How Clear is Your Code?
An Empirical Study with Programming Challenges

Eduardo Fernandes, Luiz Paulo Ferreira, Eduardo Figueiredo, Marco Tulio Valente

Department of Computer Science, Federal University of Minas Gerais, Belo Horizonte, Brazil

{eduardofernandes, luizpcf, figueiredo, mtov}@dcc.ufmg.br

Abstract. To maintain a software system, developers have to read and properly understand the source code. Previous work confirms that developers spend too much time reading code before maintaining it. Clear code is the one that is easy to read, to understand and, consequently, to maintain. Since readability and understandability are subjective properties of code, to understand how developers characterize clear code may support maintainability. Software metrics, such as Source Lines of Code (SLOC) and McCabe's Complexity (McCabe), are the basic means to quantify clear code properties. However, we still lack empirical knowledge on the correlation between such metrics and the subjective view of developers on clear code. In this paper, we investigate how developers characterize clear code by analyzing 6,775 alternative code solutions to 131 programming challenges from CheckIO, an online coding challenge platform. In CheckIO, developers submit supposedly clear code solutions to a challenge and other developers provide positive votes to the clearest ones. We aim to identify correlations between the number of positive votes for a clear solution and six metrics: SLOC, McCabe, Number of Commented Lines, Comments Rate, Developer Experience, and Days After 1st Solution. Our results suggest a high correlation between clear code and the two last metrics. We also observed a higher correlation of clear code with SLOC and McCabe when the programming challenges have few (less than 20) solutions. However, we could not find correlation of clear code with Number of Commented Lines and Comments Rate.

Keywords: Software Maintainability, Clear Code, Software Metrics

1 Introduction

To maintain a software system, developers spend a significant effort reading the source code [4, 20]. Previous work shows that developers initially use approximately a quarter of their time reading code before maintaining it [13]. In fact, before moving to other development activities, such as code implementation and testing, developers have to read and properly understand the existing code [10]. Therefore, code has to be clear. In this paper, *clear code* is the source code that is easy to read, to understand and, consequently, to maintain [8, 19]. Since readability and understandability are subjective properties of code, to understand how developers characterize clear code may support saving maintenance efforts.

Software metrics aim to estimate the maintainability of code, including clear code properties [14]. Empirical studies apply different metrics, e.g., *Source Lines of Code (SLOC)* and *McCabe's Complexity (McCabe)*, to assess the code quality [1, 5]. For instance, previous work [1] proposes a technique for maintainability assessment via metrics. Another previous study [5] compares different techniques for quantifying maintainability with software metrics. However, to the best of our knowledge, we did not find empirical studies on the correlation between metrics and the subjective view of developers on clear code that rely on large data sets and several developers.

In this paper, we address the aforementioned gap with an empirical study on the main properties of clear code as perceived by developers in CheckIO [2]. CheckIO is a popular online coding challenge platform for Python and JavaScript, with more than 132 thousand users. In CheckIO, developers propose novel programming challenges and submit solutions to the existing ones. Developers can categorize each of their own solutions as a *Clear*, a *Creativ*e, or a *Speedy* solution. In addition, other developers can provide positive votes for each existing solution that better fits the solution's category. Our study targets on both Python and *Clear* solutions, i.e., the ones that are easy to read and to understand according to the CheckIO platform [2].

We analyze 6,775 *Clear* solutions for 131 challenges collected from CheckIO. We compute, per solution, four metrics: *SLOC* [9], *McCabe* [17], *Number of Commented Lines (Comments)* [15], and *Comments Rate (%C)* – i.e., the number of commented lines divided by the total number of lines of the solution. We also consider two metrics provided by CheckIO: *Developer Experience (Experience)* and *Days After 1st Solution (DA1st)*. The former increases as the developer solves challenges and the latter counts the number of days of a solution after the submission of the first solution for a challenge. To be fair in our analysis, we split the challenges into two sets: challenges with few (i.e., ≤ 20) solutions and challenges with many (i.e., > 20) solutions.

As a result, our data suggest that, the more experienced the developer is, the clearer is the code. The correlations obtained for *Experience* and *DA1st* support this finding. We also obtained a positive correlation to some extent in the cases of *SLOC* and *McCabe*. This correlation was stronger when there were few solutions for a single programming challenge. Meanwhile, since we could not find correlation for *Comments* and *%C*, we conclude that both metrics do not correlate with clear code. In summary, experienced developers submit the clearest code solutions for challenges in CheckIO, and such solution tend to have few lines of code and non-complex logic.

The remainder of this paper is organized as follows. Section 2 provides background information. Section 3 describes the study settings. Section 4 presents the analysis of the data distribution. Section 5 presents the analysis of correlation between clear code and the selected metrics. Section 6 discusses related work. Section 7 presents threats to the study validity. Finally, Section 8 concludes the paper and suggests future work.

2 Background

This section presents background information of the paper. Section 2.1 presents CheckIO. Finally, Section 2.2 discusses clear code in CheckIO.

2.1 CheckIO

Released in 2012, CheckIO is an online coding challenge platform to support learning of programming in Python and JavaScript [2]. Developers can submit code solutions for existing programming challenges in the platform. They also can propose novel challenges to the users. Challenges have four increasing difficulty levels: *Elementary*, *Simple*, *Moderate*, and *Challenging*. Thus, developers can unlock harder challenges as their skills evolve. CheckIO has more than 130 thousand developers and about 200 programming challenges with almost 500 thousand solutions proposed by the developers. The platform also provides an integrated coding environment.

Developers can categorize their code solutions as *Clear*, *Creative*, and *Speedy*. According to CheckIO website, a *Clear* is the one that is easy to read, to understand, and to maintain, in accordance with the definition of *clear code* used in our empirical study. In a leaderboard fashion, CheckIO ranks developers based on experience levels, from one (beginner) to 20 (most experienced). The computation of levels relies on the number of solved challenges, the number of novel challenges proposed by the developer, and the number of positive votes received for submitted solutions. Each solved challenge grants different points to the developer level according to its difficulty.

2.2 Clear Code

Clear code is the one that is easy to read and to understand [8, 19]. Since developers tend to spend less time to understand a code that is clear than other, clear code supports the software maintenance. However, both readability and understandability are subjective properties of code. Aimed at understanding how developers characterize clear code, we investigate the subjective view of developers on clear code via CheckIO. Thus, given the programming challenges available at the online platform, we collect and analyze only the *Clear* code solutions. As an example, let us consider the programming challenge of computing the median value of an array of integer numbers.

Listing 1 presents a simplistic solution in Python for the aforementioned challenge. This solution received 238 positive votes and it is the most voted as *Clear* for such challenge. It solves the problem in three steps. First, it sorts the array of integers in line 5. Second, it sums the two centering elements of the array in line 6. Third, it divides the sum by 2 in line 7. If the array has an odd number of integers, the solution is correct because the two centering elements are the same. This solution has six lines of code, no comments, and no branches. We then assume that the small solution size and the low cyclomatic complexity may relate to the votes as *Clear* code.

Listing 1. The most voted *Clear* solution for the median challenge

```
1. from math import floor, ceil

2. def CheckIO(data):
3.     off = len(data) // 2
4.     data.sort()
5.     med = data[off] + data[-(off + 1)]
6.     return med / 2
```

3 Study Settings

Our study goal is to understand how developers characterize clear code. We rely on the analysis of different code solutions for programming challenges. We compute the correlation between six metrics and the number of votes that the solutions received as *Clear* in CheckIO. To support our empirical study, we designed three steps. Sections 3.1, 3.2, and 3.3 discuss the steps *Data Collection, Code Measurement,* and *Analysis*.

3.1 Data Collection

In the *Data Collection* step, we developed a Web crawler to collect data from CheckIO. We collected all challenges (131 at that time), the public solutions in Python per challenge (6,775 solutions), and the number of positive votes per solution. We also collected the experience level of developers in CheckIO, as the submission date per solution to compute *DA1st* (Section 3.2). We had no access to the actual professional experience of developers. Thus, we analyzed their experience level in CheckIO. A high level means that the developer solved several hard challenges and it has many positive votes. We then assume that, the higher the level, the more experienced the developer is. In the platform, developers can see solutions only for challenges solved by them, but the CheckIO team gently provided us access to all challenges and solutions.

3.2 Code Measurement

In the *Code Measurement* step, we computed metrics per solution collected in *Data Collection*. We grouped the metrics into three categories: *Size and Complexity*, *Documentation*, and *Others*.

Size and Complexity. The concision of code may be relevant to assess the code maintainability. That is, the common sense suggests that small-sized code solutions are easier to understand than large-sized code solutions. However, it does not imply that a clear solution is necessarily the smallest one. Aimed at investigating the correlation between clear code and code size, we selected two metrics for analysis. *Source Line of Code (SLOC)* [9] is the number of lines of code in a source file, discarding blank lines and comments. In turn, *McCabe's Cyclomatic Complexity (McCabe)* [17] measures the code complexity in terms of the number of alternative execution paths in a file.

Documentation. Previous work suggests that developers often write comments to hide code quality problems [10]. However, comments can also be considered a good programming practice for code documentation [11]. To investigate both facets of comments, we selected two metrics for analysis: *Number of Commented Lines (Comments)* and *Comments Rate (%C)* [9, 15]. Comments count the total number of commented code lines. On the other hand, %C is the number of commented lines divided by the total number of lines of code in a file.

Others. Although this paper focuses on code metrics, we analyzed additional aspects that may affect the subjective view of clear code. We then selected two metrics: *Developer Experience (Experience)* and *Days After 1st Solution (DA1st)*. Experience indicates the experience level of the solution submitter and DA1st counts the number of days of a solution after the submission of the first solution for the target challenge.

3.3 Analysis

In the *Analysis* step, we conducted two steps discussed as follows. In Section 4, we present the distribution of the collected metrics for the code solutions. We aim to identify the most appropriate metrics for characterizing clear code. As an example, although a given metric may have a heavy-tail distribution, the majority of the available solutions have similar values (i.e., high or low values) [6, 16]. Thus, a heavy-tail distribution suggests that the metric is inappropriate to identify *Clear* solutions. Otherwise, the metric may be useful to identify *Clear* solutions. In Section 5, we discuss the correlation between each metric and the number of votes that the code solutions received as *Clear*.

4 Analysis of Distribution

In this section, we present the analysis of distribution per metric. Section 4.1 presents our data set. Sections 4.2, 4.3, and 4.4 present, respectively, the analysis of metrics regarding *Size and Complexity*, *Documentation*, and *Others*.

4.1 Data Set

Our data set has 131 programming challenges and 6,775 code solutions. Note that 115 out of the 131 challenges (about 88%) have less than one hundred solutions, and only three have more than 500 solutions. The highest number of solutions for a challenge is 628 and the median is 20. That is, about half of the challenges have 20 or less solutions. Overall, the distribution of solutions per challenge is left-skewed, i.e., the challenges have mostly few solutions. With respect the difficulty of all challenges, about 54% of them are *Simple* or *Elementary* (the easiest ones), and 46% of them are *Challenging* or *Moderate* (the hardest ones). When considering only the challenges with up to 20 solutions, the respective percentages are 65% and 35%. The easiest challenges usually have more solutions than the hardest ones. In fact, the harder challenges are only unlocked as the developer levels increase. Thus, few developers can solve the hardest challenges.

Due to two reasons, we decided to analyze challenges with 20 or less solutions separately. First, we could split our data set into the groups *Few* and *Many*, described in Table 1, with almost the same number of challenges (66 and 65, respectively). Second, a list of solutions in CheckIO has up to 20 solutions of length. Thus, we assume that several developers may have ignored larger lists and they only looked at solutions in the first page. Table 1 presents two challenge groups: *Few*, i.e., challenges with up to 20 solutions, and *Many*, i.e., challenges with more than 20 solutions. In addition, there

are four subgroups described as follows. *Few-None* and *Many-None* indicate code solutions that did not receive positive votes as *Clear*, while *Few-Clear* and *Many-Clear* indicate solutions that received at least one positive vote as *Clear*.

Table 1. Groups of challenges and the number of available code solutions

Group	Description	Solutions
Few	**Programming challenges with 20 or less solutions**	**709**
Few-None	Challenges with up to 20 solutions and no vote	551
Few-Clear	Challenges with up to 20 solutions and at least one vote	158
Many	**Programming challenges with more than 20 solutions**	**6066**
Many-None	Challenges with more than 20 solutions and no vote	5520
Many-Clear	Challenges with more than 20 solutions and at least one vote	546

4.2 Distribution of Size and Complexity

This section analyzes the distribution of *SLOC* and *McCabe*. Table 2 presents the distribution of both metrics considering all 6,775 solutions for the 131 challenges. The first and second columns inform the metrics and the challenge categories. The third and fourth columns present the minimum (Min) and maximum (Max) values for the metrics. The four following columns provide the first quantile (Q1), the median (Median), the third quartile (Q3), and the standard deviation (SD) of the distribution. From Table 2, we observed that *SLOC* and *McCabe* follow heavy-tail distributions in the four categories. Thus, their averages are insignificant since the metrics mostly assume low values.

Table 2. *SLOC* and *McCabe* of all solutions grouped by challenge category

Metric	Category	Min	Max	Q1	Median	Q3	SD
SLOC	Few-None	1	424	13	24	39	33.52
	Few-Clear	1	596	10	17.5	28.75	54.68
	Many-None	1	239	7	11	18	12.48
	Many-Clear	1	155	6	11	18	12.87
McCabe	Few-None	0	104	5	9	16	11.37
	Few-Clear	0	207	3	7	12	17.7
	Many-None	0	58	2	4	7	4.6
	Many-Clear	0	39	2	4	6	4.22

Overall, there is almost no difference between the distributions in the groups *Many-None* and *Many-Clear*. Thus, we assume that neither *SLOC* nor *McCabe* has influence on the votes as *Clear* to solutions for challenges with more than 20 solutions (the *Many* group). This finding confirms our assumption in Section 3.2 that a small-sized solution is eventually harder to read than a large-sized solution. The same reasoning is valid for *McCabe*. In turn, there is a notable difference between the groups *Few-None* and *Few-Clear*. This difference suggests a possible impact of *SLOC* and *McCabe* on the number of *Clear* votes if challenges have few (i.e., 20 or less) solutions. Note that, since unexperienced developers are unable to solve the hardest challenges, challenges with few solutions are generally the most complex ones.

Summary of Section 4.2. *SLOC* and *McCabe* follow heavy-tail distributions, but a slight difference between groups suggests a potential for indicating clear code.

4.3 Distribution of Documentation

This section analyzes the distribution of *Comments* and *%C*. Table 3 presents the distribution of both metrics per category (Table 1). As observed in Section 4.2, both *Comments* and *%C* have heavy-tail distributions with values mostly close to zero. It indicates that developers in CheckIO often do not comment their code. In fact, we expected a few commented lines per code solutions, since each solution has generally one class and one method. In addition, we could not observe expressive differences among groups for both metrics, except regarding SD and Max. Thus, we conclude that number of comments is not a relevant indicator for a solution voted as *Clear*.

Table 3. *Comments* and *%C* of all solutions grouped by challenge category

Metric	Category	Min	Max	Q1	Median	Q3	SD
Comments	Few-None	0	167	0	1	6	12.52
	Few-Clear	0	193	0	1	7	22.18
	Many-None	0	57	0	0	4	4.96
	Many-Clear	0	56	0	1	5	6.93
%C	Few-None	0	73.33	0	2.41	18.47	14.8
	Few-Clear	0	66.67	0	5.88	24.38	17.41
	Many-None	0	92.86	0	0	23.08	20.65
	Many-Clear	0	89.74	0	4	27.65	21.96

Summary of Section 4.3. Both *Comments* and *%C* have heavy-tail distributions and, therefore, they probably do not suffice to indicate clear code.

4.4 Distribution of Others

This section analyzes the distribution of *Experience* and *DA1st*. Table 4 presents the distribution of both metrics per challenge category (see Table 1). Since our conclusions are different for each metric, we divided our discussion as follows.

Table 4. *Experience* and *DA1st* of all solutions grouped by challenge category

Metric	Category	Min	Max	Q1	Median	Q3	SD
Experience	Few-None	7	20	12	14	15	2.15
	Few-Clear	9	20	14	16	17	2.12
	Many-None	1	20	6	9	11	3.52
	Many-Clear	2	20	9	12	15	4
DA1st	Few-None	0	623	12.5	77	232	139.54
	Few-Clear	0	581	0	2	52	140.05
	Many-None	0	1288	98	252	457	209.65
	Many-Clear	0	1098	2	78	304	198.66

Experience. Table 4 suggest that this metric follows a normal (or close to normal) distribution. We observed that the metric in the groups *Few-Clear* and *Many-Clear* (i.e., the solutions voted are *Clear*) are higher than in the groups *Few-None* and *Many-None* (i.e., the solutions with no vote). As an example, considering only challenges with less than 20 solutions (the *Few* group), developers of the solutions voted as *Clear* tend to be more experienced. In fact, CheckIO enables the hardest challenges only for the most experienced developers to solve. Thus, such developers have more chance to receive

votes in the challenges of the *Few* group. We then conclude that, the more experienced the developers are, the more votes their solutions receive.

DA1st. Table 4 suggests the opposite to *Experience* for *DA1st*. The solutions submitted closer to the first submitted solution (i.e., the ones with lowest values of *DA1st*) tend to receive more positive votes as *Clear* than others. Two reasons may justify that situation. First, new solutions have had less time to receive votes. That is, fewer days since their submission. Thus, even if the new solution is better than the previous ones, it probably will not receive several votes. Second, CheckIO sorts the solutions first by the number of positive votes and, then, by the date of submission. Consequently, if a developer submitted a solution earlier, this solution appears to more developers and may receive more votes. In fact, 55% of the solutions were submitted up to 6 months after the first submitted solution and have only 29% of the positive votes. Meanwhile, solutions with more than 6 months sum 45% of all solutions and are responsible for 71% of the solutions with votes. Only 16% of the solutions where solved within 1 month after the first submitted solution and have 48% of the votes. Finally, only in the first week, we have 9% of solutions with 38% of the positive votes.

Summary of Section 4.4. Both *Experience* and *DA1st* have (close to) normal distributions and, therefore, they potentially suffice to indicate clear code.

5 Analysis of Correlation

In this section, we present the analysis of correlation between each metric of Section 3.2 and the number of positive votes as *Clear* of the solutions. Sections 5.1, 5.2, and 5.3 present our results per type of metrics: *Size and Complexity*, *Documentation*, and *Others*. We computed the Pearson's correlation coefficient [12] that ranges in the interval [-1, 1]. A value close to -1 indicates a negative correlation, i.e., solutions with low metric values are more prone to receive votes as *Clear*. A value close to 1 indicates a positive correlation, i.e., solutions with high metric values are more prone to receive the votes. A value close to 0 indicates no correlation between the metric and the votes.

5.1 Correlation of Size and Complexity

This section discusses the correlation between the number of positive votes as *Clear* and two metrics: *SLOC* and *McCabe*. Figure 1 presents the correlation between code solutions with at least one vote as *Clear* and *SLOC*. In Figure 1, the dotted line represents the programming challenges in the *Few* group, i.e., the challenges with 20 or less code solutions. In turn, the regular line represents the challenges in *Many* group, i.e., challenges with more than 20 code solutions. We computed the correlation for each programming challenge, considering all the respective code solutions votes as *Clear*.

Our data suggest that mostly of the challenges have a weak negative correlation for *SLOC*, since the Pearson's coefficient is usually close to 0. In fact, the correlation is concentrated in the interval [-0.4, 0.2] for challenges in both *Few* and *Many* groups. Regarding the challenges with 20 or less solutions (*Few*), there is a significant negative

correlation. In fact, about 78% of the challenges have a negative correlation in [-1, 0], while only 22% of the challenges in the *Many* group have a positive correlation. Thus, we conclude that the negative correlation is stronger in challenges with 20 or less solutions. It suggests that, when developers compare just a few solutions for a single challenge, they often consider as *Clear* the one with lower SLOC.

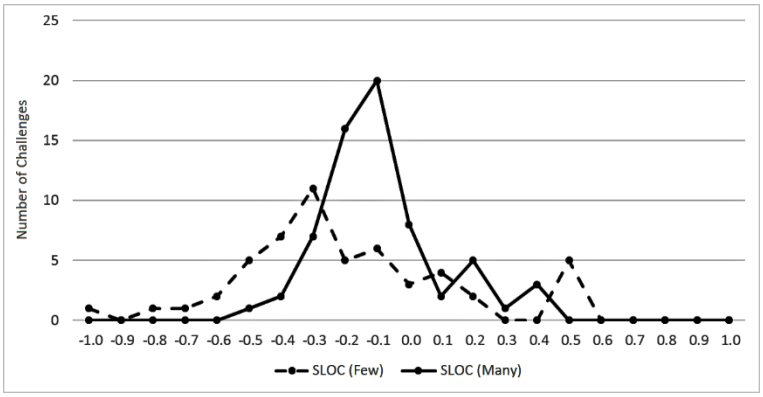

Fig. 1. Correlation between *Source Lines of Code (SLOC)* and *Clear* votes

Figure 2 presents the correlation between the number of positive votes as *Clear* and *McCabe*. As in Figure 1, the dotted line represents challenges in the *Few* group, while the regular line represents challenges in the *Many* group. Overall, the correlation for *McCabe* follows a similar trend to *SLOC*. The correlation between *McCabe* and the number of votes as *Clear* is usually weak. However, the challenges with 20 or less solutions, i.e., in the group *Few*, have a stronger negative correlation than challenges with more than 20 solutions, i.e., in the *Many* group.

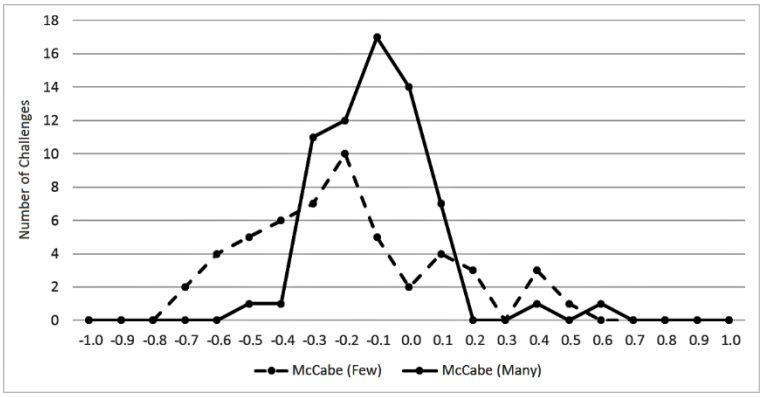

Fig. 2. Correlation between *McCabe's Complexity (McCabe)* and *Clear* votes

Summary of Section 5.1. We have found evidence that both *SLOC* and *McCabe* have a weak negative correlation with the number of votes as *Clear* when the challenges are small-sized and lower cyclomatic complexity.

5.2 Correlation of Documentation

Figure 3 presents the correlation between *Comments* and the number of votes as *Clear* for the code solutions. Similarly, Figure 4 presents the correlation between *%C* and the number of votes as *Clear*. In both figures, the dotted lines represent the programming challenges with 20 or less code solutions, while the regular lines represent the challenges with more than 20 code solutions. We discuss each figure as follows.

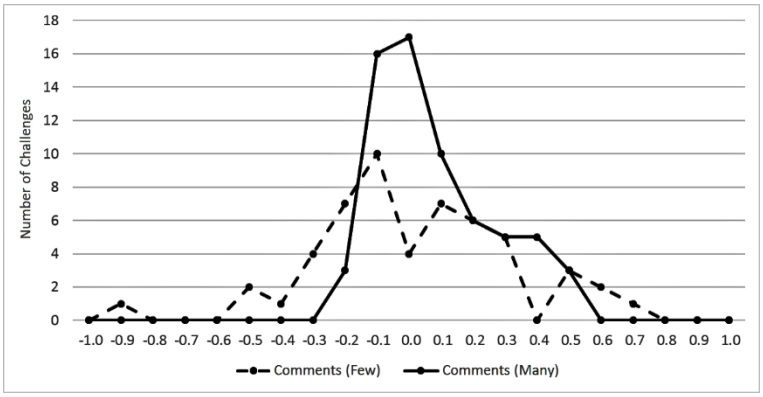

Fig. 3. Correlation between *Number of Commented Lines (Comments)* and *Clear* votes

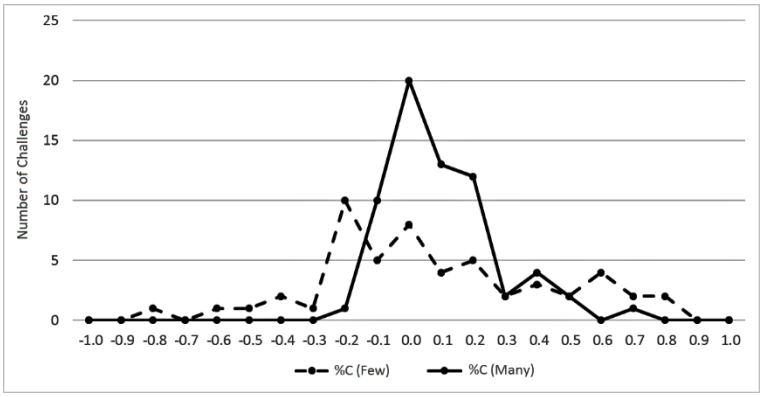

Fig. 4. Correlation between *Comments Rate (%C)* and *Clear* votes

From Figures 3 and 4, we could not find a correlation between the number of votes as *Clear* and both *Comments* and *%C*, since the values are concentrated around 0, specifically in the interval [-0.2, 0.3]. Due to the lack of a strong correlation, negative or positive, we are unable to claim that comments in the source code are sufficient indicators of clear code, as common sense suggests. In fact, we observed mostly a positive than a negative correlation (28.57% against 9.24%, respectively). Differently of *SLOC* and *McCabe*, we did not observe significant changes when selecting a subset of challenges, e.g., the ones with less than 20 solutions.

Summary of Section 5.2. We have found evidence that the impact of *Comments* and *%C* is generally low on the view of developers about clear code.

5.3 Correlation of Others

Figure 5 presents the correlation between *Experience* and the number of votes as *Clear* for the solutions. Figure 6 presents the correlation between *DA1st* and the number of *Clear* votes. In both figures, the dotted lines represent the challenges with 20 or less solutions (the *Few* group) and the regular lines indicate the challenges with more than 20 solutions (the *Many* group). Unlike the analyzed in Sections 5.1 and 5.2, *Experience* and *DA1st* have strong correlations with the *Clear* votes, with values mostly in the intervals [0.1, 0.7] and [-0.7, -0.2], respectively. Thus, *Experience* has a high positive correlation and *DA1st* has a negative correlation.

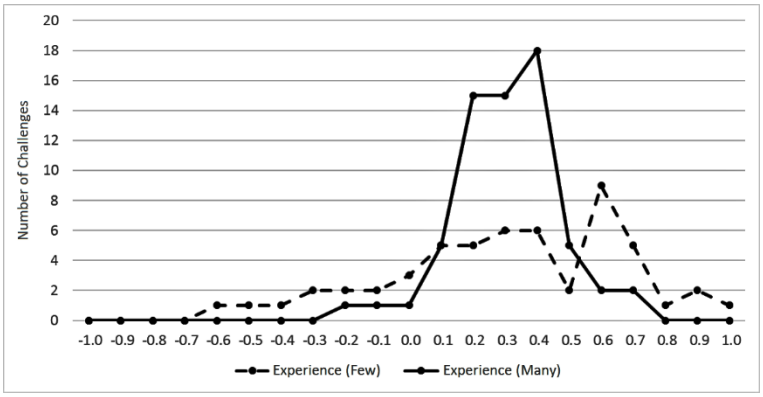

Fig. 5. Correlation between *Developer Experience (Experience)* and *Clear* votes

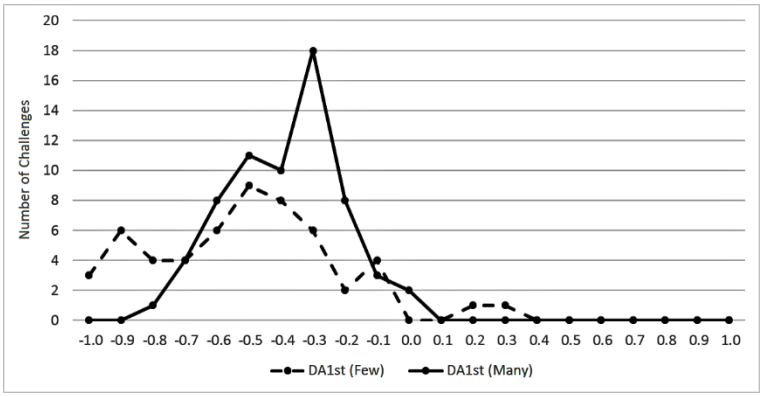

Fig. 6. Correlation between *Days After 1st Solution (DA1st)* and *Clear* votes

We remind that correlation does not imply a cause-effect relationship. For instance, the developer experience may increase in two ways at CheckIO. The first way is by

solving programming challenges that allows developer to solve the most complex challenges and, eventually, the less complex ones in a clearer manner. The second way is by receiving positive votes as *Clear* for the submitted solutions. In this context, note that the programming practices adopted by the experienced developers may influence the less experienced developers. Consequently, it may affect the metrics.

Summary of Section 5.3. Overall, the earlier submitted solutions (i.e., solutions with low *DA1st*), as well as the solutions provided by experienced developers (i.e., solution with high *Experience*), tend to receive more votes as *Clear* than other solutions.

6 Related Work

Previous work [1, 5, 7] investigates software metrics as indicators of the source code quality. For instance, a previous study [1] proposes a technique for source code analysis and quality assessment. Such technique aims to support the improvement of the software maintainability by relying on the analysis of source code metrics. To assess the maintainability of a system, the technique computes metrics aimed at measuring six code properties, including both size and complexity. Similarly, another previous work [5] compares five techniques for quantifying software maintainability with metrics. They also discuss the support of such techniques in saving maintenance efforts.

Finally, a previous study [7] proposes a technique based on data mining for analyzing software metrics and the source code quality. Their technique aims to support the understanding of metrics and their impact on different aspects of the software quality. The authors also discuss the difficulties of controlling the quality of systems, since the relationship between metrics and the software attributes, e.g., maintainability, is complex to assess. However, none of these papers present an empirical study on the correlation between software metrics and clear code, a specific source code property, specifically from the viewpoint of software developers.

In despite of that, we have found one previous work [21] that is the closest to ours. As in our empirical study, the author investigates the correlation between metrics and clear code. However, they focus on size and complexity metrics. In turn, our study also analyzes metrics regarding comments and the developer experience, for instance. Moreover, the previous work targets on undergraduate students in Computer Science. Meanwhile, our study analyzes code solutions collected from an online platform, in which unexperienced and experience developers submit their solutions to challenges.

7 Threats to Validity

We carefully conducted our study, but there are threats to the study validity. We discuss *construct*, *internal*, *conclusion*, and *external validity* [22] as follows.

Construct and Internal Validity. We designed our empirical study for replication. For instance, we described in detail the steps for data collection, metric computation, and data analysis (Section 3). However, a major threat relates to the set of computed metrics, since they may not suffice to indicate clear code properties. We are not concerned

on object-oriented metrics, e.g., coupling and cohesion, since the solutions available in CheckIO are not complete systems with a class hierarchy, for instance. They generally require no more than one class or method. Thus, coupling and cohesion metrics are out of our study scope. Due to the limited size of the code solutions, and to minimize this threat, we rely on traditional, largely used metrics, e.g., *SLOC* and *McCabe*. With respect to the data collection from CheckIO, we developed a Web crawler. To prevent errors in the data retrieval, we manually verified the integrity of the crawler and the collected solutions via code parsing.

Conclusion and External Validity. Regarding the data analysis, we rely on recurring techniques, as the analysis of metric distributions and the analysis of correlation based on the Pearson's coefficient [12]. In addition, two authors checked the integrity of the data analysis reports to minimize threats related to our analysis. However, we were not able to address a specific threat with respect to the analyzed code solutions. There may be several solutions for a single. This, solutions in the last pages of the solution list tend to receive less votes as clear. This threat may have affected the analysis results. Finally, regarding the generalization of findings, there are threats to consider. First, we analyzed only the code solutions developed in the Python language and available at CheckIO. Therefore, our results may apply mostly to Python code. However, we expect that the generalization of findings to other object-oriented language, although further studies are required. Second, we computed the correlation between metrics and clear code subjective view of developers. Since correlation does not imply causation, we may not generalize our findings to any other development context.

8 Conclusion and Future Work

In this paper, we assessed the subjective view of developers on clear code, i.e., a code that is easy to read, understand, and maintain. We analyzed 6,775 code solutions for 131 programming challenges available at CheckIO, an online coding challenge platform. In CheckIO, developers submit solutions to the existing challenges and may categorize each solution as *Clear* (clear code). Other developers provide positive votes for the clearest ones. We collected the available clear solutions to assess (i) the distribution of six metrics, e.g., Source Lines of Code and McCabe's Complexity, and (ii) the correlation between each metric and the number of *Clear* votes for the solutions.

As a result, we observed that earlier submitted solutions, as the ones provided by experienced developers, tend to receive more votes as *Clear* than others. In addition, specifically when there are a few solutions for a single challenge, the correlation between size or complexity metrics, such as *SLOC* and *McCabe*, and the number of positive votes as *Clear* is significant. Finally, we observed no correlation between comment-related metrics and positive votes as *Clear*. As future work, we intend to analysis other metrics, e.g., in the context of software reuse. We also intend to investigate the prediction of votes as *Clear* for new code solutions.

Acknowledgments. This work was partially supported by CAPES, CNPq (grants 424340/2016-0 and 290136/2015-6), and FAPEMIG (grant PPM-00382-14).

References

1. Baggen, R., Correia, J., Schill, K., Visser, J.: Standardized code quality benchmarking for improving software maintainability. Softw. Qual. J. **20**(2), 287–307 (2012). doi: 10.1007/s11219-011-9144-9
2. CheckIO – Online Game for Python and JavaScript Coders. https://checkio.org/
3. Chidamber S., Kemerer, C.: A metrics suite for object oriented design. IEEE Trans. Softw. Eng. **20**(6), 476–493 (1994). doi: 10.1109/32.295895
4. Coleman, D., Lowther B., Oman, P.: The application of software maintainability models in industrial software systems. J. Syst. Softw. **29**(1), 3–16 (1995). doi: 10.1016/0164-1212(94)00125-7
5. Coleman, D., Ash, D., Lowther, B., Oman, P.: Using metrics to evaluate software system maintainability. IEEE Comput. **27**(8), 44–49 (1994). doi: 10.1109/2.303623
6. Concas, G., Marchesi, M., Pinna, S., Serra, N.: Power-laws in a large object-oriented software system. IEEE Trans. Softw. Eng. **33**(10), 687–708 (2007). doi: 10.1109/TSE.2007.1019
7. Dick, S., Meeks, A., Last, M., Bunke, H., Kandel, A.: Data mining in software metrics databases. Fuzzy Set. Syst. **145**(1), 81–110 (2004). doi: 10.1016/j.fss.2003.10.006
8. Dromey, R.: A model for software product quality. IEEE Trans. Softw. Eng. **21**(2), 146–162 (1995). doi: 10.1109/32.345830
9. Fenton, N., Bieman, J.: Software metrics. Florida: CRC Press (2014)
10. Fowler, M.: Refactoring. New Jersey: Addison-Wesley Professional (1999)
11. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design patterns. New Jersey: Addison-Wesley Professional (1995)
12. Kitchenham, B.: Empirical studies of assumptions that underlie software cost-estimation models. Inform. Softw. Tech. **34**(4), 211–218 (1992). doi: 10.1016/0950-5849(92)90077-3
13. Ko, A., Aung, H., Myers, B.: Eliciting design requirements for maintenance-oriented IDEs. In: 27th ICSE, pp. 126–135. IEEE Press, New York (2005). doi: 10.1145/1062455.1062492
14. Lanza, M. and Marinescu, R.: Object-oriented metrics practice. Berlin: Springer Science & Business Media (2006)
15. Lorenz, M., Kidd, J.: Object-oriented software metrics. New Jersey: Prentice Hall (1994)
16. Louridas, P., Spinellis, D., Vlachos, V.: Power laws in software. ACM Trans. Softw. Eng. Methodol. **18**(1), 1–26 (2008). doi: 10.1145/1391984.1391986
17. McCabe, T.: A complexity measure. IEEE Trans. Softw. Eng. **2**(4), 308–320 (1976). doi: 10.1109/TSE.1976.233837
18. Munro, M.: Product metrics for automatic identification of "bad smell" design problems in Java source-code. In: 11th METRICS, pp. 15. IEEE Press, New York (2005). doi: 10.1109/METRICS.2005.38
19. Radatz, J., Geraci, A., Katki, F.: IEEE standard glossary of software engineering terminology. IEEE Std **610.12-1990** (121990), 1–83 (1990)
20. Riel, A.: Object-oriented design heuristics. New Jersey: Addison-Wesley Professional (1996)
21. Scott, T.: Size and complexity metrics and introductory programming students. J. Comput. Small Coll. **11**(2), 165–173 (1995)
22. Wohlin, C., Runeson, P., Höst, M., Ohlsson, M., Regnell, B., Wesslén, A.: Experimentation in software engineering. Berlin: Springer Science & Business Media (2012)