Investigating the Effect of Code Comments on Source Code Comprehension - A Reproduction Study

Bachelor of Science Thesis in Software Engineering and Management

Christofer Jidarv
Robin Hansen

Department of Computer Science and Engineering
UNIVERSITY OF GOTHENBURG
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2023

Investigating the Effect of code comments on Source Code Comprehension – A Reproduction Study

Supervisor: Francisco Gomes de Oliveira Neto
Examiner: Richard Berntsson Svensson

University of Gothenburg
Chalmers University of Technology
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
UNIVERSITY OF GOTHENBURG
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2023

Investigating the Effect of Code Comments on Source Code Comprehension - A Reproduction Study

Christofer Jidarv
Department of Computer Science and Engineering
University of Gothenburg
Gothenburg, Sweden
gusjidach@student.gu.se

Robin Hansen
Department of Computer Science and Engineering
University of Gothenburg
Gothenburg, Sweden
gushanrod@student.gu.se

***Abstract*—Software comments are written to get an understanding of what code does, its function, and its purpose. There is a consensus in the industry that code comments improve software comprehension, but is this really the case? In this research, we investigate the effect of code comments on software comprehension by conducting an experiment where participants are to study one of three types of code snippets containing the same code. The only difference is the code comments, one contains well-written code comments, one contains poorly-written ones and the last contain no comments at all. Our analysis is done on the data from 51 students studying a programming-related program in Sweden. We split comprehension into two parts, perceived comprehension, which is estimated by the participants after reading the code snippet. And actual comprehension which is tested. Our results show that participants receiving the well-written code comments perceive to comprehend the code snippet better, but in actuality, all three comment types score very similarly on the actual comprehension, with the well-written comments scoring the lowest.**

I. Introduction

Software is read and developed daily, and an important aspect of software development is the comprehension of the code that makes up software. The lack of code readability can have an immense effect on both productivity and comprehension and adds an additional risk of introducing bugs and flaws into code [15]. In order to reduce the number of bugs and flaws, as well as increase overall productivity and comprehension, readability has to remain at a high and acceptable level. By assuring that the readability is at an acceptable and satisfactory level, code comments can assist the developer in understanding the current state of the code base, and reduce the risk of confusion when being read by other developers.

Even if code comments are used in industry to improve comprehension, many researchers today argue that there is no consensus on the best way of using code comments for this purpose. According to Rani et al. [14], there is no agreement on what quality means related to code comments. Despite the lack of formal agreements on what defines quality regarding code comments, Rani et al. [14] propose using quality attributes to associate code comments with quality aspects. Quality attributes such as comprehensibility, cohesion, consistency, and usefulness, can be chosen. Adhering to quality attributes when writing comments, allows developers to focus on one or more aspects or quality attributes when writing code comments. Researchers have not been able to find an agreement on what qualities a comment should have in order to define what a good comment is [14]. Comments vary in their respective use cases, purpose, scope, and usage, and no comment structure fits all needs. A definition of what a good comment is in relation to comprehensibility is needed.

The use of source code comments is vast. Everything from bug fixing to code reviews relies upon code comments. Buse et al. [2] argue that one of the most time-consuming and important code-related tasks, code maintenance, becomes more effective and less time-consuming when well-written source code comments are applied. Maintenance of code involves understanding the existing code and making changes according to changing demands. In the interest of maintainability, developers first must understand the code that was written, in order to make changes that involve improving the software product. Buse et al. [2] state that at least 70% of the software development life cycle is in the form of maintenance, which means that due to the linkage between maintenance of code and readability, assurances that the code remains readable after changes are needed for future changes to be performed without issues.

Alternatively to source code comments, a trend of 'self-documenting code' is starting to be more and more common. Self-documenting code is an approach to creating comprehensible code, where the aim includes choosing the most optimal identifier names and assuring that the code is able to be read without the need for extensive documentation (code comments). Code that is written with the aim of being self-documenting, is meant to be readable to every individual, although the comprehension of code is subjective, it is, therefore, difficult to assure that every individual is able to fully understand the written code. This point is strengthened by Posnett et al. [13], who stated that studies of readability are difficult to perform due to the subjective answers given by the participants. On the grounds that code readability is subjective, different aspects of a code base, including the choice of identifier names, may not always be clear to other individuals. This means that code may benefit from comments as a way to increase comprehension.

This research is a reproduction study of Börstler and Paech [1], by this we mean that we have a different method to the baseline experiment, but is still based on the same theory structure [7]. Börstler and Paech conducted a study [1] consisting of method chaining combined with the effect of code comments on the readability of source code. In order to measure the perceived readability, metrics were collected in the form of Likert scales between 1 and 5. The initial questions that included the question 'I am an experienced Java programmer´, and the question relating to perceived readability of 'Based on your programming experience, how would you rate the readability of the previous piece of code´ both used Likert scales between 1 representing 'Strongly disagree´/'Very difficult´ and 5 representing 'Strongly agree´/'Very easy´. In addition to the perceived readability, actual readability was measured in the form of cloze tests, where sections of code were removed and the participants were tested on how well they were able to follow their task of filling in the removed sections with suitable code.

This research is conducted with an experiment investigating participants' perceived and actual comprehension of one of three types of code snippets. The experiment was conducted similarly to an ABC test to determine how and if code comments affect the comprehension of source code. The participants were asked to rate their comprehension of the code snippet from 'very difficult´ to 'very easy´ (Likert scale). This perceived comprehension was later tested with the use of further questions, including a cloze test. We found that the code snippet with the most well-written code comments was perceived as a lot easier, but when tested, it scored lower than any of the other two comment types, even if it only was with a small margin. We also saw that code comments did not significantly increase actual code comprehension, the code snippet with no comments scored better than the snippet with the well-written comment. The findings of our thesis lead to the following contributions to Software Engineering research and practice: (i) that the consensus in the industry that code comments improve code comprehension, might be misleading, (ii) that well-written code comments improve the perceived comprehension of source code, but not its actual comprehension, and (iii) this research's findings, which contradict the consensus, shows the need for further research in this topic.

II. Related Work

Borstler and Paech state in their related work *"... that the role of comments for code quality should be studied in more detail"* [1]. This need for further research into code comment quality is further expressed by Steidl et al. [18]. As discussed in [1] most of the research regarding code comments is more than 20 years old, this may indicate a need for new and updated research, as also indicated by Steidl et al. [18]. In order to get a better view of the frameworks and guidelines of comments and their relation to improving readability, Fakhoury et al. [5] discovered that there exist multiple frameworks and guidelines on how to write comments to improve readability. This is without a consensus on which frameworks and guidelines improve readability the most [5] [11]. Fakhoury et al. [5] continued by stating that by analyzing state-of-the-art readability models, they fail to be suitable for daily maintenance. Scalabrino et al. [16] state that state-of-the-art code readability models do not take source code lexicon into consideration, and added that analyses of source code lexicon may improve the state-of-the-art readability models, and suggest utilizing textual features in combination with structural features to enhance code readability models.

A. Comprehension and Readability

The ability to read code does not necessarily mean that one comprehends the code. Readability and comprehension are often linked with one another but are not interchangeable [13]. Posnett et al. [13] define readability as "...the 'accidental´ component of code understandability...", using understandability rather than comprehension. By that definition, comprehension is a key part of reaching readability.

The study of Piantadosi et al. [12] came to the conclusion that new code in big commits has a higher risk of affecting code readability. This shows that developers should preferably focus on limiting the size of the commits and properly assure that the code and its associated comments within the commit are readable, prior to introducing new code into the main branch. Assuring that new code and comments have a high level of readability and comprehensibility, will result in the combined code base having a higher level of readability and comprehensibility. Piantadosi et al. [12] also mention that code bases with a low level of readability will continue to have a lower level of readability. From this information, readability and comprehensibility can be difficult to improve once the code base has reached a level of low readability, and developers should always strive to reach a high level of readability within a code base.

Gopstein et al. [8] conducted an experimental research study that involved comparing the comprehension between obfuscated and non-obfuscated code, including the comprehension and confusion relating to both of those categorical types of software. The conclusions gathered from the study from Gopstein et al. [8] is that obfuscated code has a significant and direct impact on comprehension, resulting in substantial increases in the misunderstanding of the code snippets of obfuscated code compared to non-obfuscated code. Obfuscated code consists of code that is purposely written to be difficult to read, where it can be used to avoid malicious intent through the lack of comments and difficulty to correlate identifier names. Writing readable code and comments can be considered tremendously critical in order to improve comprehension, reduce misunderstandings, and reduce the risk of introducing bugs and flaws, along with making code more maintainable for the developers that read and maintains the code base.

Minimizing nesting can be seen through the experiment conducted by Johnson et al. [10] as increasing the perceived comprehension and reducing the amount of time spent reading and attempting to understand the written code. Johnson et al [10] continue to state that 86.5% of the participants felt that minimizing nesting is more readable than the alternative version that does not minimize nesting and that 51.6% believed that readable code includes reducing the level of nesting. From this information, nesting can potentially be seen as reducing readability and could affect comprehensibility negatively, and developers should find alternatives to nesting whenever possible in order to increase the level of readability and comprehensibility of their written code.

B. Code comments

Fakhoury et al. [4] came to the conclusion that having comments and code containing linguistic anti-patterns, which includes sub-optimal practices in relation to documentation and identifiers, results in a higher mental burden during code comprehension tasks. Code comprehension may be severely impacted when linguistic anti-patterns and structural imperfections (e.g. line length, number of spaces, etc) are combined. This suggests that code and comments play an important role in relation to comprehension, where comprehension can be reduced significantly if developers do not take linguistic anti-patterns and structural aspects into consideration.

Rani et al. [14] conducts an SLR on the field of code comments and finds that there is much more to discover. Furthermore, they state that most of the research done on the subject is done in Java, rather than any other code language, and therefore might not translate across all Software Engineering fields. There are multiple types of code comments with multiple types of use cases, Pascarella et al. [11] developed in their study a taxonomy of the types of comments. They did this by analyzing more than 40,000 lines of code comments from 14 different Java projects. They categorized them into six different types with their corresponding sub-types. In this study, we primarily focus on the type of comments Pascarella et al. call *purpose comments*. Purpose comments are code comments that *"... describe the functionality of linked source code either in a shorter way than the code itself or in a more exhaustive manner"* [11].

In industry, it is commonly used that AI create code comments for undocumented or old code, to improve comprehension [17]. But even if this is the case, Stapleton et al. [17] state that human-written code comments exceed the ability to improve code comprehension in source code, even though the study showed no difference in quality between human-written and machine-written code comments.

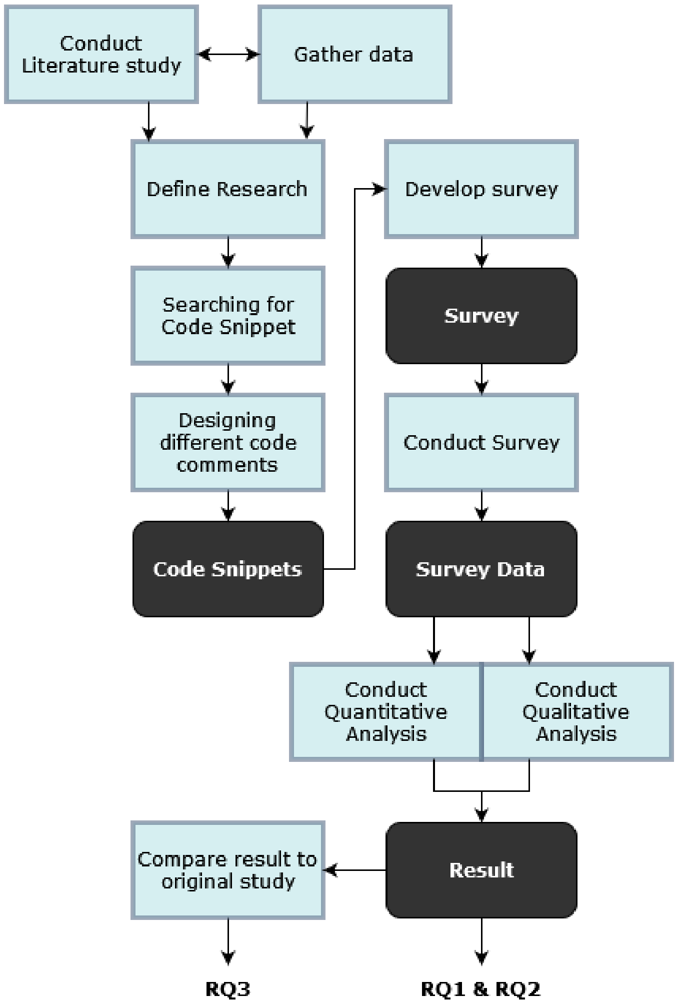

Conduct Literature study
Gather data
Define Research
Develop survey
Searching for Code Snippet
Survey
Designing different code comments
Conduct Survey
Code Snippets
Survey Data
Conduct Quantitative Analysis
Conduct Qualitative Analysis
Compare result to original study
Result
RQ3
RQ1 & RQ2

Fig. 1: Figure explaining the flow of the methodology in this research, where the arrows tell the flow. Teal rectangles are depicting actions taken and the dark rectangles are depicting the artifacts created.

III. Research Methodology

The purpose of this research is to investigate how code comments in source code affect the comprehension of said source code. We do this by extending upon the research by Börstler and Paech [1] and conducting an experiment. We aim to answer the following research questions:

- **RQ1:** What effects do source code comments have on software comprehension?

- **RQ2:** To what extent do different code comments of contrasting quality affect software comprehension?
- **RQ3:** How do our results compare to the result of Börstler and Paech [1]?

This experiment is built similarly to an A/B/C test or a split test, where the subject only gets to take part in one (of three) versions of a survey. Each version of the survey is divided into three parts 2: the *first part* includes some personal questions regarding the subject's coding experience as well as potential reading disorders. The *second part* consisted of an initial look at a code snippet, as well as questions asking the subject to estimate their comprehension (perceived comprehension) of the code snippets. Furthermore, each snippet is broken down into parts that the subjects rate individually. The *third and final part* include questions testing the subjects' actual comprehension of the code snippets. The process of developing the survey, as well as all other parts of this research, is visualized in figure 1.

Reproduction study

While conducting our literature study, we found that only studies older than 20 years conducted research on this topic (code comments' effect on source code comprehension), while newer research regarding code comments mainly focused on automated comment generation with the help of AI, and assumed that older research on the topic was still valid. This fact was true until we found Börstler and Paech [1] that themselves saw this lack of current research. We adapt and extend their work on multiple points, we detail the differences between the original study and ours below.

It is easy to view this as a replication study, but this is not the case, it is a reproduction study. A reproduction study is defined as using a different method to the baseline experiment but still based on the same theory structure [7], while a replication study is doing the same study again, meaning that you would follow the methodology as closely as possible, and by doing so repeating the research. As mentioned prior, we are not repeating the same research, we extend upon it [7].

This research study did not include method chaining as part of its scope, it would therefore not be valid to call it a replication study. Nevertheless, the scope and RQs of this research overlapped significantly with Börstler and Paech [1], which in turn justifies extending upon it. Some researchers even see reproduction studies as a better alternative to replication studies, the reason for this is that by using the same method *"... there could be a cause-effect relationship between the method and the observation"* [7]. Gomez et al. [7] continue describing that some researchers see reproduction studies as a strategy to mitigate errors caused by methods and procedures, even though the better the methods and procedures are, the less need there is to conduct a reproduction study [7].

One key difference between Börstler and Paech [1] and this study is the omitting of method-chaining as a part of its research focus. This is further a reason that validates the choice of reproduction over a replication study.

As we are doing a reproduction study it is important to visualize what differs between this study and the core research. The list below depicts changes in our methodology compared to the research conducted by Börstler and Paech [1].

- **Differing population** - [1] had a population of only 1st and 2nd-year university students studying Computer Science from Heidelberg University, Germany. We have a population of university students studying Software Engineering, Computer Science, and other programming-related programs from universities in Sweden
- **Differing code snippets** - [1] had code snippets focusing on method chaining and including comments. We have code snippets that focus on general Java code, which includes three variants of code comments.
- **Differing cloze tests** - as we have different code snippets we also therefore have different cloze tests, as they are building upon the code in the code snippet. But worth noting here is that our cloze test build upon and is closely adapted from the cloze test used by [1].
- **Differing Research Questions(RQ)** - [1] has one RQ involving method chaining, and one RQ involving the amount and quality of comments that are not specific to any programming language. Our RQs focus on how comments affect the comprehensibility in code.

Data collection and Subjects

The collection of data was conducted using an online tool called Qualtrics. The primary reason for this choice was the need for a tool that allowed for A/B/C testing as well as supported the disabling of returning to the previous question. The survey was published and managed through Qualtrics and was conducted online seeing that the need for larger amounts of participants was imperative, along with removing the restrictions related to timing and location. The questions and code snippets used in the experiment can be found in our online repository. [1]

The subjects in this study all consist of University students of Data Science, Software Engineering, and similar programming-related programs in Sweden. By reaching out to universities, we collected contact information for relevant students of any year. The only difference in this strategy was when reaching out to students of Software Engineering and Management at Gothenburg University, here we used Discord and Slack as preferred platforms.

[1]https://github.com/RobbanGit/CodeCommentsAndComprehension

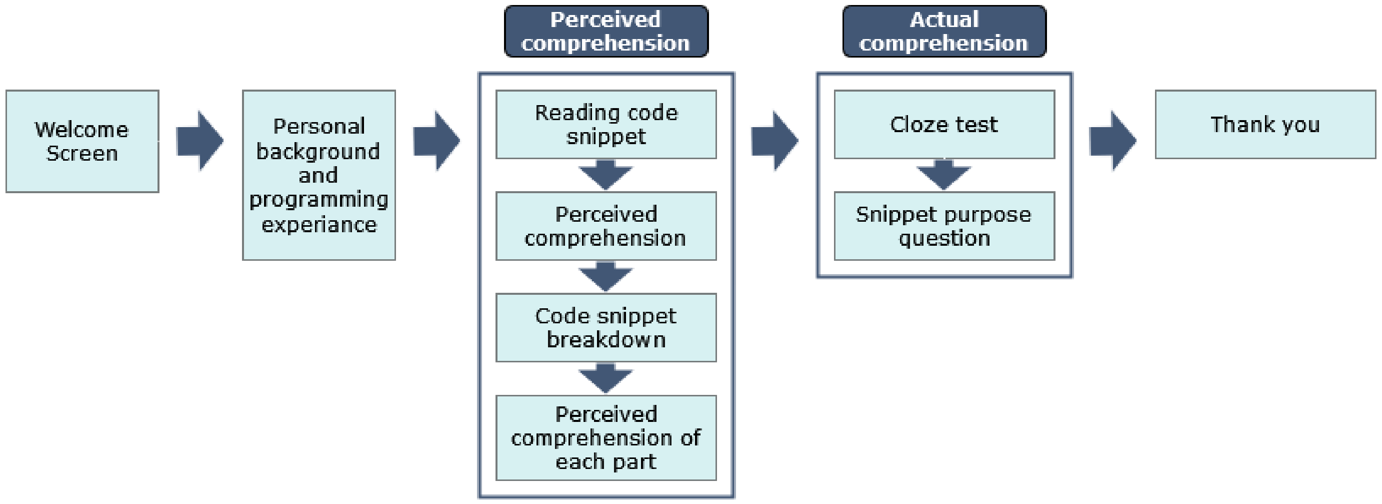


Fig. 2: Figure displaying the survey structure used in this experiment.

We found the subjects by scouring the web through universities in Sweden looking for universities that have programming-related programs. We then contacted these universities/programs by email, listed on their websites. The list below shows the universities that either sent some sort of mailing list (for us to use) or published our survey on any of their forums/platforms:

- University of Gothenburg
- Chalmers University
- University of Stockholm
- University of Luleå
- University of Mälardalen
- University of Karlstad
- University of Örebro

Instruments

Code Snippets: Unlike an ordinary survey, the subjects are only able to see and answer questions regarding one of the three different code snippets. The code snippets included identical source code, the only difference between the snippets was the code comments or lack thereof. We defined these as:

- **Good Comments** *(GC)*: A well-structured and written *purpose comment* [11] that explains the code snippet and its purpose and includes further information beyond the code it depicts (see listing 1).
- **Bad Comments** *(BC)*: A code comment that only explains what the code does and not its purpose (see listing 2).
- **No Comments** *(NC)*: The same code as the two aforementioned snippets but without any code comments

```
/*
* Returns a new fresh folder with the given
* paths under the temporary folder.
* For example, if you pass in the strings
* {@code "parent"} and {@code "child"}
* then a directory named {@code "parent"}
* will be created under the temporary folder
* and a directory named {@code "child"}
* will be created under the newly-created
* {@code "parent"} directory.
*/
```

Listing 1: Code comments used in Good Comment snippet.

```
/*
  * Creates folder in sequence based on paths.
  */
```

Listing 2: Code comments used in Bad Comment snippet.

Importantly, the code used in this experiment needed to be generic and without subject or application dependencies. We also wanted code that was well-reviewed and used, to mitigate the risk for bugs and issues. With this in mind, we scoured through open-source repositories, such as GitHub, GitLab, and Google Open Source. We chose code from the JUnit4 repository [2] since it fits our criteria. This code had more than 8000 stars and 3000 forks on GitHub, something that ensured that the code was well-reviewed and used. The code was also non-application and non-domain specific and only demanded from the reader sufficient knowledge in Java to be able to read. As our populous consists of university students, the complexity of the code needed to be taken into account. Even if the code we chose consisted of multiple for-loops and if-statements, we do not consider this code to be too complex for any university students that have taken an introductory course in Java, to understand.

[2]https://github.com/junit-team/junit4

Comprehension Test: Börstler and Paech tested the subjects' actual comprehension of the code snippets by conducting cloze tests [1]. The cloze tests used were a series of missing segments in the code snippet for the subject to fill in. We adapted these comprehension techniques on our own code snippets and developed a similar cloze test. As in Börstler and Paech [1] experiment, we do not expect the subjects to be able to fill in the blanked-out parts identically to the original code. The blanked-out part is for the subjects to show that they understand the code and its function. We use the cloze test in this experiment because it is a well-proven way to test reading comprehension, even code comprehension [3], [9].

In addition to the cloze test, we chose to add a question that answers if the subjects understand the function of the method that the code snippets depict. We found that by doing an initial test of the survey, there was a possibility to answer it correctly without fully understanding the complete function of the method. We, therefore, developed the question to fill this gap, *If the input to this piece of code would be ["Apple", "Orange", "Lemon", "Banana"], please explain what the output would be, and why.* The author Feitelson [6] strengthens the need for such a question to be implemented into experiments relating to comprehension of code. Feitelson [6] explains that interpretation is close to "real" understanding of code and that individuals can solve the expected output of a program to show that they comprehend the code, which means that a question relating to the execution of a program is necessary to assure comprehension.

Data analysis

This analysis section covers the quantitative analysis of the results gathered from the Likert questions relating to perceived comprehension, and in the cases where a snippet of code was perceived as difficult, a follow-up question was asked. This follow-up question was answered in the free-form format and allowed the subject to explain what or why it was difficult. The analysis continues with the analysis of the data from the cloze test and data from the last question that concludes the actual comprehension. Both the quantitative and qualitative data were extracted from Qualtrics into a TSV file, where only the quantitative and qualitative data for submissions that included answers to most questions and had completion of 100% were analyzed.

a) Quantitative data analysis: After filtering out metadata and non-finished submissions from the TSV file due to easier handling and formatting that was more suitable for us to extract the useful data, the file was saved in the CSV format, to be used for plotting. The reason for exporting a TSV file that we later save into a CSV file is that Qualtrics does not export data into a CSV file correctly, this method is a workaround. The plotting of the quantitative data was performed in the scripting language, R, and utilized the ggplot2 and HH libraries. Quantitative data in the TSV, and by extension, CSV file, was ranged between 1 and 5, where 1 is equal to 'Very Difficult' and 5 is equal to 'Very Easy'. This allowed for an easier generation process of diagrams, where R and its associated libraries were able to read through the CSV file, and based on value (1–5), were able to sum each corresponding value. Stacked bar plots were used for grouping together the different comment types under their respective answer category for the initial questions (see figure 3a) Divergent bar plots were used in question 4 as well as in questions 5–8, which easily display the divergences across the different comment types (see figure 4a for an example of a divergent bar chart). The generated diagrams consist of data pertaining to the initial questions (questions 1–3), perceived comprehension of the entire code snippet per each comment type (question 4), and perceived comprehension of each of the four parts of the code snippets (Marked Code 1-4) per comment type (question 5–8). The two diagrams that pertain to perceived comprehension, both equal up to 100% for their respective comment type and show the proportions and percentages for their respective level of difficulty. A vertical line at 0 percent is present to ease the comparison between the proportions. All diagrams relating to perceived comprehension were marked between very difficult and very easy, and are ordinal.

b) Actual Comprehension: Actual comprehension (AC) was collected through the cloze test (Q11) and the question pertaining to the expected output of a method (the code snippet) (Q12), where each submitted answer was rated based on levels of correctness.

These levels of correctness spanned between 0 and 2, where 0 was for incorrect answers that did not resemble an understanding of what the section of code did, 1 was for partially-correct answers, and 2 was for completely correct answers. These scores were assigned manually by the two researchers of this study and were done cooperatively throughout all the submitted answers of Q11 and Q12. Before analyzing the data, we discussed and concluded (for each question) what the criteria were for each score. These conclusions were used to maintain a fair and balanced scoring. The managing of the answers relating to actual comprehension, the score-keeping, and the calculations of the average/median/SD values for the scores, were done using Microsoft Excel.

c) Qualitative analysis: In the cases where one or more code snippets were found to have been difficult by the subjects, an additional question per code snippet was shown, asking what the subject believed to be difficult to understand about the code snippet that they previously read. This type of question allows for qualitative analysis to be performed on the answer that the subject gives, and gives an insight into what may have created hindrances in the comprehension of the code snippet. Upon reading the data, content analysis was made, in order for us to be able to create codes about the potential hindrances in comprehension. The answers were collected in the form of free text, through a text box. The qualitative data analysis was performed using Microsoft Excel for reading the data, and Microsoft Word for noting the observations and creating codes.

IV. Result

We received 179 submissions of the survey, while only 51 of these submissions were fully completed (28,5%).

Number of participants per type:

- **Good Comments *(GC)*** : 18 participants.
- **Bad Comments *(BC)*** : 17 participants.
- **No Comments *(NC)***: 16 participants.

39 the subjects have concluded 2 years or more of their studies, 16 of which have concluded 4 or more years (see Figure 3b). Of all 51 subjects, 27 have more than 1 year of Java experience, whereas 24 have less than one year of experience in Java (see figure 3a). When we look closer at the split between the three code snippet types, we see that most of the participants have concluded two years or more of their studies. 13 out of 18 of the subjects that participated in GC had concluded 2 years of studies or more. While for BC these numbers were 14 out of 17, and NC 12 out of 16 (see figure 3b). When it comes to the number of years of Java experience, we see that BC consists mostly of subjects with less than 1 year of experience (70.5%), while GC and NC have a more even split between the years of experience (see figure 3a)

A. Research Question 1 & 2

As mentioned before, the survey was broken down into three parts, initial questions, perceived comprehension (PC), and actual comprehension (AC). The PC was documented through question 4 (Code Question) (see figure 4a) and Questions 5-8 (Marked Questions 1-4) (see figure 4b). Figure 4a shows us that GC perceived the comprehension of the code snippet to be mostly easy and very easy, with 61% for both levels. While only 12% perceived the snippet to be difficult or very difficult to comprehend. When viewing BC's and NC's answers, we see that the comprehension of the snippet is perceived as more difficult. Viewing BC, we see that 24% perceive the snippet to be very difficult or difficult. 25% of NC perceive the snippet to be difficult to comprehend, while no one of NC perceived it as very difficult. Noticeable is also that 41% and 38% of BC and NC respectively, chose to answer the question with neutral, while only 28% of GC.

When viewing the breakdown of the perceived comprehension in regards to every marked code snippet (see figure 4a for the breakdown, and 4c for the snippet), we see a clear indication that "marked 3" is perceived as a lot harder to comprehend than the other three marked parts for all comment types. 41% of BC perceived "marked 3" to be difficult or very difficult while 25% of NC perceived it to be difficult or very difficult (only type to have subjects that perceive it as very difficult). Only 17% of GC perceived it to be difficult, while 39% and 22% perceived it to be easy and very easy, respectively. Marked part 4 was perceived to be the easiest to comprehend, only 12% of NC perceived it to be difficult, while GC and BC did not perceive it to be difficult at all. 94% of GC perceived "marked 4" to be easy or very easy to comprehend.

When comparing the PC to the AC we see that even though GC perceived to comprehend the code snippet better than both BC and NC, the AC shows another story. In table I we see how the median of GC is only 1 while for both BC and NC, it's 2, while the average is roughly the same across all three types.

TABLE I: Actual Comprehension

	GC	BC	NC
Median:	1	2	2
Average:	1.141	1.269	1.3
Standard Deviation:	0.852	0.888	0.869

This table shows that the standard deviation (SD) is close to the same across all comment types, where an SD of 0.8 means that the spread of data points is clustered around the average. In the case of NC, it means that there exist outliers but most data points are closer to 1.3 than the opposite. We see that GC has an average of 1 rather than 2 (as the other types have), as there are only three values (0, 1, and 2) the jump from 1 to 2 might seem large, but one has to take into account that there is no other option than these three values.

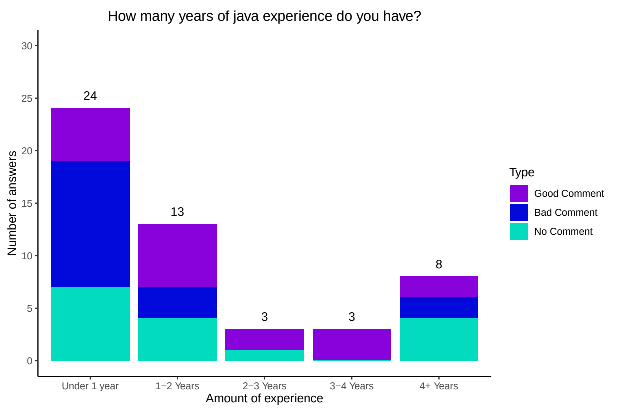
RQ1 What effects do source code comments have on software comprehension?

We note that participants that received Good Comment (GC) perceived the code to be easier to comprehend than both BC and NC, while still scoring a median AC of 1 (while Bad Comment (BC) and No Comment (NC) scored 2). GC and NC also scored similarly on both PC and AC. We can therefore not say that code comments have any positive effect on source code comprehension, The data indicates that the truth might be the opposite.

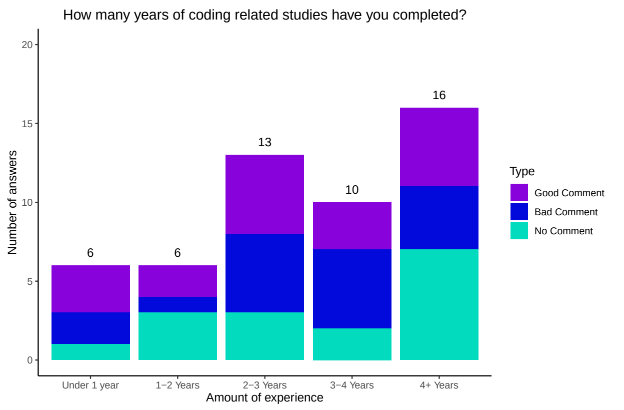
RQ2 To what extent do different code comments of contrasting quality affect software comprehension?

When comparing the scores of the two types of comments used (GC and BC) we see that while GC perceive to comprehend the code snippet better than BC (see figure 4a), when tested GC shows a lower actual comprehension than BC, even if it by a small margin (see table I). Therefore we can not conclude that the quality of the code comment affects software comprehension differently.

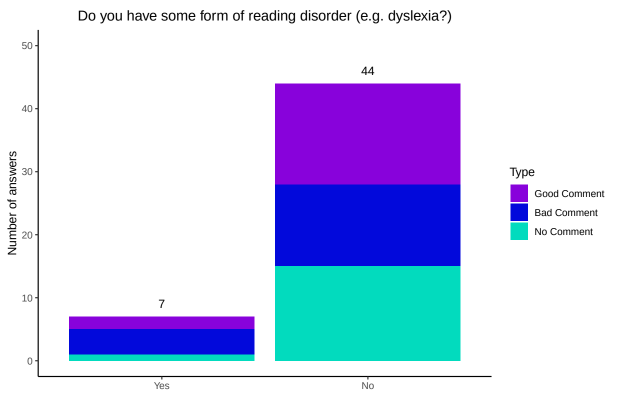
Fig. 3: Figures for each of the three initial questions (the question is displayed on the top of each figure)



(a) Figure showing the number of participants by number of years of experience with Java. Bars are split into the three comment types (Initial Question 1) (the number on top of the bar shows the total number of participants).

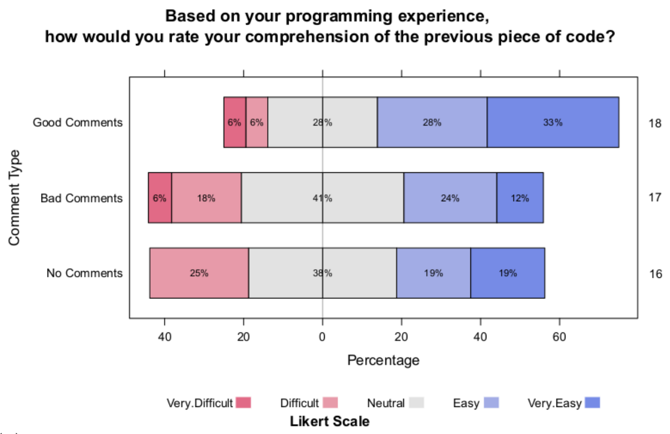


(b) Figure showing the number of participants by the number of years of concluded coding-related studies. Bars are split into the three comment types (Initial Question 2) (the number on top of the bar shows the total number of participants).

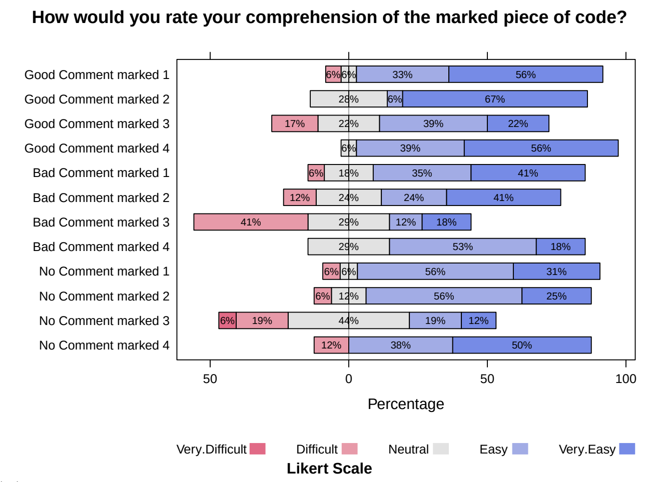


(c) Figure showing the number of participants with or without a reading disorder. Bars are split into three comment types (Initial Question 3) (the number on top of the bar shows the total number of participants).

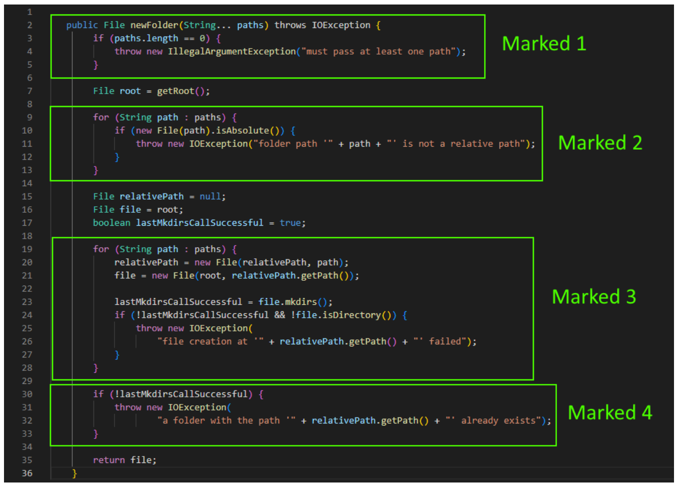
Fig. 4: Figures for perceived comprehension of the Likert questions



(a) Figure showing the perceived comprehension of the code snippet, grouped by comment type. The number on the right shows the number of participants.



(b) Figure showing the perceived comprehension of each of the marked parts of the code snippet. Grouped by comment type and marked part.


```java

public File newFolder(String... paths) throws IOException {
    if (paths.length == 0) {
        throw new IllegalArgumentException("must pass at least one path");
    }

    File root = getRoot();

    for (String path : paths) {
        if (new File(path).isAbsolute()) {
            throw new IOException("folder path '" + path + "' is not a relative path");
        }
    }

    File relativePath = null;
    File file = root;
    boolean lastMkdirsCallSuccessful = true;

    for (String path : paths) {
        relativePath = new File(relativePath, path);
        file = new File(root, relativePath.getPath());

        lastMkdirsCallSuccessful = file.mkdirs();
        if (!lastMkdirsCallSuccessful && !file.isDirectory()) {
            throw new IOException(
                "file creation at '" + relativePath.getPath() + "' failed");
        }
    }

    if (!lastMkdirsCallSuccessful) {
        throw new IOException(
                "a folder with the path '" + relativePath.getPath() + "' already exists");
    }

    return file;
}
```

(c) Image showing each marked part of the code snippet.

a) Qualitative Analysis: When we try to figure out why some participants find it hard to comprehend the code snippet, by analyzing the follow-up questions to the marked questions by doing content analysis. We did this by coding the root cause of the difficulty based on the answers, these codes were then analyzed to see patterns, and we categorized them by these patterns. The categories we use are ”Java Syntax”, ”Method Confusion” and ”File Library”.

By ’Java syntax’, we refer to the lack of knowledge or confusion regarding parts of the code that pertains to syntax found in the Java programming language. One participant stated that the participant did not know what the syntax ”String...” meant as a parameter in a method, while others stated the misunderstanding regarding Enhanced for-loops (also known as for-each loops). *”I sometimes have a bit of a hard time with for-each loops.”* stated by one participant.

```java
    for(String path : paths)
```

Listing 3: For-each loop used in snippet

```java
for (String path : paths) {
    relativePath = new File(relativePath, path);
    file = new File(root, relativePath.getPath());
    lastMkdirsCallSuccessful = file.mkdirs();

    if (!lastMkdirsCallSuccessful && !file.isDirectory()) {
        throw new IOException(
            "file creation at '"
            + relativePath.getPath()
            + "' failed");
    }
}
```

Listing 4: The third piece of marked code that perceived difficult to understand.

”Method confusion” is a category representing the ability to comprehend the functionality and purpose of the method (code snippet). Here participants did not define the reason for low comprehension of the syntax itself, but rather on the methods parts and function. One example can be *”Why must the path be relative?”* stated by one participant, while another wrote *”Second, the only thing we use relativePath for, is to prepare the next File constructor, surely there’s a better solution”*. Many participants did not understand the use of relative versus absolute path, while others state confusion regarding the use of IOExceptions and their use in the code.

’File Library’ might fall under any of the previous categories, but as so many stated issues with the file library, it got its own category. The File library consists of various aspects and functions related to the creation and managing of files. Multiple subjects mentioned that they did not properly understand or had never interacted with the File library before. An example of this, as stated by one subject, is: *”Never used java File class libraries”*. This code was prevalent in the third marked code snippet, in all comment types.

B. Research Question 3

TABLE II: The table presents the Median(Med), Average(Avg), and Standard Deviation(SD) for PC and AC

	PC			**PC [1]**			**AC [1]**		
	GC	BC	NC	GC	BC	NC	GC	BC	NC
Med	4	3	3	3	3	2	0.33	0.33	0.33
Avg	3.78	3.18	3.31	2.95	2.64	2.51	0.41	0.41	0.43
SD	1.13	1.04	1.04	1.02	1.02	1.02	0.35	0.34	0.36

In the table above (see Table II), ’PC´ refers to the perceived comprehension of our experiment, whilst PC [1] and AC [1] refers to the perceived and actual comprehension of Börstler and Paech [1]. In this section, the terminology of ’their‘, directly relates to the study of Börstler and Paech [1]. The data gathered for our experiment is further compared with the data and conclusion of the original study conducted by Börstler and Paech [1].

In Table II, the standard deviation(SD) is very similar across the comment types for our perceived comprehension. Each of the standard deviation values is close to 1, which indicates that the data points are relatively clustered around the average. For GC, this indicates that the majority of the data points are floating around the average value of 3.78, instead of the other values. In the table, similar to our standard deviation values, PC [1] have standard deviation values that also happen to float around 1, which indicates that the spread of their data points is floating around the average value of their comment types.

Similarities have been found in the perceived comprehension of source code, where GC is associated with higher levels than BC and NC (see figure II), which our results show the average values of 3.78 for GC, compared to 3.18 for BC and 3.31 for NC. Similarly, the results of Börstler and Paech [1] indicate the same, with GC having an average value of 2.95, BC having 2.64, and NC having 2.51. Our results are, however, not completely akin. One contradiction exists in the ordering of the different comment types for perceived comprehension, where our results show that NC has a slightly higher average value of 3.31 compared to BC with 3.18. Their results show, however, that BC has a higher perceived comprehension with a higher average value of 2.64 than NC with 2.51. This contradiction means that our code snippet with no comments was more comprehensible than our code snippet with bad comments.

An additional similarity found between our and Börstler and Paech [1] results, is that the values for the standard deviation are similar between both the experiments, in which both are close to the value of 1.

Our data for actual comprehension, once again, differs from the data noted in the study conducted by Börstler and Paech [1]. The first difference is the ordering of comprehensible comment types. In our results, GC is seen as the least comprehensible comment type with an average value of 1.141, followed by BC with 1.269, and ended with NC as the most comprehensible comment type with 1.3

(see table I). This ordering from least comprehensible to most comprehensible shows that in our experiment, NC is more comprehensible than GC, although by a smaller amount (see table I). The ordering for Börstler and Paech [1] are moderately different, with BC and GC having a lower average value of 0.41, and NC with a higher average value of 0.43 (see table II).

RQ3 How does our result compare to the result of Börstler and Paech [1]?

Our results show both similarities and contradictions to the findings by Börstler and Paech [1]. Similarities were found to be that GC was perceived as more comprehensible than BC and NC, and that the standard deviation between all comment types of both studies was similar. Contradictions were found to be that NC was perceived as being slightly more comprehensible than BC according to our data, as well as that our actual comprehension results show slightly larger differences between our average values for GC, BC, and NC. Börstler and Paech [1], on the other hand, have slightly more similar values between their comment types, whereas GC and BC share the same average value. Despite the different orderings from least- to most-comprehensible comment types for actual comprehension, our differences between the comment types are small enough to be seen as relatively similar, which is a similarity to Börstler and Paech [1].

V. Discussion

As the introduction entails, code comments are widely used in industry to improve readability or comprehension of the source code [14], while this statement might be true, our data shows no indication that code comments in source code improve comprehension. With a small sample size like ours, we can not infer causation. But the data still makes an interesting statement.

The lack of a clearly better performing 'type' is intriguing. The AC between all three types differs by such a small amount that one can argue that with a larger sample size, any of the three types could perform better. This goes against most if not all, previous findings on the topic, such as Rani et al. [14] that state "As developers spend significant amount of time reading code, including comments, having readable comment can help them in understanding code easier" and Stapleton el al. [17] that states "Source code comments play an invaluable role in facilitating program comprehension".

If using the comparison of perceived comprehension and actual comprehension in BC and NC as a baseline, we see a significant difference in GC. GC perceived the code to be so much easier, while their actual comprehension is worse than both BC and NC. An explanation for this might be that longer and more thorough comments strain on the participant's mental capacity and therefore do not benefit improved comprehension, something Rani et al. [14] experienced. Another explanation could be that the discrepancy between the three types in regard to the number of years of code-related studies that have been concluded (see figure 3b) would be the reason for these results. We see that GC has more participants with under 1 year of concluded studies while BC has more than twice the participants that concluded 3-4 years and NC has almost twice the number of participants who concluded 4+ years of studies.

GC having a higher level of perceived comprehension than BC and NC in both our and Börstler and Paech [1]'s study was always expected, but what was not expected is the different ordering of the comment types in the results for the actual comprehension. Our results show that both NC and BC are more comprehensible than GC, which is a surprise, which also contradicts the results from the study of Börstler and Paech [1]. It is possible that BC had comments that reduced perceived comprehension and that those comments made the subjects question the code itself, or alternatively, the splitting of subjects between BC and NC were giving NC subjects that answered more positively. Due to BC and NC having similar values, they can be comparable to Börstler and Paech's [1] closely-valued BC and NC.

Considering that their results show similar values between all comment types, it is interesting that GC was the least comprehensible comment type according to our results. There may be differences in quality between our comments in our code snippets and their comments in their code snippets, which resulted in the difference in the ordering of comprehension of the comment types.

As mentioned in the result we got 51 responses, these are the responses from participants that finished the entire survey. If we add the ones that did not finish the entire survey we end up with 179 responses. This means that 128 participants did not finish. We see this as a significant number and wonder how this comes to be. One participant (that did finish the survey) wrote that the survey took more time than anticipated, this might be the reason for a part of all unfinished submissions. We also believe that the effort it takes to participate in a survey where the participants are to read and understand a piece of code might be too high for some. Some participants may have been under the impression that the survey would not ask for any extraneous mental effort from the participants and therefore gave up as soon as they realized what the survey entailed. There could be numerous reasons as to why a participant did not finish the survey that they began. From our submission data, some participants wrote that they thought the survey was long, either in the number of questions or the amount of time required to finish the survey. Many of those non-finished surveys were close to the final question of function output. There were also participants who did not proceed past the initial questions, which could be due to the same reason of not knowing the amount of time required for completion. This can be seen as a way to improve further research, inform the participants about the estimated time or the total number of questions included in the survey, and give information about how many questions are left before completion (at each question). It is possible that the non-finished submissions would have concluded the survey if these strategies would have been in effect, and this in turn would have affected the result.

A. Threats to validity

The number of participants is limited by the participation scope that we did set. We are limited to collecting data from participants who are students of Computer Science and similar educations at universities in Sweden. Our limitation in participation size is also relevant to the platforms that we utilize to contact potential participants.

Collecting a larger amount of data than we accounted for may require additional time for the analysis, which is related to the limitation of participation size.

Our form of collecting data from our participants over the Internet may have an impact on the reliability of our data. By conducting the data collection over the internet, the participants can participate in our research study in a location of their choice, which in turn might lead to more participants, but also enhances the risk for "cheating" where the subject uses other tools that were not given to answer the questions.

By voluntarily participating in the experiment, we can conclude that the participants were interested in the topic and due to their own interests, are unlikely to "cheat" in ways that could have an impact on their results. Despite the low risk of "cheating", the back button was disabled, meaning that once an answer had been given, the participants were unable to go back to the previous question. In addition, if identifiable data was collected in the survey, restrictions regarding attempts for the same participant to perform the survey multiple times could potentially have been introduced to reduce the risk of data unreliability, although introducing questions that collect identifiable data significantly reduces the level of anonymization of the submissions, which could have negative impacts on the participants and their associated submitted results.

In future research, this may be converted to an in-person observation or interview to reduce this limitation further.

When reviewing this study, one needs to take into account that comments do not necessarily have to be the primary contributor to the differences between perceived comprehension and actual comprehension, where other contributors, such as previous experience, could be valid. This could be an interesting factor as to why there are differences between perceived comprehension and actual comprehension.

As seen in the result of perceived comprehension of the marked parts of the code snippet (see figure 4b), all parts are not perceived to be equally hard to understand. The third part of the code snippet was perceived to be a lot harder to comprehend than the other three parts, this, in turn, might lead to affect individuals' estimated comprehension of the whole snippet differently. For instance one participant might see three easy parts and one difficult part and then rate the entire snippet as easy, while another participant might also see three easy parts and one hard part and rate the entire snippet as hard. Having an uneven complex code snippet might make the code snippet harder for the participants to rate, and therefore affect the result.

Provided that we reach out to individuals that we find, we are not able to completely randomly pick participants to invite to participate in our research study. This takes the form of participation bias. Despite the invitation to the survey not being randomly picked, the survey tool, Qualtrics, was set up to strive for an even distribution between the variations of code snippets. Even if this is the case, we can not guarantee a 100% even split between the code snippets.

We believe that not every individual that we invite to the survey in our research study will want to participate. This may have been reduced when we provided additional information regarding our research study. By conducting the experiment over the Internet in the form of a questionnaire, potential hindrances to participating in our research study may also have been reduced.

Due to not forming questions relating to what university a participant studies at, we are unable to predict the number of participants at each university. Furthermore, it means that we are unable to see differences in characteristics of the participants from each university, and may therefore be unable to compare the potential results between participants from different universities.

We saw that enough individuals took part as participants in our research study, although non-response bias could not be completely eliminated, since a portion of the individuals that received an invitation likely did not want to participate in our research study. Strides to reduce this bias was taken, including giving a short and brief message about the questionnaire, along with mentioning the importance of their participation in our research study, to increase the likelihood of participation.

Provided that the vast majority of the participants are serious about participating in our research study, a large portion of those participants would choose to not skip the qualitative questions of our questionnaire. A participation percentage of 100% in the qualitative questions of the survey may be unreasonable to expect, although it may not be unreasonable to expect that a vast majority of the participants are willing to answer the qualitative questions seriously, primarily considering their participation in the other sections of the questionnaire. As some of the text boxes prompt only when criterias are met, there is a possibility that these criteria are not met to a degree that would allow for enough data.

The comments used in this research are taken from industry and adapted to fit the definition of what we, in conjunction with Börstler and Paech [1], defined as good comments and bad comments respectively. As stated prior, there is no consensus on this matter, and in what contexts a good and bad comment is subjective, and we cannot, therefore, guarantee that the subjects in this study agree with our definition. There might be the case that the code comments used in this research are constructed poorly and therefore derives the results. It should be noted that the comments used were not written by us, and have been reviewed countless times by experienced programmers, something we are not. We can therefore not argue whether the code comments are well-written or construed using good standards.

VI. Conclusion

In this research paper, we have reported the results gathered from our experiment relating to the perceived and actual comprehension of source code comments. These results were collected through Likert scales for the perceived comprehension, as well as cloze tests and an open-ended question relating to the output of the code for the actual comprehension. This was a reproduction study of the original study conducted by Börstler and Paech [1], in which our results were compared against. Our results showed that well-written comments have an effect on perceived comprehension, although well-written comments do not have the same effect on actual comprehension. We find that the effect of well-written code comments does not improve the actual comprehension more than poor-written comments do. We also see that contradicting previous research, the code snippet with no comments was just as easy to comprehend as any of the other code snippets that had code comments in them. Despite our interesting and somewhat conflicting results, the sample size of our participants is limited, and further research on this topic may be needed in order to provide conclusive results. Future work that entails a larger sample size, a different population, and/or multiple comment snippets may provide an engaging contribution to the topic of comments and their effects on comprehension. Similarly to Börstler and Paech [1], we also see the need for additional research to be made relating to this topic.

References

[1] Jürgen Börstler and Barbara Paech. The role of method chains and comments in software readability and comprehension—an experiment. *IEEE Transactions on Software Engineering*, 42(9):886–898, 2016.

[2] Raymond PL Buse and Westley R Weimer. Learning a metric for code readability. *IEEE Transactions on software engineering*, 36(4):546–558, 2009.

[3] Curtis Cook, William Bregar, and David Foote. A preliminary investigation of the use of the cloze procedure as a measure of program understanding. *Information Processing & Management*, 20(1-2):199–208, 1984.

[4] Sarah Fakhoury, Yuzhan Ma, Venera Arnaoudova, and Olusola Adesope. The effect of poor source code lexicon and readability on developers' cognitive load. In *Proceedings of the 26th Conference on Program Comprehension*, pages 286–296, 2018.

[5] Sarah Fakhoury, Devjeet Roy, Adnan Hassan, and Vernera Arnaoudova. Improving source code readability: Theory and practice. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, pages 2–12. IEEE, 2019.

[6] Dror G Feitelson. Considerations and pitfalls for reducing threats to the validity of controlled experiments on code comprehension. *Empirical Software Engineering*, 27(6):123, 2022.

[7] O Gómez, N Juristo, and S Vegas. Replication, reproduction and re-analysis: three ways for verifying experimental findings in software engineering. In *1st International Workshop on Replication in Empirical Software Engineering Research (RESER'10), May*, volume 4.

[8] Dan Gopstein, Jake Iannacone, Yu Yan, Lois DeLong, Yanyan Zhuang, Martin K-C Yeh, and Justin Cappos. Understanding misunderstandings in source code. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 129–139, 2017.

[9] William E Hall and Stuart H Zweben. The cloze procedure and software comprehensibility measurement. *IEEE transactions on software engineering*, (5):608–623, 1986.

[10] John Johnson, Sergio Lubo, Nishitha Yedla, Jairo Aponte, and Bonita Sharif. An empirical study assessing source code readability in comprehension. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 513–523. IEEE, 2019.

[11] Luca Pascarella, Magiel Bruntink, and Alberto Bacchelli. Classifying code comments in java software systems. *Empirical Software Engineering*, 24(3):1499–1537, 2019.

[12] Valentina Piantadosi, Fabiana Fierro, Simone Scalabrino, Alexander Serebrenik, and Rocco Oliveto. How does code readability change during software evolution? *Empirical Software Engineering*, 25:5374–5412, 2020.

[13] Daryl Posnett, Abram Hindle, and Premkumar Devanbu. A simpler model of software readability. In *Proceedings of the 8th working conference on mining software repositories*, pages 73–82, 2011.

[14] Pooja Rani, Arianna Blasi, Nataliia Stulova, Sebastiano Panichella, Alessandra Gorla, and Oscar Nierstrasz. A decade of code comment quality assessment: A systematic literature review. *Journal of Systems and Software*, page 111515, 2022.

[15] Simone Scalabrino, Mario Linares-Vásquez, Rocco Oliveto, and Denys Poshyvanyk. A comprehensive model for code readability. *Journal of Software: Evolution and Process*, 30(6):e1958, 2018.

[16] Simone Scalabrino, Mario Linares-Vasquez, Denys Poshyvanyk, and Rocco Oliveto. Improving code readability models with textual features. In *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, pages 1–10. IEEE, 2016.

[17] Sean Stapleton, Yashmeet Gambhir, Alexander LeClair, Zachary Eberhart, Westley Weimer, Kevin Leach, and Yu Huang. A human study of comprehension and code summarization. In *Proceedings of the 28th International Conference on Program Comprehension*, pages 2–13, 2020.

[18] Daniela Steidl, Benjamin Hummel, and Elmar Juergens. Quality analysis of source code comments. In *2013 21st international conference on program comprehension (icpc)*, pages 83–92. Ieee, 2013.