. Measuring program comprehension: A large-scale field study with professionals. *IEEE Transactions on Software Engineering*, v. 44, n. 10, p. 951–976, 2018.

13 MINELLI, R.; MOCCI, A.; LANZA, M. I know what you did last summer - an investigation of how developers spend their time. In: *2015 IEEE 23rd International Conference on Program Comprehension*. [S.l.: s.n.], 2015. p. 25–35.

14 SCHRÖTER, I.; KRÜGER, J.; SIEGMUND, J.; LEICH, T. Comprehending studies on program comprehension. In: IEEE. *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*. [S.l.], 2017. p. 308–311.

15 HOU, X.; ZHAO, Y.; LIU, Y.; YANG, Z.; WANG, K.; LI, L.; LUO, X.; LO, D.; GRUNDY, J.; WANG, H. *Large Language Models for Software Engineering: A Systematic Literature Review*. 2023.

16 SANTOS, R. M. a. dos; GEROSA, M. A. Impacts of Coding Practices on Readability. In: *Proceedings of the 26th Conference on Program Comprehension (ICPC '18)*. New York, NY, USA: ACM, 2018. p. 277–285. ISBN 978-1-4503-5714-2. Disponível em: <http://doi.acm.org/10.1145/3196321.3196342>.

17 FAKHOURY, S.; ROY, D.; MA, Y.; ARNAOUDOVA, V.; ADESOPE, O. Measuring the impact of lexical and structural inconsistencies on developers' cognitive load during bug localization. *Empirical Software Engineering*, Springer US, p. 1–39, ago. 2019. ISSN 1573-7616. Disponível em: <https://doi.org/10.1007/s10664-019-09751-4>.

18 MEDEIROS, F.; LIMA, G.; AMARAL, G.; APEL, S.; KäSTNER, C.; RIBEIRO, M.; GHEYI, R. An investigation of misunderstanding code patterns in C open-source software projects. *Empirical Software Engineering*, Kluwer Academic Publishers, Norwell, MA, USA, v. 24, n. 4, p. 1693–1726, ago. 2019. ISSN 1382-3256. Disponível em: <https://doi.org/10.1007/s10664-018-9666-x>.

19 WIESE, E. S.; RAFFERTY, A. N.; FOX, A. Linking Code Readability, Structure, and Comprehension among Novices: It's Complicated. In: *Proceedings of the 41st International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET '19)*. Piscataway, NJ, USA: IEEE Press, 2019. p. 84–94. Disponível em: <https://doi.org/10.1109/ICSE-SEET.2019.00017>.

20 LANGHOUT, C.; ANICHE, M. Atoms of Confusion in Java. In: IEEE. *Proceedings of the 29th IEEE/ACM International Conference on Program Comprehension (ICPC '21)*. [S.l.], 2021. p. 25–35.

21 MIARA, R. J.; MUSSELMAN, J. A.; NAVARRO, J. A.; SHNEIDERMAN, B. Program Indentation and Comprehensibility. *Communications of the ACM*, ACM, New York, NY, USA, v. 26, n. 11, p. 861–867, nov. 1983. ISSN 0001-0782. Disponível em: <http://doi.acm.org/10.1145/182.358437>.

22 FURMAN, S.; BOEHM-DAVIS, D. A.; HOLT, R. W. A Look at Programmers Communicating through Program Indentation. *Journal of the Washington Academy of Sciences*, Washington Academy of Sciences, v. 88, n. 2, p. 73–88, 2002. ISSN 00430439.

23 BAUER, J.; SIEGMUND, J.; PEITEK, N.; HOFMEISTER, J. C.; APEL, S. Indentation: Simply a Matter of Style or Support for Program Comprehension? In: *Proceedings of the 27th International Conference on Program Comprehension (ICPC '19)*. Piscataway, NJ, USA: IEEE Press, 2019. p. 154–164. Disponível em: <https://doi.org/10.1109/ICPC.2019.00033>.

24 DOLADO, J. J.; HARMAN, M.; OTERO, M. C.; HU, L. An Empirical Investigation of the Influence of a Type of Side Effects on Program Comprehension. *IEEE Transactions on Software Engineering*, IEEE Press, Piscataway, NJ, USA, v. 29, n. 7, p. 665–670, jul. 2003. ISSN 0098-5589. Disponível em: <https://doi.org/10.1109/TSE.2003.1214329>.

25 GOPSTEIN, D.; IANNACONE, J.; YAN, Y.; DELONG, L.; ZHUANG, Y.; YEH, M. K.-C.; CAPPOS, J. Understanding Misunderstandings in Source Code. In: *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE '17)*.

New York, NY, USA: ACM, 2017. p. 129–139. ISBN 978-1-4503-5105-8. Disponível em: <http://doi.acm.org/10.1145/3106237.3106264>.

26 SMIT, M.; GERGEL, B.; HOOVER, H. J.; STROULIA, E. Code Convention Adherence in Evolving Software. In: *Proceedings of the 27th IEEE International Conference on Software Maintenance (ICSM '11)*. USA: IEEE Computer Society, 2011. p. 504–507. ISBN 9781457706639.

27 BACCHELLI, A.; BIRD, C. Expectations, outcomes, and challenges of modern code review. In: *2013 35th International Conference on Software Engineering (ICSE)*. [S.l.: s.n.], 2013. p. 712–721.

28 SADOWSKI, C.; SÖDERBERG, E.; CHURCH, L.; SIPKO, M.; BACCHELLI, A. Modern code review: A case study at google. In: *International Conference on Software Engineering, Software Engineering in Practice track (ICSE SEIP)*. [S.l.: s.n.], 2018.

29 CUNHA, A.; CONTE, T.; GADELHA, B. Code review is just reviewing code? a qualitative study with practitioners in industry. In: *Proceedings of the XXXV Brazilian Symposium on Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2021. (SBES '21), p. 269–274. ISBN 9781450390613. Disponível em: <https://doi.org/10.1145/3474624.3477063>.

30 BUSE, R. P. L.; WEIMER, W. R. Learning a Metric for Code Readability. *IEEE Transactions on Software Engineering*, IEEE Press, Piscataway, NJ, USA, v. 36, n. 4, p. 546–558, jul. 2010. ISSN 0098-5589. Disponível em: <https://doi.org/10.1109/TSE.2009.70>.

31 ALMEIDA, J. R. de; CAMARGO, J. B.; BASSETO, B. A.; PAZ, S. M. Best practices in code inspection for safety-critical software. *IEEE software*, IEEE, v. 20, n. 3, p. 56–63, 2003.

32 LIN, J.-C.; WU, K.-C. Evaluation of Software Understandability Based On Fuzzy Matrix. In: *Proceedings of the 2008 IEEE International Conference on Fuzzy Systems (IEEE World Congress on Computational Intelligence)*. [S.l.: s.n.], 2008. p. 887–892.

33 GOUGH, P. B.; TUNMER, W. E. Decoding, Reading, and Reading Disability. *Remedial and special education*, SAGE Publications Sage CA: Los Angeles, CA, v. 7, n. 1, p. 6–10, 1986.

34 HOOVER, W. A.; GOUGH, P. B. The simple view of reading. *Reading and writing*, Springer, v. 2, n. 2, p. 127–160, 1990.

35 DUBAY, W. H. The principles of readability. *Online Submission*, ERIC, 2004.

36 TEKFI, C. Readability Formulas: An Overview. *Journal of documentation*, MCB UP Ltd, v. 43, p. 261–273, 1987.

37 DAKA, E.; CAMPOS, J.; FRASER, G.; DORN, J.; WEIMER, W. Modeling Readability to Improve Unit Tests. In: *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE '15)*. New York, NY, USA: Association for Computing Machinery, 2015. p. 107–118. ISBN 9781450336758. Disponível em: <https://doi.org/10.1145/2786805.2786838>.

38 STRIZVER, I. *Type Rules: The designer's guide to professional typography*. [S.l.]: John Wiley & Sons, 2013.

39 ZUFFI, S.; BRAMBILLA, C.; BERETTA, G.; SCALA, P. Human computer interaction: Legibility and contrast. In: IEEE. *Proceedings of the 14th International Conference on Image Analysis and Processing (ICIAP '07)*. [S.l.], 2007. p. 241–246.

40 SAMPAIO, I. B.; BARBOSA, L. Software readability practices and the importance of their teaching. In: IEEE. *Proceedings of the 7th International Conference on Information and Communication Systems (ICICS '16)*. [S.l.], 2016. p. 304–309.

41 HOFMEISTER, J. C.; SIEGMUND, J.; HOLT, D. V. Shorter Identifier Names Take Longer to Comprehend. *Empirical Software Engineering*, Kluwer Academic Publishers, Norwell, MA, USA, v. 24, n. 1, p. 417–443, fev. 2019. ISSN 1382-3256. Disponível em: <https://doi.org/10.1007/s10664-018-9621-x>.

42 AJAMI, S.; WOODBRIDGE, Y.; FEITELSON, D. G. Syntax, predicates, idioms – what really affects code complexity? *Empirical Software Engineering*, Kluwer Academic Publishers, Norwell, MA, USA, v. 24, n. 1, p. 287–328, fev. 2019. ISSN 1382-3256. Disponível em: <https://doi.org/10.1007/s10664-018-9628-3>.

43 ARAB, M. Enhancing Program Comprehension: Formatting and Documenting. *ACM SIGPLAN Notices*, ACM, New York, NY, USA, v. 27, n. 2, p. 37–46, fev. 1992. ISSN 0362-1340. Disponível em: <http://doi.acm.org/10.1145/130973.130975>.

44 WANG, X.; POLLOCK, L.; VIJAY-SHANKER, K. Automatic Segmentation of Method Code into Meaningful Blocks: Design and Evaluation. *Journal of Software: Evolution and Process*, v. 26, n. 1, p. 27–49, 2014. Disponível em: <https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.1581>.

45 TEASLEY, B. E. The effects of naming style and expertise on program comprehension. *International Journal of Human-Computer Studies*, Academic Press, Inc., Duluth, MN, USA, v. 40, n. 5, p. 757–770, maio 1994. ISSN 1071-5819. Disponível em: <http://dx.doi.org/10.1006/ijhc.1994.1036>.

46 BLINMAN, S.; COCKBURN, A. Program Comprehension: Investigating the Effects of Naming Style and Documentation. In: *Proceedings of the 6th Australasian User Interface Conference (AUIC '05)*. Newcastle, Australia: ACS, 2005. p. 73–78.

47 LAWRIE, D.; MORRELL, C.; FEILD, H.; BINKLEY, D. Effective identifier names for comprehension and memory. *Innovations in Systems and Software Engineering*, v. 3, n. 4, p. 303–318, dez. 2007. ISSN 1614-5054. Disponível em: <https://doi.org/10.1007/s11334-007-0031-2>.

48 BINKLEY, D.; LAWRIE, D.; MAEX, S.; MORRELL, C. Identifier length and limited programmer memory. *Science of Computer Programming*, Elsevier North-Holland, Inc., Amsterdam, The Netherlands, The Netherlands, v. 74, n. 7, p. 430–445, maio 2009. ISSN 0167-6423. Disponível em: <http://dx.doi.org/10.1016/j.scico.2009.02.006>.

49 Ceccato, M.; Di Penta, M.; Nagra, J.; Falcarin, P.; Ricca, F.; Torchiano, M.; Tonella, P. The Effectiveness of Source Code Obfuscation: an Experimental Assessment. In: *Proceedings of the IEEE 17th International Conference on Program Comprehension (ICPC '09)*. [S.l.]: IEEE, 2009. p. 178–187. ISSN 1092-8138.

50 SCANNIELLO, G.; RISI, M. Dealing with Faults in Source Code: Abbreviated vs. Full-Word Identifier Names. In: *Proceedings of the 2013 IEEE International Conference on Software Maintenance (ICSM '13)*. USA: IEEE Computer Society, 2013. p. 190–199. ISBN 9780769549811. Disponível em: <https://doi.org/10.1109/ICSM.2013.30>.

51 AVIDAN, E.; FEITELSON, D. G. Effects of Variable Names on Comprehension: An Empirical Study. In: *Proceedings of the 25th International Conference on Program Comprehension (ICPC '17)*. Piscataway, NJ, USA: IEEE Press, 2017. p. 55–65. ISBN 978-1-5386-0535-6. Disponível em: <https://doi.org/10.1109/ICPC.2017.27>.

52 BENIAMINI, G.; GINGICHASHVILI, S.; ORBACH, A. K.; FEITELSON, D. G. Meaningful Identifier Names: The Case of Single-Letter Variables. In: *Proceedings of the 25th International Conference on Program Comprehension (ICPC '17)*. Piscataway, NJ, USA: IEEE Press, 2017. p. 45–54. ISBN 978-1-5386-0535-6. Disponível em: <https://doi.org/10.1109/ICPC.2017.18>.

53 SCHANKIN, A.; BERGER, A.; HOLT, D. V.; HOFMEISTER, J. C.; RIEDEL, T.; BEIGL, M. Descriptive Compound Identifier Names Improve Source Code Comprehension. In: *Proceedings of the 26th International Conference on Program Comprehension (ICPC '18)*. New York, NY, USA: ACM, 2018. p. 31–40. ISBN 978-1-4503-5714-2. Disponível em: <http://doi.acm.org/10.1145/3196321.3196332>.

54 BENANDER, A. C.; BENANDER, B. A.; PU, H. Recursion vs. Iteration: An Empirical Study of Comprehension. *Journal of Systems and Software*, v. 32, n. 1, p. 73–82, 1996. ISSN 0164-1212. Disponível em: <http://www.sciencedirect.com/science/article/pii/0164121295000437>.

55 SIEGMUND, J.; SCHUMANN, J. Confounding parameters on program comprehension: a literature survey. *Empirical Software Engineering*, Springer, v. 20, p. 1159–1192, 2015.

56 OLIVEIRA, D.; BRUNO, R.; MADEIRAL, F.; CASTOR, F. Evaluating Code Readability and Legibility: An Examination of Human-centric Studies. In: IEEE. *Proceedings of the 36th IEEE International Conference on Software Maintenance and Evolution (ICSME '20)*. [S.l.], 2020. p. 348–359.

57 OLIVEIRA, D.; SANTOS, R.; MADEIRAL, F.; MASUHARA, H.; CASTOR, F. A systematic literature review on the impact of formatting elements on code legibility. *Journal of Systems and Software*, v. 203, p. 111728, 2023. ISSN 0164-1212. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0164121223001231>.

58 KITCHENHAM, B. A.; BUDGEN, D.; BRERETON, P. *Evidence-Based Software Engineering and Systematic Reviews*. Chapman & Hall/CRC, 2015. p. ISBN 1482228653, 9781482228656.

59 PARSIFAL. [S.l.], 2020. Disponível em: <https://parsif.al/>.

60 ACM Digital Library. [S.l.], 2020. Disponível em: <http://dl.acm.org/>.

61 IEEE Explore. [S.l.], 2020. Disponível em: <http://ieeexplore.ieee.org/>.

62 SCOPUS. [S.l.], 2020. Disponível em: <http://www.scopus.com/>.

63 COHEN, J. A Coefficient of Agreement for Nominal Scales. *Educational and Psychological Measurement*, v. 20, n. 1, p. 37–46, 1960. Disponível em: <https://doi.org/10.1177/001316446002000104>.

64 KEELE, S. et al. *Guidelines for performing systematic literature reviews in software engineering*. [S.l.], 2007.

65 BUSE, R. P.; WEIMER, W. R. A Metric for Software Readability. In: *Proceedings of the 2008 International Symposium on Software Testing and Analysis (ISSTA '08)*. New York, NY, USA: ACM, 2008. p. 121–130. ISBN 978-1-60558-050-0. Disponível em: <http://doi.acm.org/10.1145/1390630.1390647>.

66 BINKLEY, D.; DAVIS, M.; LAWRIE, D.; MALETIC, J. I.; MORRELL, C.; SHARIF, B. The impact of identifier style on effort and comprehension. *Empirical Software Engineering*, Kluwer Academic Publishers, Hingham, MA, USA, v. 18, n. 2, p. 219–276, abr. 2013. ISSN 1382-3256. Disponível em: <http://dx.doi.org/10.1007/s10664-012-9201-4>.

67 BOYSEN, J. P.; KELLER, R. F. Measuring Computer Program Comprehension. In: *Proceedings of the 11th Technical Symposium on Computer Science Education (SIGCSE '80)*. New York, NY, USA: ACM, 1980. p. 92–102. ISBN 0-89791-013-3. Disponível em: <http://doi.acm.org/10.1145/800140.804619>.

68 CHAUDHARY, B. D.; SAHASRABUDDHE, H. V. Meaningfulness as a Factor of Program Complexity. In: *Proceedings of the ACM 1980 Annual Conference (ACM '80)*. New York, NY, USA: ACM, 1980. p. 457–466. ISBN 0-89791-028-1. Disponível em: <http://doi.acm.org/10.1145/800176.810001>.

69 JBARA, A.; FEITELSON, D. G. On the Effect of Code Regularity on Comprehension. In: *Proceedings of the 22nd International Conference on Program Comprehension (ICPC '14)*. New York, NY, USA: ACM, 2014. p. 189–200. ISBN 978-1-4503-2879-1. Disponível em: <http://doi.acm.org/10.1145/2597008.2597140>.

70 JBARA, A.; FEITELSON, D. G. How programmers read regular code: a controlled experiment using eye tracking. *Empirical Software Engineering*, Kluwer Academic Publishers, Hingham, MA, USA, v. 22, n. 3, p. 1440–1477, jun. 2017. ISSN 1382-3256. Disponível em: <https://doi.org/10.1007/s10664-016-9477-x>.

71 KASTO, N.; WHALLEY, J. Measuring the difficulty of code comprehension tasks using software metrics. In: *Proceedings of the 15th Australasian Computing Education Conference - Volume 136 (ACE '13)*. Darlinghurst, Australia, Australia: Australian Computer Society, Inc., 2013. p. 59–65. ISBN 978-1-921770-21-0. Disponível em: <http://dl.acm.org/citation.cfm?id=2667199.2667206>.

72 LOVE, T. An Experimental Investigation of the Effect of Program Structure on Program Understanding. *ACM SIGOPS Operating Systems Review – Proceedings of an ACM conference on Language design for reliable software*, ACM, New York, NY, USA, v. 11, n. 2, p. 105–113, mar. 1977. ISSN 0163-5980. Disponível em: <http://doi.acm.org/10.1145/390018.808317>.

73 WIEDENBECK, S. Beacons in computer program comprehension. *International Journal of Man-Machine Studies*, Academic Press Ltd., London, UK, UK, v. 25, n. 6, p. 697–709, dez. 1986. ISSN 0020-7373. Disponível em: <http://dx.doi.org/10.1016/S0020-7373(86)80083-9>.

74 WIEDENBECK, S. The initial stage of program comprehension. *International Journal of Man-Machine Studies*, Academic Press Ltd., London, UK, UK, v. 35, n. 4, p. 517–540, nov. 1991. ISSN 0020-7373. Disponível em: <http://dx.doi.org/10.1016/S0020-7373(05)80090-2>.

75 WOODFIELD, S. N.; DUNSMORE, H. E.; SHEN, V. Y. The Effect of Modularization and Comments on Program Comprehension. In: *Proceedings of the 5th International Conference on Software Engineering (ICSE '81)*. Piscataway, NJ, USA: IEEE Press, 1981. p. 215–223. ISBN 0-89791-146-6. Disponível em: <http://dl.acm.org/citation.cfm?id=800078.802534>.

76 O'NEAL, M. B.; EDWARDS, W. R. Complexity Measures for Rule-Based Programs. *IEEE Transactions on Knowledge and Data Engineering*, IEEE Educational Activities Department, Piscataway, NJ, USA, v. 6, n. 5, p. 669–680, out. 1994. ISSN 1041-4347. Disponível em: <https://doi.org/10.1109/69.317699>.

77 Geffen, Y.; Maoz, S. On Method Ordering. In: *Proceedings of the 24th International Conference on Program Comprehension (ICPC '16)*. [S.l.]: IEEE, 2016. p. 1–10.

78 ISELIN, E. R. Conditional statements, looping constructs, and program comprehension: an experimental study. *International Journal of Man-Machine Studies*, Academic Press Ltd., London, UK, UK, v. 28, n. 1, p. 45–66, jan. 1988. ISSN 0020-7373. Disponível em: <http://dx.doi.org/10.1016/S0020-7373(88)80052-X>.

79 MAćKOWIAK, M.; NAWROCKI, J.; OCHODEK, M. On Some End-User Programming Constructs and Their Understandability. *Journal of Systems and Software*, v. 142, p. 206–222, 2018. ISSN 0164-1212. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0164121218300633>.

80 SCHULZE, S.; LIEBIG, J.; SIEGMUND, J.; APEL, S. Does the Discipline of Preprocessor Annotations Matter? A Controlled Experiment. In: *Proceedings of the 12th International Conference on Generative Programming: Concepts & Experiences (GPCE '13)*. New York, NY, USA: ACM, 2013. p. 65–74. ISBN 978-1-4503-2373-4. Disponível em: <http://doi.acm.org/10.1145/2517208.2517215>.

81 SIEGMUND, J.; PEITEK, N.; PARNIN, C.; APEL, S.; HOFMEISTER, J.; KäSTNER, C.; BEGEL, A.; BETHMANN, A.; BRECHMANN, A. Measuring Neural Efficiency of Program Comprehension. In: *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE '17)*. New York, NY, USA: ACM, 2017. p. 140–150. ISBN 978-1-4503-5105-8. Disponível em: <http://doi.acm.org/10.1145/3106237.3106268>.

82 SYKES, F.; TILLMAN, R. T.; SHNEIDERMAN, B. The Effect of Scope Delimiters on Program Comprehension. *Software: Practice and Experience*, v. 13, n. 9, p. 817–824, 1983. Disponível em: <https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.4380130908>.

83 TROCKMAN, A.; CATES, K.; MOZINA, M.; NGUYEN, T.; KäSTNER, C.; VASILESCU, B. "Automatically Assessing Code Understandability" Reanalyzed: Combined Metrics Matter. In: *Proceedings of the 15th International Conference on Mining Software Repositories (MSR '18)*. New York, NY, USA: ACM, 2018. p. 314–318. ISBN 978-1-4503-5716-6. Disponível em: <http://doi.acm.org/10.1145/3196398.3196441>.

84 WIESE, E. S.; RAFFERTY, A. N.; KOPTA, D. M.; ANDERSON, J. M. Replicating Novices' Struggles with Coding Style. In: *Proceedings of the 27th International Conference*

on Program Comprehension (ICPC '19). Piscataway, NJ, USA: IEEE Press, 2019. p. 13–18. Disponível em: <https://doi.org/10.1109/ICPC.2019.00015>.

85 YEH, M. K.-C.; GOPSTEIN, D.; YAN, Y.; ZHUANG, Y. Detecting and Comparing Brain Activity in Short Program Comprehension Using EEG. In: *Proceedings of the 2017 IEEE Frontiers in Education Conference (FIE '17)*. Los Alamitos, CA, USA: IEEE Computer Society, 2017. p. 1–5. Disponível em: <https://doi.ieeecomputersociety.org/10.1109/FIE.2017.8190486>.

86 SCALABRINO, S.; BAVOTA, G.; VENDOME, C.; LINARES-VáSQUEZ, M.; POSHYVANYK, D.; OLIVETO, R. Automatically Assessing Code Understandability. *IEEE Transactions on Software Engineering*, p. 1–1, 2019. ISSN 2326-3881.

87 Tenny, T. Program Readability: Procedures Versus Comments. *IEEE Transactions on Software Engineering*, v. 14, n. 9, p. 1271–1279, set. 1988. ISSN 2326-3881.

88 OMAN, P. W.; COOK, C. R. A Paradigm for Programming Style Research. *ACM SIGPLAN Notices*, ACM, New York, NY, USA, v. 23, n. 12, p. 69–78, dez. 1988. ISSN 0362-1340. Disponível em: <http://doi.acm.org/10.1145/57669.57675>.

89 OMAN, P. W.; COOK, C. R. Typographic Style is More than Cosmetic. *Communications of the ACM*, ACM, New York, NY, USA, v. 33, n. 5, p. 506–520, maio 1990. ISSN 0001-0782. Disponível em: <http://doi.acm.org/10.1145/78607.78611>.

90 Kleinschmager, S.; Robbes, R.; Stefik, A.; Hanenberg, S.; Tanter, E. Do Static Type Systems Improve the Maintainability of Software Systems? An Empirical Study. In: *Proceedings of the 20th IEEE International Conference on Program Comprehension (ICPC '12)*. [S.l.: s.n.], 2012. p. 153–162.

91 MALAQUIAS, R.; RIBEIRO, M.; BONIFáCIO, R.; MONTEIRO, E.; MEDEIROS, F.; GARCIA, A.; GHEYI, R. The Discipline of Preprocessor-Based Annotations Does #ifdef TAG n't #endif Matter. In: *Proceedings of the 25th International Conference on Program Comprehension (ICPC '17)*. Piscataway, NJ, USA: IEEE Press, 2017. p. 297–307. ISBN 978-1-5386-0535-6. Disponível em: <https://doi.org/10.1109/ICPC.2017.41>.

92 STEFIK, A.; SIEBERT, S. An Empirical Investigation into Programming Language Syntax. *ACM Transactions on Computing Education (TOCE)*, ACM, New York, NY, USA, v. 13, n. 4, p. 19:1–19:40, nov. 2013. ISSN 1946-6226. Disponível em: <http://doi.acm.org/10.1145/2534973>.

93 POSNETT, D.; HINDLE, A.; DEVANBU, P. A Simpler Model of Software Readability. In: *Proceedings of the 8th Working Conference on Mining Software Repositories (MSR '11)*. New York, NY, USA: Association for Computing Machinery, 2011. p. 73–82. ISBN 9781450305747. Disponível em: <https://doi.org/10.1145/1985441.1985454>.

94 SCALABRINO, S.; LINARES-VáSQUEZ, M.; OLIVETO, R.; POSHYVANYK, D. A Comprehensive Model for Code Readability. *Journal of Software: Evolution and Process*, v. 30, n. 6, p. e1958, 2018. Disponível em: <https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.1958>.

95 STEFIK, A.; GELLENBECK, E. Empirical studies on programming language stimuli. *Software Quality Journal*, Kluwer Academic Publishers, Hingham, MA,

USA, v. 19, n. 1, p. 65–99, mar. 2011. ISSN 0963-9314. Disponível em: <http://dx.doi.org/10.1007/s11219-010-9106-7>.

96 BLOOM, B.; ENGELHART, M.; FURST, E.; HILL, W. H.; KRATHWOHL, D. R. *Taxonomy of educational objectives: The classification of educational goals. Handbook I: Cognitive domain*. New York: David McKay Company, 1956.

97 BIGGS, J. *Teaching for quality learning at university*. Buckingham: Open University Press, 1999.

98 ANDERSON, L.; BLOOM, B.; KRATHWOHL, D.; AIRASIAN, P.; CRUIKSHANK, K.; MAYER, R.; PINTRICH, P.; RATHS, J.; WITTROCK, M. *A Taxonomy for Learning, Teaching, and Assessing: A Revision of Bloom's Taxonomy of Educational Objectives*. Longman, 2001. ISBN 9780801319037. Disponível em: <https://books.google.com.br/books?id=EMQIAQAAIAAJ>.

99 ROSSBACH, C. J.; HOFMANN, O. S.; WITCHEL, E. Is transactional programming actually easier? In: *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '10)*. New York, NY, USA: Association for Computing Machinery, 2010. p. 47–56. ISBN 9781605588773. Disponível em: <https://doi.org/10.1145/1693453.1693462>.

100 CASTOR, F.; OLIVEIRA, J. a. P.; SANTOS, A. L. Software transactional memory vs. locking in a functional language: A controlled experiment. In: *Proceedings of the Compilation of the Co-Located Workshops on DSM'11, TMC'11, AGERE! 2011, AOOPES'11, NEAT'11, & VMIL'11*. [S.l.]: Association for Computing Machinery, 2011. p. 117–122.

101 WOOD, J. R.; WOOD, L. E. Card Sorting: Current Practices and Beyond. *Journal of Usability Studies*, Usability Professionals' Association, Bloomingdale, IL, v. 4, n. 1, p. 1–6, nov 2008.

102 SHARIF, B.; MALETIC, J. I. An Eye Tracking Study on camelCase and under_score Identifier Styles. In: *Proceedings of the 2010 IEEE 18th International Conference on Program Comprehension (ICPC '10)*. Washington, DC, USA: IEEE Computer Society, 2010. p. 196–205. ISBN 978-0-7695-4113-6. Disponível em: <https://doi.org/10.1109/ICPC.2010.41>.

103 JOHNSON, M.; BEEKMAN, G. *Oh! Thinks Lightspeed Pascal!* [S.l.]: WW Norton, 1988.

104 RITCHIE, D. M.; KERNIGHAN, B. W.; LESK, M. E. *The C Programming Language*. [S.l.]: Prentice Hall Englewood Cliffs, 1988.

105 BROOKS, R. Using a Behavioral Theory of Program Comprehension in Software Engineering. In: *Proceedings of the 3rd International Conference on Software Engineering (ICSE '78)*. [S.l.]: IEEE Press, 1978. p. 196–201.

106 CHANCE, B.; ZHUANG, Z.; UNAH, C.; ALTER, C.; LIPTON, L. Cognition-activated low-frequency modulation of light absorption in human brain. *Proceedings of the National Academy of Sciences*, National Acad Sciences, v. 90, n. 8, p. 3770–3774, 1993.

107 PETERSON, J. L. On the Formatting of Pascal Programs. *ACM SIGPLAN Notices*, Association for Computing Machinery, New York, NY, USA, v. 12, n. 12, p. 83–86, dec 1977. ISSN 0362-1340.

108 CRIDER, J. E. Structured Formatting of Pascal Programs. *ACM SIGPLAN Notices*, Association for Computing Machinery, New York, NY, USA, v. 13, n. 11, p. 15–22, nov 1978. ISSN 0362-1340.

109 GUSTAFSON, G. G. Some Practical Experiences Formatting Pascal Programs. *ACM SIGPLAN Notices*, Association for Computing Machinery, New York, NY, USA, v. 14, n. 9, p. 42–49, sep 1979. ISSN 0362-1340.

110 ELLIS, P. D. *The Essential Guide to Effect Sizes: Statistical Power, Meta-Analysis, and the Interpretation of Research Results*. [S.l.]: Cambridge University Press, 2010.

111 COHEN, J. Statistical Power Analysis. *Current Directions in Psychological Science*, Sage Publications Sage CA: Los Angeles, CA, v. 1, n. 3, p. 98–101, 1992.

112 GOPSTEIN, D.; ZHOU, H. H.; FRANKL, P.; CAPPOS, J. Prevalence of Confusing Code in Software Projects: Atoms of Confusion in the Wild. In: *Proceedings of the 15th International Conference on Mining Software Repositories (MSR '18)*. New York, NY, USA: ACM, 2018. p. 281–291. ISBN 978-1-4503-5716-6. Disponível em: <http://doi.acm.org/10.1145/3196398.3196432>.

113 KERNIGHAN, B. W.; PLAUGER, P. J. *Elements of Programming Style*. McGraw-Hill, 1978. p.

114 JANSEN, A. R.; BLACKWELL, A. F.; MARRIOTT, K. A tool for tracking visual attention: The restricted focus viewer. *Behavior research methods, instruments, & computers*, Springer, v. 35, n. 1, p. 57–69, 2003.

115 SCALABRINO, S.; BAVOTA, G.; VENDOME, C.; LINARES-VÁSQUEZ, M.; POSHYVANYK, D.; OLIVETO, R. Automatically assessing code understandability: How far are we? In: IEEE. *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. [S.l.], 2017. p. 417–427.

116 PIANTADOSI, V.; FIERRO, F.; SCALABRINO, S.; SEREBRENIK, A.; OLIVETO, R. How does code readability change during software evolution? *Empirical Software Engineering*, Springer, v. 25, p. 5374–5412, 2020.

117 FAKHOURY, S.; ROY, D.; HASSAN, S. A.; ARNAOUDOVA, V. Improving Source Code Readability: Theory and Practice. In: *Proceedings of the 27th International Conference on Program Comprehension (ICPC '19)*. Piscataway, NJ, USA: IEEE Press, 2019. p. 2–12. Disponível em: <https://doi.org/10.1109/ICPC.2019.00014>.

118 ROY, D.; FAKHOURY, S.; LEE, J.; ARNAOUDOVA, V. A model to detect readability improvements in incremental changes. In: *28th ICPC*. [S.l.: s.n.], 2020. p. 25–36. ISBN 9781450379588.

119 Dantas, C.; Rocha, A.; Maia, M. How do developers improve code readability? an empirical study of pull requests. In: *2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. [S.l.: s.n.], 2023.

120 CHONG, C. Y.; THONGTANUNAM, P.; TANTITHAMTHAVORN, C. Assessing the students' understanding and their mistakes in code review checklists: An experience report of 1,791 code review checklist questions from 394 students. In: *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)*. [S.l.: s.n.], 2021. p. 20–29.

121 GAWANDE, A. *The Checklist Manifesto: How to Get Things Right*. [S.l.]: Profile books, 2011.

122 ISRAEL, G. D. et al. Determining sample size. University of Florida Cooperative Extension Service, 1992.

123 PAWLAK, R.; MONPERRUS, M.; PETITPREZ, N.; NOGUERA, C.; SEINTURIER, L. Spoon: A Library for Implementing Analyses and Transformations of Java Source Code. *Software: Practice and Experience*, Wiley-Blackwell, v. 46, p. 1155–1179, 2015. Disponível em: <https://hal.archives-ouvertes.fr/hal-01078532/document>.

124 BROWN, C.; PARNIN, C. Understanding the impact of github suggested changes on recommendations between developers. In: *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. [S.l.: s.n.], 2020. p. 1065–1076.

125 TORRES, A.; OLIVEIRA, C.; OKIMOTO, M. V.; MARCILIO, D.; QUEIROGA, P.; CASTOR, F.; BONIFÁCIO, R.; CANEDO, E. D.; RIBEIRO, M.; MONTEIRO, E. An investigation of confusing code patterns in javascript. *J. Syst. Softw.*, v. 203, p. 111731, 2023.

126 NUCCI, D. D.; PALOMBA, F.; TAMBURRI, D. A.; SEREBRENIK, A.; LUCIA, A. D. Detecting code smells using machine learning techniques: Are we there yet? In: *25th International Conference on Software Analysis, Evolution and Reengineering, SANER 2018, Campobasso, Italy, March 20-23, 2018*. [S.l.]: IEEE Computer Society, 2018. p. 612–621.

127 KALLIAMVAKOU, E.; GOUSIOS, G.; BLINCOE, K.; SINGER, L.; GERMAN, D. M.; DAMIAN, D. The promises and perils of mining github. In: *Proceedings of the 11th working conference on mining software repositories*. [S.l.: s.n.], 2014. p. 92–101.

128 SHIMAGAKI, J.; KAMEI, Y.; MCINTOSH, S.; PURSEHOUSE, D.; UBAYASHI, N. Why are commits being reverted?: a comparative study of industrial and open source projects. In: IEEE. *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. [S.l.], 2016. p. 301–311.

129 SADOWSKI, C.; GOGH, J. van; JASPAN, C.; SÖDERBERG, E.; WINTER, C. Tricorder: Building a program analysis ecosystem. In: *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*. [S.l.]: IEEE Computer Society, 2015. p. 598–608.

130 VASSALLO, C.; PANICHELLA, S.; PALOMBA, F.; PROKSCH, S.; GALL, H. C.; ZAIDMAN, A. How developers engage with static analysis tools in different contexts. *Empirical Software Engineering*, Springer, v. 25, p. 1419–1457, 2020.

131 LENARDUZZI, V.; PECORELLI, F.; SAARIMAKI, N.; LUJAN, S.; PALOMBA, F. A critical comparison on six static analysis tools: Detection, agreement, and precision. *Journal of Systems and Software*, Elsevier, v. 198, p. 111575, 2023.

132 SPOTBUGS - Bug Descriptions. Accessed: 2023–08–05. <https://spotbugs.readthedocs.io/en/stable/bugDescriptions.html>.

133 PMD - Java Rules. Accessed: 2023–08–05. <https://docs.pmd-code.org/latest/pmd_rules_java.html>.

134 SONARQUBE - Java static code analysis. Accessed: 2023–08–05. <https://rules.sonarsource.com/java/>.

135 CHECKSTYLE - Checks. Accessed: 2023–08–05. <https://checkstyle.sourceforge.io/checks.html>.

136 STEFANOVIĆ, D.; NIKOLIĆ, D.; DAKIĆ, D.; SPASOJEVIĆ, I.; RISTIĆ, S. Static code analysis tools: A systematic literature review. *Annals of DAAAM & Proceedings*, v. 7, n. 1, 2020.

137 TÓMASDÓTTIR, K. F.; ANICHE, M. F.; DEURSEN, A. van. The adoption of javascript linters in practice: A case study on eslint. *IEEE Trans. Software Eng.*, v. 46, n. 8, p. 863–891, 2020.

138 DABIC, O.; AGHAJANI, E.; BAVOTA, G. Sampling projects in github for MSR studies. In: *18th IEEE/ACM International Conference on Mining Software Repositories, MSR 2021*. [S.l.]: IEEE, 2021. p. 560–564.

139 GROVES, R. M.; JR., F. J. F.; COUPER, M. P.; LEPKOWSKI, J. M.; SINGER, E.; TOURANGEAU, R. *Survey Methodology*. 2nd. ed. [S.l.]: Wiley, 2009.

140 FOWLER, M. *Refactoring: improving the design of existing code*. [S.l.]: Addison-Wesley Professional, 2018.

141 PULL Request #1488 - Opemedx/edx-app-android. Accessed: 2023–08–05. Available in: <https://github.com/openedx/edx-app-android/pull/1488#discussion_r537391027>.

142 PULL Request #792 - Apache/jena. Accessed: 2023–08–05. Available in: <https://github.com/apache/jena/pull/792#discussion_r486429425>.

143 PULL Request #2772 - CorfuDB/CorfuDB. Accessed: 2023–08–05. Available in: <https://github.com/CorfuDB/CorfuDB/pull/2772#discussion_r497201087>.

144 PULL Request #2172 - Mapstruct/mapstruct. Accessed: 2023–08–05. Available in: <https://github.com/mapstruct/mapstruct/pull/2172#discussion_r463982271>.

145 PULL Request #3337 - EssentialsX/Essentials. Accessed: 2023–08–05. Available in: <https://github.com/EssentialsX/Essentials/pull/3337#discussion_r441271747>.

146 PULL Request #12579 - Alluxio/alluxio. Accessed: 2023–08–05. Available in: <https://github.com/Alluxio/alluxio/pull/12579#discussion_r530723377>.

147 EBERT, F.; CASTOR, F.; NOVIELLI, N.; SEREBRENIK, A. An exploratory study on confusion in code reviews. *Empirical Software Engineering*, Springer, v. 26, n. 12, January 2021. Disponível em: <https://doi.org/10.1007/s10664-020-09909-5>.

148 PULL Request #256 - TeamNewPipe/NewPipeExtractor. Accessed: 2023–08–05. Available in: <https://github.com/TeamNewPipe/NewPipeExtractor/pull/256#discussion_r378437693>.

149 PULL Request #7241 - Triplea-game/triplea. Accessed: 2023–08–05. Available in: <https://github.com/triplea-game/triplea/pull/7241#discussion_r460364607>.

150 TUFANO, R.; PASCARELLA, L.; TUFANO, M.; POSHYVANYK, D.; BAVOTA, G. Towards automating code review activities. In: *Proceedings of the 43rd International Conference on Software Engineering*. [S.l.]: IEEE Press, 2021. (ICSE '21), p. 163–174.

151 LIN, H. Y.; THONGTANUNAM, P. Towards automated code reviews: Does learning code structure help? In: *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. [S.l.: s.n.], 2023. p. 703–707.

152 EBERT, F.; CASTOR, F.; NOVIELLI, N.; SEREBRENIK, A. Confusion detection in code reviews. In: *2017 IEEE International Conference on Software Maintenance and Evolution, ICSME 2017, Shanghai, China, September 17-22, 2017*. [S.l.]: IEEE Computer Society, 2017. p. 549–553.

153 COSTA, J. A. S. da; GHEYI, R.; CASTOR, F.; OLIVEIRA, P. R. F. de; RIBEIRO, M.; FONSECA, B. Seeing confusion through a new lens: on the impact of atoms of confusion on novices' code comprehension. *Empir. Softw. Eng.*, v. 28, n. 4, p. 81, 2023.

154 LUCAS, W.; BONIFáCIO, R.; CANEDO, E. D.; MARCÍLIO, D.; LIMA, F. Does the introduction of lambda expressions improve the comprehension of java programs? In: *Proceedings of the XXXIII Brazilian Symposium on Software Engineering*. [S.l.]: Association for Computing Machinery, 2019. (SBES '19), p. 187–196.

155 HALSTEAD, M. H. *Elements of Software Science (Operating and Programming Systems Series)*. USA: Elsevier Science Inc., 1977. ISBN 0444002057.

156 LAVAZZA, L.; MORASCA, S.; GATTO, M. An empirical study on software understandability and its dependence on code characteristics. *Empirical Software Engineering*, Springer, v. 28, n. 6, p. 155, 2023.

157 MI, Q.; KEUNG, J.; XIAO, Y.; MENSAH, S.; MEI, X. An inception architecture-based model for improving code readability classification. In: *Proceedings of the 22nd International Conference on Evaluation and Assessment in Software Engineering 2018*. New York, NY, USA: Association for Computing Machinery, 2018. (EASE '18), p. 139–144. ISBN 9781450364034. Disponível em: <https://doi.org/10.1145/3210459.3210473>.

158 MI, Q.; KEUNG, J.; XIAO, Y.; MENSAH, S.; GAO, Y. Improving code readability classification using convolutional neural networks. *Information and Software Technology*, v. 104, p. 60–71, 2018. ISSN 0950-5849. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0950584918301496>.

159 MI, Q.; ZHAN, Y.; WENG, H.; BAO, Q.; CUI, L.; MA, W. A graph-based code representation method to improve code readability classification. *Empirical Softw. Engg.*, Kluwer Academic Publishers, USA, v. 28, n. 4, may 2023. ISSN 1382-3256. Disponível em: <https://doi.org/10.1007/s10664-023-10319-6>.

160 VITALE, A.; PIANTADOSI, V.; SCALABRINO, S.; OLIVETO, R. Using deep learning to automatically improve code readability. In: IEEE. *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. [S.l.], 2023. p. 573–584.

161 FENG, Z.; GUO, D.; TANG, D.; DUAN, N.; FENG, X.; GONG, M.; SHOU, L.; QIN, B.; LIU, T.; JIANG, D.; ZHOU, M. CodeBERT: A pre-trained model for programming and natural languages. In: *Findings of the Association for Computational Linguistics: EMNLP*

2020. Online: Association for Computational Linguistics, 2020. p. 1536–1547. Disponível em: <https://aclanthology.org/2020.findings-emnlp.139>.

162 PASCARELLA, L.; SPADINI, D.; PALOMBA, F.; BRUNTINK, M.; BACCHELLI, A. Information needs in contemporary code review. *Proceedings of the ACM on Human-Computer Interaction*, ACM New York, NY, USA, v. 2, n. CSCW, p. 1–27, 2018.

163 TUFANO, R.; PASCARELLA, L.; TUFANO, M.; POSHYVANYK, D.; BAVOTA, G. Towards automating code review activities. In: IEEE. *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. [S.l.], 2021. p. 163–174.

164 TUFANO, R.; MASIERO, S.; MASTROPAOLO, A.; PASCARELLA, L.; POSHYVANYK, D.; BAVOTA, G. Using pre-trained models to boost code review automation. In: *Proceedings of the 44th International Conference on Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2022. (ICSE '22), p. 2291–2302. ISBN 9781450392211. Disponível em: <https://doi.org/10.1145/3510003.3510621>.

165 HONG, Y.; TANTITHAMTHAVORN, C.; THONGTANUNAM, P.; ALETI, A. Commentfinder: A simpler, faster, more accurate code review comments recommendation. In: *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2022. (ESEC/FSE 2022), p. 507–519. ISBN 9781450394130. Disponível em: <https://doi.org/10.1145/3540250.3549119>.

166 FREGNAN, E.; PETRULIO, F.; GERONIMO, L. D.; BACCHELLI, A. What happens in my code reviews? an investigation on automatically classifying review changes. *Empirical Software Engineering*, Springer, v. 27, n. 4, p. 89, 2022.