A Replicated Study on Factors Affecting Software Understandability

Georgia M. Kapitsaki[1][a], Luigi Lavazza[2][b], Sandro Morasca[2][c] and Gabriele Rotoloni[2][d]
[1]*Department of Computer Science, University of Cyprus, Nicosia, Cyprus*
[2]*Dipartimento di Scienze Teoriche e Applicate, Università degli Studi dell'Insubria, Varese, Italy*

Keywords: Software Code Understanding, Code Static Metrics, Replicated Experiments.

Abstract: *Background*. Understandability is an important characteristic of software code that can largely impact the effectiveness and cost of software maintenance. *Aim*. We investigate if and to what extent the characteristics of source code captured by static metrics affect understandability. *Method*. We replicated an empirical study which provided some insights and highlighted some code characteristics that seem to affect understandability. The replication took place in a different country and was conducted with a different set of developers, i.e., Bachelor's students, instead of Master's students. The same source code was used in both studies. *Results*. The data collected in the replication do not corroborate the results of the initial study, since no correlation between code measures and code understanding could be found. The reason seems to be that the initial study involved developers with very similar skills and experience, while the replication involved developers with quite different skills. *Conclusions*. Code understanding appears to be affected much more by developers' skills than by code characteristics. The extent to which code understanding depends on code characteristics is observable only for a homogeneous population of developers. Our results can be useful for software practitioners and for future software understandability studies.

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

1 INTRODUCTION

Software code understanding often requires a large amount of time and effort (Minelli et al., 2015; Xia et al., 2017). To reduce the amount of resources needed for code understanding, it would be very useful to recognize which parts of software code are difficult to understand, so as to improve that code to make it more easily understandable and maintainable. The availability of understandability models could also lead to guidelines for writing understandable code.

Specifically, it would be very useful to be able to assess code understandability based on structural characteristics of the code, as represented by static measures. Many code measures, such as the number of Lines of Code, McCabe's Cyclomatic Complexity (McCabe, 1976), Halstead's measures (Halstead, 1977) and Maintainability Indices (Heitlager et al., 2007; Oman and Hagemeister, 1992) can be used for this purpose.

In 2023, an empirical study was carried out with the goal of investigating whether various source code measures could be useful to build accurate understandability prediction models (Lavazza et al., 2023). The time needed to correctly complete some code maintenance tasks was used as a measure of code understandability. It is reasonable to expect that the more understandable the code, the easier and faster the debugging (note that, in the empirical study, code correction was generally trivial, once the defect had been identified). To minimize the impact of developers' capability and experience on the maintenance time, a set of developers with similar experience and capability was selected: the participants were Master's students from the University of Insubria at Varese, Italy. The models obtained from the empirical study's data indicate that code understandability depends on structural code properties.

Since the original empirical study involved only three participants, the results of that study can only be viewed as preliminary indications. The relationship between structural code properties and understandability needs to be explored further. Therefore, in 2024, we carried out a replication of the empirical study, addressing the following Research Questions (RQs):

[a] https://orcid.org/0000-0003-3742-7123
[b] https://orcid.org/0000-0002-5226-4337
[c] https://orcid.org/0000-0003-4598-7024
[d] https://orcid.org/0000-0003-2046-0090

Kapitsaki, G. M., Lavazza, L., Morasca, S. and Rotoloni, G.
A Replicated Study on Factors Affecting Software Understandability.
DOI: 10.5220/0013339500003928
In *Proceedings of the 20th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2025)*, pages 582-591
ISBN: 978-989-758-742-9; ISSN: 2184-4895
Copyright © 2025 by Paper published under CC license (CC BY-NC-ND 4.0)

- **RQ1:** Is it possible to define practically useful models for code understandability based on source code measures?
- **RQ2:** If the answer to RQ1 is positive, to what extent are the models obtained similar to those from the 2023 empirical study? If the answer to RQ1 is negative, which factors appear to affect code understandability?

The replicated study involved 7 undergraduate (Bachelor' s level) students, following the Computer Science program at the University of Cyprus located in Nicosia, Cyprus. They were asked to carry out the same realistic maintenance tasks that were assigned to the participants in the original 2023 empirical study. These tasks involved the correction of 28 methods from two real-life Open Source Software applications. Like in the former empirical study, understandability was measured as the time needed to correctly complete a maintenance task. Since the code used in the study is the same used in the original study, we used the code measures already collected and available from the replication package. The analysis of the collected data was performed like in the previous study (using the same analysis scripts, which were available from the replication package).

Unlike in the 2023 empirical study, the data collected in the new study do not provide any evidence that code understandability is related to structural characteristics of code. This lack of relationship happens if code understandability is indeed much more related to the other factors that intervene when understanding software code. In fact, it must be noted that in experiments code understandability is evaluated by observing and measuring code *understanding*, which depends on the characteristics of both the object to be understood (the software code) and the subject that tries to understand it (the professional who is required to understand the code). Code characteristics appear to be only a second-order set of influencing factors. In the replication, the developer population of the new empirical study turned out to be not as homogeneous as in the original one; these differences appear to affect understanding much more than differences in software code characteristics.

To confirm the above explanation, we looked for similar studies on code understandability that had revealed little or nonexistent correlation with code properties. We found that also in a relevant prior study, the involved developers were very different from each other in terms of skills and experience.

The remainder of the paper is organized as follows. Section 2 provides some background on code understandability and its measurement, including a summary of the original 2023 empirical study. The new empirical study and its results are illustrated in Section 3. Section 4 reports the results of our analysis based on data from a similar study that failed to correlate understandability to code measures. The answers to the Research Questions are in Section 5. The threats to the validity of the empirical study are discussed in Section 6. Section 7 accounts for related work. Finally, Section 8 illustrates the conclusions and outlines future work.

2 BACKGROUND

2.1 Source Code Understandability

Maintenance and evolution activities absorb a large part of software development. Quite often, maintainers are not familiar with the source code they have to modify, hence they have to go through a potentially difficult and expensive code understanding phase before they are able to work on the code and be productive.

Both this study and the original one deal with understandability. Maintainability and understandability are external software properties (Fenton and Bieman, 2014), since they involve the relationships of software with its "environment," usually maintainers. Therefore, there are many ways to measure understandability, depending on the objectives and characteristics of software development and the related research. Specifically, we use time (the time taken to understand software code is used as proxy of understandability (Ajami et al., 2019; Börstler and Paech, 2016; Dolado et al., 2003; Hofmeister et al., 2019; Peitek et al., 2020; Salvaneschi et al., 2014; Scalabrino et al., 2019; Siegmund et al., 2012)) and correctness (related to the result of maintenance tasks that require understanding code (Börstler and Paech, 2016; Dolado et al., 2003; Salvaneschi et al., 2014; Scalabrino et al., 2019; Siegmund et al., 2012)).

As described in detail in Section 2.3, we measure understandability via the time needed to correctly perform a maintenance activity (namely, bug fixing) on a software method, since in the experiments the time needed to fix a defect was negligible with respect to the time needed to find it.

2.2 Source Code Measures

Multiple measures are available to represent internal software properties, i.e., the properties of code that can be measured based only on the code itself (Fenton and Bieman, 2014). These measures are useful when they are associated with a quality of interest,

like understandability, that concerns the development process or the software product (Fenton and Bieman, 2014; Morasca, 2009). In this study, we have considered the same code measures as the original study, which are some of the most popular code measures used in the research literature and the software industry, as described below.

- **Logical Lines of Code:** The number of lines of code (*LOC*) is by far the most widely used measure of software size. Logical *LOC* (*LLOC*) accounts only for non-empty and non-comment code lines.
- **McCabe's Complexity:** *McCC* is used to indicate the complexity of a program, being the number of linearly independent paths through a program's source code. It is generally used to measure software complexity, hence understandability and maintainability.
- **Nesting Level Else-If:** Nesting Level Else-If (*NLE*) measures the maximum nesting depth of a method's code block.
- **HVOL:** Halstead proposed several code metrics (Halstead, 1977), based on the total number of occurrences of operators N_1, the total number of occurrences of operands N_2, the number of distinct operators η_1 and the number of distinct operands η_2. Halstead Volume (*HVOL*) is defined as follows:

$$HVOL = (N_1 + N_2) \cdot log_2(\eta_1 + \eta_2) \quad (1)$$

- **HCPL:** Halstead Calculated Program Length (*HCPL*) is defined as follows:

$$HCPL = \eta_1 \cdot log_2(\eta_1) + \eta_2 \cdot log_2(\eta_2) \quad (2)$$

- **Maintainability Index** is defined as follows (Welker et al., 1997):

$$MI = 171 - V - M - L \quad (3)$$

where

$$V = 5.2 \cdot ln(HVOL) \quad (4)$$

$$M = 0.23 \cdot (McCC) \quad (5)$$

$$L = 16.2 \cdot ln(LLOC) \quad (6)$$

- **Cognitive Complexity** was introduced in 2018 as a new measure for code understandability. It takes into account the number of decision points weighted according to their nesting level, the structure of Boolean predicates and several aspects of code. For a complete description of Cognitive Complexity, the reader can refer to the definition (Campbell, 2018).

These metrics were collected in the original study and reused in our replicated study.

2.3 The Original Empirical Study

We here summarize the original empirical study, whose full details are available in (Lavazza et al., 2023). A replication package is also available online at

http://www.dista.uninsubria.it/supplemental_material/understandability/replication_package.zip.

To evaluate code understandability in realistic conditions, participants were given tasks resembling parts of the actual work carried out by professional programmers, concerning Java code of non-trivial size and complexity that were selected for the empirical study. To make the results as independent as possible of the context and the participants, the original study involved a quite homogeneous set of participants, working in a scenario resembling real-world development. Specifically, the participants were Master's students in Computer Science, all having similar levels of knowledge of the coding language and similar levels of programming experience. In practice, the proficiency in Java programming of participants was similar to that of junior professionals (Carver et al., 2010).

As in most studies addressing code understandability (Oliveira et al., 2020), also in the original study understandability was characterized via two measures: the overall time required to solve a maintenance task, and the correctness of the proposed solution. However, no time limit was enforced, and all participants were able to successfully complete all tasks. Thus, code understandability was finally measured only via the time taken by each participant to produce a correct solution.

Each participant was given a set of faulty methods, with the objective of removing their defects. Every method was equipped with a set of unit tests, so that correctness could be quickly evaluated. For each method, participants had to locate the fault in the method, devise a way to correct the faulty method, perform the correction, and test the modified code by running the available test cases. The code used in the empirical study consisted of 14 methods from the *JSON-Java* project (JSO, 2022) and 14 methods from the *Jsoniter* project (jso, 2022).

The empirical study was carried out in two sessions, lasting four hours each, in different days, to avoid fatigue effects. In each session, each participant had to perform the corrective maintenance of eight methods (four from *JSON-Java* and four from *Jsoniter*). Half of the methods were assigned to multiple participants, to be able to compare participants on the basis of common assignments.

Participants used the Eclipse IDE on their own

machine, and they could take breaks, whose duration was not counted in the task execution time. Participants were instructed not to communicate with each other, and they were also informed that they were not being evaluated in any way via the empirical study. To make the environment as friendly as possible, sessions were supervised by another Master's student.

Different participants obtained similar results for common methods. Also the mean times taken by participants to complete all the tasks assigned were similar. No participant was consistently better or worse than the others: as in real organizations each developer performed better than colleagues in some tasks and worse in others.

Models of task completion time as a function of code measures were built using several techniques, namely Support Vector Regression (SVR), Random Forests (RF) and Neural Networks (NN). Models used up to four code features, to avoid overfitting. The obtained models were evaluated based on the Mean Absolute Residual (MAR), also known as Mean Absolute Error, and the mean of relative residuals, which is the ratio of MAR and the mean of the considered property (in this case, the task completion time).

The models built with different techniques were similarly accurate. The greatest majority of the obtained models had relative errors in the 27%–32% range. This result indicates that there is a correlation between the measured characteristics of code and understandability; however, the extent of the error indicates that understandability depends also on other factors, not quantified by the considered measures.

3 THE REPLICATION

The replicated empirical study took place in April 2024 at the University of Cyprus in Nicosia, Cyprus, with the participation of undergraduate students attending the Bachelor program of Computer Science at the Department of Computer Science. The third and fourth year students were informed of the empirical study via email, and interested students indicated their willingness to participate. The prospective participants were informed that no personal data would be used in any way (actually, no personal data were collected). Only the results of the tasks would be used, with no connection to the identity or other characteristics of who carried out the task. All participants gave their consent to the above. The empirical study was not linked to any specific course in the Bachelor program or any grading process, but a small monetary reward was given to each participant regardless of the result of the empirical study.

The supplemental material of the new empirical study is available online at https://anonymous.4open.science/r/code-understandability-replicated-experiment-0E6B/README.md. Since the original study was already provided with the replication package, our supplemental material includes only the raw data collected and the results of the experiment. The Java projects and the Java methods used in the new empirical study are as in the original one.

The original study suggests that there can be multiple factors that affect code understandability, beyond those represented by the considered static code metrics. These additional factors are out of the scope of the new study, which replicates the original study, seeking confirmation of the previous findings.

3.1 Organization of the Empirical Study

The empirical study was executed in one four-hour session, with all seven participants: participants were all third and fourth year undergraduate students that had regularly followed the Computer Science Bachelor's program, consisting of compulsory and elective courses. Concerning their prior knowledge, all participants had attended a course on object-oriented programming, one course on software engineering (and in total three courses that used Java as main programming language); however, participants had attended a different set of elective courses based on their preferences and the year of study, so their programming competences might differ. One of the authors was responsible for conducting the empirical study and was present in the room when it took place, but did not intervene in any of the assigned tasks.

At the beginning of the four-hour session, instructions were given to the participants to replicate the settings of the original empirical study. The participants were given access to the Open Source Java projects (*JSON-Java* and *Jsoniter*) used in the original empirical study, and were asked to perform maintenance tasks, i.e., bug fixing on the code: each participant was given eight methods to fix (four methods from each software project), whereas some methods were assigned to more than one participant. 14 methods from each project were used in total.

The participants were requested to document the time needed to correctly complete the tasks. Specifically, they were asked to document the time spent to identify and fix the bug. To check whether a bug was correctly fixed, unit tests were used, as in the original empirical study. Participants were also free to take breaks that did not count towards the task completion time. The Eclipse IDE was used as in the original study and participants were free to use resources on

Table 1: Descriptive statistics of the Java methods used in the empirical study.

	Time	CoCo	HVOL	HCPL	McCC	LLOC	NLE	MI
mean	28.1	16.5	936.1	271.2	10.7	33.6	2.8	78.2
st. dev.	11.8	10.9	418.4	97.0	6.5	14.8	1.5	11.2
median	26.5	12.0	838.7	254.1	10.0	31.0	3.0	77.0
min	9	2	244	104	1	10	0	59
max	77	43	1956	522	28	68	7	105

the Internet but they were not allowed to use generative AI tools, such as ChatGPT.

3.2 Replication Results

At the end of the empirical study, the participants provided access to their code and the supervisor of the study verified whether the tasks were successfully completed. The time used in the results analysis, measured in minutes, was self-reported by the students. In what follows, we describe the relationship between some selected code measures and the time taken to accomplish the task. Not all the assigned tasks were successfully completed; tasks that were attempted but not successfully completed are also positioned with respect to the measure of the code.

We built Neural Networks models of successful completion times, using code metrics as features: we used two metrics at the time, to avoid overfitting. The models' hyperparameters were chosen using the same tuning process as in the original study. The best models were obtained with the pairs (MI,Cognitive Complexity), and (McCC,Cognitive Complexity); when evaluated via Leave One Out Cross-validation, the mean relative error was 42% and 48%, respectively. These results do not seem representative of a relationship between code measures and understandability, especially considering that the aforementioned error rate accounts only for tasks that were successfully completed.

Figure 1 and Figure 2 show task completion time and task failures as a function of LOC, HVOL, McCC and LLOC. It is easy to see that statistical tests are hardly necessary to conclude that there is no correlation. No correlation was found with the other metrics mentioned in Section 2 either.

The lack of correlation illustrated by Figure 1 and Figure 2 suggests that code understanding was affected mainly by other factors than the code characteristics represented by the considered metrics. To check if understanding was affected by participants' different performances, we computed the success rate (the ratio between the numbers of successfully completed and assigned tasks) and the mean completion time for all participants. Fig. 3 shows how each participant is positioned in the (success rate, mean time) plan.

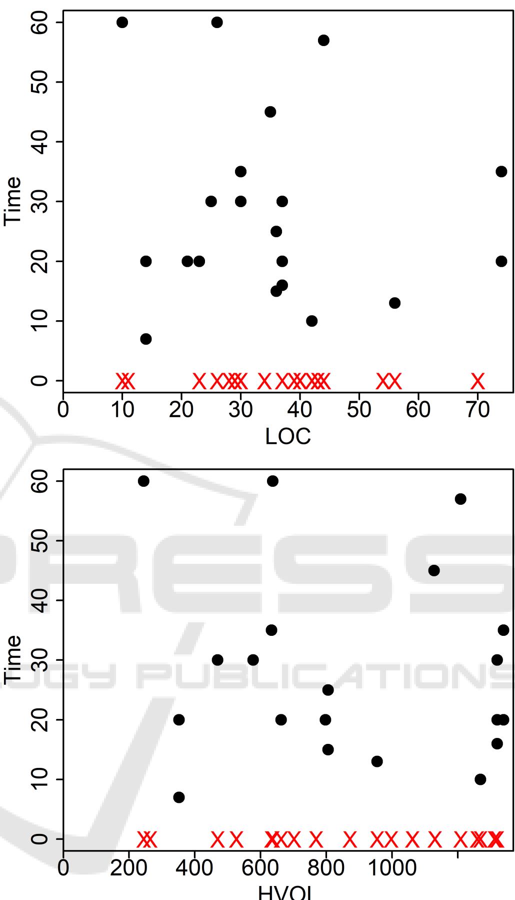

Figure 1: Task completion time (circles) and task failures (red Xes) as a function of LOC and HVOL.

It is easy to see that one participant achieved a relatively high success rate along with the lowest mean completion time, but several participants with lower success rates had quite different mean completion times. In practice, Fig. 3 tells that participants performed quite differently.

However, we could wonder if the different performances described in Fig. 3 were due to differences in the difficulties of the assigned tasks. Thus, we compared the success rates and completion times of the tasks that were assigned to multiple participants and were solved by at least one participant. Table 2 shows that 8 tasks were solved by one participant and failed

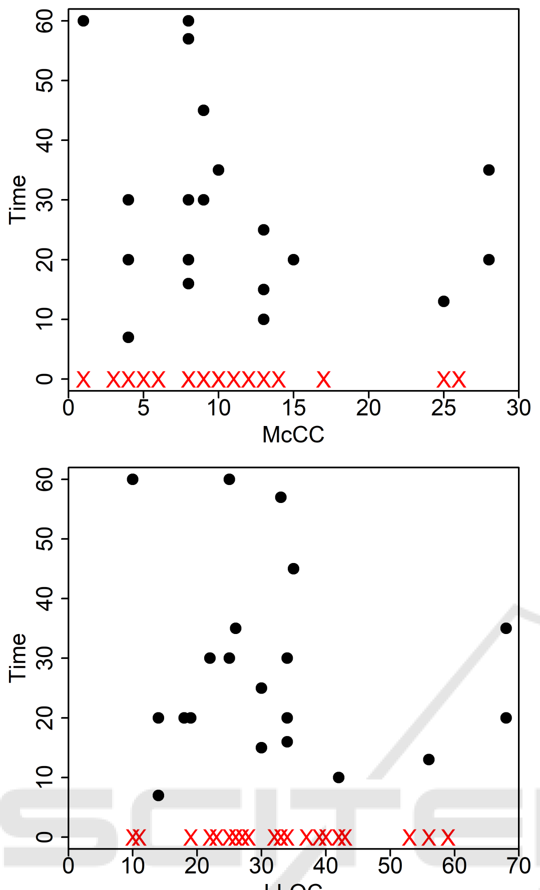

Figure 2: Task completion time (circles) and task failures (red Xes) as a function of McCC and LLOC.

by one or more others. Table 3 shows the minimum and maximum completion times of tasks that were successfully completed by multiple participants: the maximum time is often close to twice the minimum time.

Table 2: Tasks that were assigned to more that one participant and were successfully solved by only one.

Method name	# failures
createDecoder(String cacheKey, Type type)	1
findStringEnd(JsonIterator iter)	3
toJSONObject(String string)	1
nextMeta()	1
read()	1
stringToValue(String string)	1
unescape(String string)	1
enableDecoders()	1

Based on the observations summarized in Tables 2 and 3, we can conclude that participants actually performed quite differently, independent of the characteristics of the source code they had to understand. This seems to imply that the structural characteristics

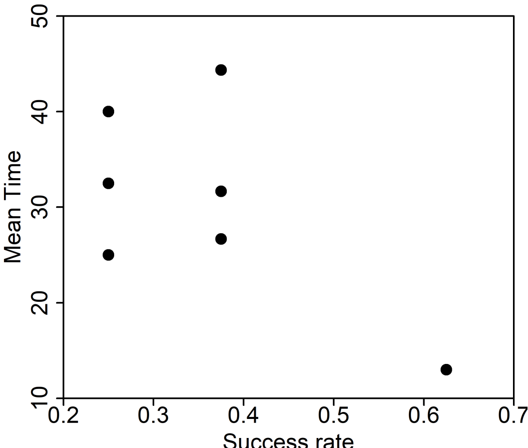

Figure 3: Participants' performances measured via the success rate and the mean task completion time.

Table 3: Tasks that were successfully completed by multiple participants.

Method name	# successes	min time [minutes]	max time [minutes]
objectToBigInteger	2	15	25
nextToken	2	20	35
updateBindingSetOp	4	16	30
skipFixedBytes	2	7	20

of code we measured are much less important than personal skills when assessing code understandability.

SCITEPRESS
SCIENCE AND TECHNOLOGY PUBLICATIONS

4 ANALYSIS OF DATA FROM OTHER EMPIRICAL STUDIES

To seek confirmation to the hypothesis stemming from the results described in Section 3, i.e., that large differences in developers' skills dominate code characteristics in determining understanding performance, we looked for publicly available data from other empirical studies and repeated the analyses described in the previous sections.

We analyzed the dataset by Scalabrino et al. (Scalabrino et al., 2019). In their empirical study, developers were asked to read pieces of code and then, if they thought they understood it, they answered three questions about the code. For our analysis, we used the time needed to understand the code as the time (TNPU in their study, measured in seconds). Code understanding was considered successful when the developer answered at least two of the three questions correctly ($ABU_{50\%}$ in their study). Like in our study, code measures do not appear correlated with understanding time and success. This result is consistent with the observations by Scalabrino et al. (Scalabrino

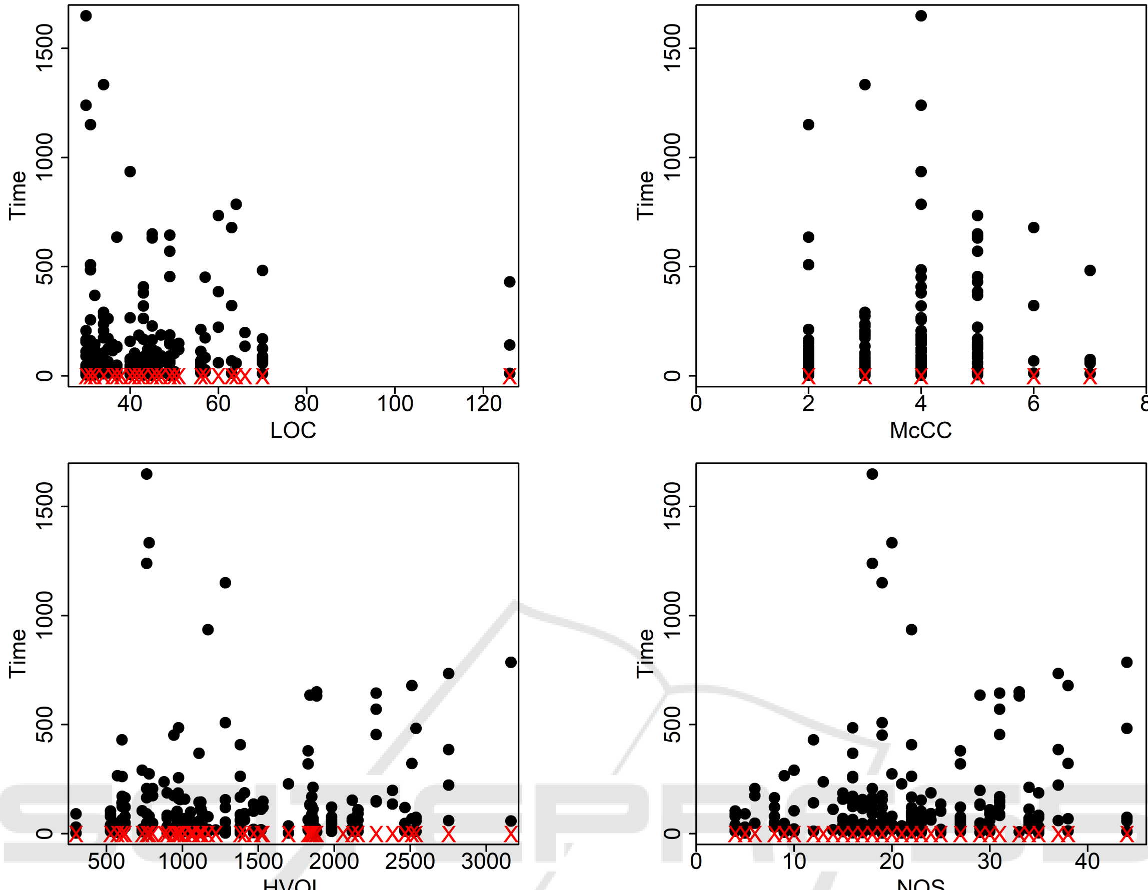

Figure 4: Task completion time (circles) and task failures (red Xes) in (Scalabrino et al., 2019) as a function of LOC and HVOL.

Figure 5: Task completion time (circles) and task failures (red Xes) in (Scalabrino et al., 2019) as a function of McCC and NOS.

et al., 2019). In their paper, they also built models with multiple metrics. The models show some discriminatory power for some proxies of understandability, but they are not good enough to be used in practice.

Figure 4 and Figure 5 show understanding times and failed tasks vs. HVOL, McCC, LOC, and the number of statements (NOS):[1] it is easy to see that there is hardly any correlation between understanding success or understanding time on the one side and code characteristics on the other side. To check whether this lack of correlation is due to different skills and consequent performances of participants, we proceeded as in the previous section: Figure 6 represents the success rate and mean understanding time for each participant. It is apparent that there is a wide variability in both success rate and mean times.

[1]Scalabrino et al. did not measure LLOC, hence we used the number of statements, which quantifies a fairly similar property of code.

Figure 7 represents the understanding time for successful attempts, where each value in the x-axis represents a task. Times longer than 500 were cut off to make the plot more readable. For each task, the understanding times vary widely, i.e., different participants took quite different times to understand the same piece of code: this confirms that tasks did not play a significant role in determining the mean time for the participants.

5 ANSWERS TO RESEARCH QUESTIONS

The analysis described in the previous sections let us answer our RQs.

RQ1. Is It Possible to Define Practically Useful Models for Code Understandability Based on Source Code Measures?

While the original study answered positively to

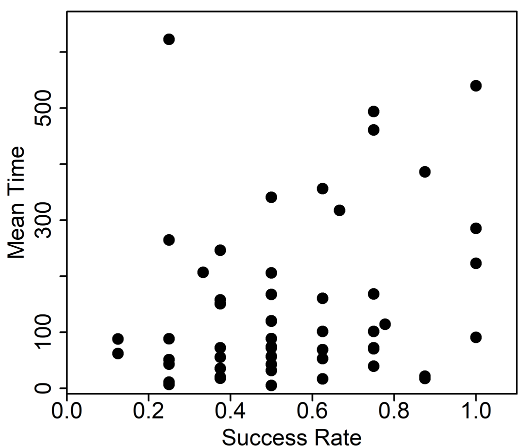

Figure 6: Participants' performances in (Scalabrino et al., 2019), measured via the success rate and the average task understanding time.

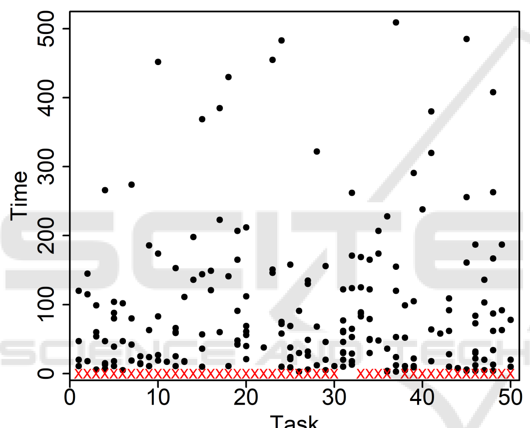

Figure 7: Understanding time for successful attempts (Os) and failures (Xes) for each task in (Scalabrino et al., 2019).

this question, in the replicated study—and in a prior empirical study from the literature—no models could be derived. In fact, a clear lack of correlation between code measures and understandability was apparent.

RQ2. If the Answer to RQ1 is Negative, Which Factors Appear to Affect Code Understandability? Differences in various developers' skills appear prevalent in determining code understanding with respect to code characteristics. In the replication, there was more variability in the participants' level, compared to the original empirical study. Different participants attended different Computer Science courses and were enrolled in different years of study. Noticeably, the difficulty to build understandability models based on code measures was observed also in similar studies characterized by participants having different skills and experience levels.

6 THREATS TO VALIDITY

Our results are based on a limited number of participants (seven participants) in a specific cultural and educational setting, so they might not generalize in other contexts or settings, affecting thus *external validity*. We focused only on corrective maintenance, so our findings may not immediately generalize across other types of software maintenance. However, this kind of generalization is not a goal of the paper: maintenance activities are considered only because they involve code understanding. At any rate, other maintenance activities necessarily involve code understanding: measuring those activities is likely to provide different results, but not significantly different ones, as long as code understanding is the core activity of the observed activities.

The presented study—like the original one—used students as participants. The undergraduate students who participated in the study also work as programmers in an internship program at the University of Cyprus, so their use in empirical studies may be considered somewhat also representative of junior professionals (Carver et al., 2010). However, empirical studies that use students have been sometimes criticized, because students may not represent well the entire gamut of skills and experiences of professional developers in industry settings. This criticism does not apply to our study, which focuses on highlighting possible relationships between code understandability and the characteristics of code represented by a set of static metrics. The point is that skilled and experienced professionals would probably understand the given code faster and better than students, but the time taken by professionals would still be longer for less understandable code than for easy code. The same relationship (probably involving longer times) would characterize understanding by students. Take for instance methods MC and MS, with MC being substantially more difficult to understand than MS. The average understanding time for professional will be $T_{pro,MC} > T_{pro,MS}$ for MC and MS, respectively. We expect student to take $T_{stu,MC} > T_{pro,MC}$ and $T_{stu,MS} > T_{pro,MS}$, but still with $T_{stu,MC} > T_{stu,MS}$.

We do not expect that *internal validity* is affected in any way, as the analysis techniques employed rely on widely used libraries of the R programming language (e.g., `dplyr`) that were also used in the original empirical study. Concerning *construct validity*, we used defect detection and correction time as a proxy of understandability, because understanding is a necessary and dominant component of these activities. This measure has been used in several studies in the literature. Also, as we already argued, correc-

tion time was negligible compared to defect detection time, which required the most part of the understanding effort. As for the time required to complete tasks and for whether an attempt was made for a specific method, we relied on what was self-reported by the participants. This subjective measure may have affected our results. At any rate, the participants did not have any interest in voluntarily biasing the self-reported times in any way. As for *conclusion validity*, it may be argued that other code measures may be correlated with understandability. While this may be true in principle, we used a set of well-known code measures that have been widely used in the past and quantify different code characteristics, such as software size, control-flow complexity, and maintainability.

7 RELATED WORK

In a recent paper, Ribeiro et al. address the contradictions in literature on how different code characteristics influence code readability and understandability (Ribeiro et al., 2023). Ribeiro et al. thought that these contradictions are caused by the interchangeable usage of the terms readability and understandability, and the different level of experience of the participants. To test this, they conducted three empirical studies in order to address the influence of comments, indentation spacing, identifiers' length, and code size on readability and comprehensibility, on developers with different levels of experience. Despite some findings with statistical significance, the controlled variables of their studies are not sufficient to explain the contradictions in literature. However, a positive trend between the presence of comments and comprehension was found in studies with novice participants, but not in the study with only experienced participants. When asked about this finding, the experienced participants answered that they did not even read them, because, as they indicated, "*comments cannot be trusted*."

Many empirical studies with human participants have been conducted, in the area of code comprehension, where a systematic mapping study has focused on the analysis of 95 published papers (Wyrich et al., 2023). Oliveira et al. (Oliveira et al., 2020) analyzed different studies and showed how readability is measured in different ways, and more than 16% of them only used personal opinions as a metric, and 37% exercised a single cognitive skill, e.g., memorization. Only few papers simulated real-world scenarios, requiring more cognitive skills. Our study tries to simulate a real-world scenario by asking Bachelor's students in their senior years of study, hence with more experience, to understand and fix code.

8 CONCLUSIONS

We have presented a replicated study on code understandability, which involved participants with different characteristics (Master's students in the original versus Bachelor's students in the replicated experiment). While the original study produced relatively accurate models of understandability vs static code measures, in the replicated study no correlations were found between the time required to complete the tasks and the code metrics. Contrary to the original study, the participants of the replicated study have different skills and experience. We argue that in the process of code understanding, the characteristics of developers play a bigger role than the characteristics of code. The study of relevant prior works confirms that it is hardly possible to create consistent models for code understandability based on source code measures, when the participants in the understanding activities are heterogeneous with respect to skills and experience.

As future work, we intend to replicate the empirical study in more locations involving also developers from the industry, in order to be able to draw conclusions from a wider set of participants. We would also like to expand to different source code understandability tasks, beyond bug fixing, such as software reuse activities.

ACKNOWLEDGEMENTS

This work has been partially supported by the "Fondo di ricerca d'Ateneo" of the Università degli Studi dell'Insubria.

REFERENCES

(2022). GitHub - json-iterator/java: jsoniter (json-iterator) is fast and flexible JSON parser available in Java and Go.

(2022). GitHub - stleary/JSON-java: A reference implementation of a JSON package in Java.

Ajami, S., Woodbridge, Y., and Feitelson, D. G. (2019). Syntax, predicates, idioms - what really affects code complexity? *Empir. Softw. Eng.*

Börstler, J. and Paech, B. (2016). The role of method chains and comments in software readability and comprehension - an experiment. *IEEE Trans. Software Eng.*

Campbell, G. A. (2018). Cognitive complexity - a new way of measuring understandability.

Carver, J. C., Jaccheri, L., Morasca, S., and Shull, F. (2010). A checklist for integrating student empirical studies with research and teaching goals. *Empir. Softw. Eng.*

Dolado, J. J., Harman, M., Otero, M. C., and Hu, L. (2003). An empirical investigation of the influence of a type of side effects on program comprehension. *IEEE Trans. Software Eng.*, 29(7):665–670.

Fenton, N. E. and Bieman, J. M. (2014). *Software Metrics: A Rigorous and Practical Approach, Third Edition*. Chapman & Hall/CRC Innovations in Software Engineering and Software Development Series.

Halstead, M. H. (1977). *Elements of software science*. Elsevier North-Holland.

Heitlager, I., Kuipers, T., and Visser, J. (2007). A practical model for measuring maintainability. In *QUATIC conference 2007*.

Hofmeister, J. C., Siegmund, J., and Holt, D. V. (2019). Shorter identifier names take longer to comprehend. *Empir. Softw. Eng.*, 24(1):417–443.

Lavazza, L., Morasca, S., and Gatto, M. (2023). An empirical study on software understandability and its dependence on code characteristics. *Empirical Software Engineering*.

McCabe, T. J. (1976). A complexity measure. *IEEE Transactions on software Engineering*.

Minelli, R., Mocci, A., and Lanza, M. (2015). I know what you did last summer-an investigation of how developers spend their time. In *2015 IEEE 23rd ICPC*.

Morasca, S. (2009). A probability-based approach for measuring external attributes of software artifacts. In *Proceedings of ESEM '09*.

Oliveira, D., Bruno, R., Madeiral, F., and Castor, F. (2020). Evaluating code readability and legibility: An examination of human-centric studies. In *IEEE ICSME 2020*.

Oman, P. and Hagemeister, J. (1992). Metrics for assessing a software system's maintainability. In *Proceedings in ICSM 1992*.

Peitek, N., Siegmund, J., Apel, S., Kästner, C., Parnin, C., Bethmann, A., Leich, T., Saake, G., and Brechmann, A. (2020). A look into programmers' heads. *IEEE Trans. Software Eng.*, 46(4):442–462.

Ribeiro, T. V., dos Santos, P. S. M., and Travassos, G. H. (2023). On the investigation of empirical contradictions-aggregated results of local studies on readability and comprehensibility of source code. *Empirical Software Engineering*.

Salvaneschi, G., Amann, S., Proksch, S., and Mezini, M. (2014). An empirical study on program comprehension with reactive programming. In *Proceedings of FSE 2014*.

Scalabrino, S., Bavota, G., Vendome, C., Poshyvanyk, D., Oliveto, R., et al. (2019). Automatically assessing code understandability. *IEEE Trans. on Software Eng.*

Siegmund, J., Brechmann, A., Apel, S., Kästner, C., Liebig, J., Leich, T., and Saake, G. (2012). Toward measuring program comprehension with functional magnetic resonance imaging. In *Proceedings of FSE 2012*.

Welker, K. D., Oman, P. W., and Atkinson, G. G. (1997). Development and application of an automated source code maintainability index. *Journal of Software Maintenance: Research and Practice*.

Wyrich, M., Bogner, J., and Wagner, S. (2023). 40 years of designing code comprehension experiments: A systematic mapping study. *ACM Computing Surveys*.

Xia, X., Bao, L., Lo, D., Xing, Z., Hassan, A. E., and Li, S. (2017). Measuring program comprehension: A large-scale field study with professionals. *IEEE Transactions on Software Engineering*.

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS