A New Perspective to Solving Technical Debt

Matthijs Blaauw (4925653)[1,a], Max de Froe (4496655)[2,b], Kouros Pechlivanidis (6527450)[3,c], and Tim Smit (6527906)[4,d]

[1,2,3,4]Department of Information and Computing Sciences, Utrecht University, Princetonplein 5, 3584 CC Utrecht, The Netherlands
[a]m.blaauw@students.uu.nl
[b]m.defroe@students.uu.nl
[c]k.pechlivanidis@students.uu.nl
[d]t.k.smit@students.uu.nl

February 7, 2022

Abstract

In 2020, a study by the Consortium for Information & Software Quality (CISQ) stated that defects caused by low-quality software development cost $2.08 trillion in the United States. Various studies take a qualitative approach towards researching antecedents that affect code quality. Taking a quantitative approach, this paper analyzes software team size, Git workflow and the lines of code per commit as factors influencing code quality. The QScored code quality dataset, contains code quality metrics of over $100,000$ open-source projects, and will thus be fit for analysis. The GitHub API was consumed to enrich the QScored dataset with the number of collaborators, the Git workflow and the average lines of code per commit. The data ($N = 5614$) was fitted to a linear regression model to test the effect of the stated antecedents on a project's code quality. Results show that the usage of branches and the average lines of code per commit are significant predictors of code quality. Code repositories with more branches had higher quality code. Codebases that used less lines of code per commit scored better in the QScored ranking system. Our work provides an actionable avenue towards further investigating code quality antecedents. A future research direction could be to find further optima of the discussed variables or to find more (hidden) variables. Lastly, these results could be used as a suggestion for the software programming industry.

Keywords: Code quality, Technical debt, Multiple regression analysis, Git workflow, Software team size, Github, QScored, Open-source.

1 Introduction

1.1 Research Context

In the current software industry, projects are developed under pressing time constraints and customer requirements are becoming increasingly complex [1]. To deal with these changing requirements, the quality of a project's source code is an important factor. The quality of a codebase often remains unknown until the code is tested, due to a lack of coding standards during development [2]. This study will focus on open-source software projects, a branch in the software industry where projects' source codes are openly accessible for anyone. In an open-source project, a product is developed without a manufacturer, meaning that the developers act as their own agents [3]. Open-source projects vary in size. Larger projects, such as the continuous integration tool Kubernetes, can have over three thousand contributors [4]. Collaboration between developers is often managed using a code repository service, such as SourceForge, Bitbucket, GitLab, or GitHub [5]. All of these services use Git, a version-control system that provides functionality for tracking, branching, merging, and managing code versions [6].

1.2 Problem Statement

In 2020, a study by the Consortium for Information & Software Quality (CISQ) stated that defects caused by low quality software development cost $2.08 trillion in the United States [7]. Guo, Spínola, and Seaman [8] also report costs related to solving defects caused by low-quality software. In this study, technical debt is defined as the result of the trade-off between time-to-market (TTM) and code-quality (CQ), where there is a positive relationship between the two variables. Guo, Spínola, and Seaman [8] conclude that the average cost of technical debt is $3.06 per line of code. Various studies imply that bad code quality not only results in malfunctioning pieces of software, but also in low system maintainability [9]–[12]. Thus, causing additional maintenance costs in the future, when requirements evolve.

1.3 Contribution

When trying to solve this technical debt dilemma, one should look for easily manipulable factors that contribute to CQ. Therefore this research looks into the effect of software production team size, Git usage, and lines of code (LoC) per commit on code quality.

By evaluating these described variables, this study aims to provide an actionable avenue towards improving code quality, so that developers can minimize time spent on solving technical debt.

1.4 Research Method

Our study will be conducted using a combination of two research methods. First, a literature review will be performed to gain a better understanding of the variables and instruments relevant to the research domain. In addition, the literature review will be used to justify the research approach and demonstrate a knowledge gap in the subject of the study. Second, the GitHub REST API will be consumed to enrich an existing code quality dataset with data on software production team size, Git usage, and LoC per commit. As this study aims to find the relationship between multiple independent variables and one dependent variable, the problem can be classified as a regression problem. During the literature review, existing code quality datasets and indices will be evaluated with the goal of selecting the most suitable dataset for this study. The main research question of this paper then goes as follows:

What easily manipulable factors related to software development influence code quality?

2 Related work

In this section, the context of our work will be introduced by examining related work. The concepts discussed in this section will be used to support the research approach in Section 3.

2.1 Team Size

There are various studies that assess the effect of team size on the work produced by a software team. A study by Fried [13] concluded that when the team size exceeds ten, the productivity of the team members drops, as team members spend more time interacting with each other. A different study by Mundra, Misra, and Dhawale [14] suggests that a development team should consist of a maximum of ten members, including testers and a product owner, as having a larger team makes the information flow difficult to manage. A third study found a negative relationship between software production team size and the ability to gain a thorough understanding of the entire codebase. When the team size grows, it is increasingly difficult to keep track of what each collaborator worked on, thus decreasing system-context understanding [13]. The discussed studies used developer productivity as a dependent variable, not considering code quality. Thus, demonstrating a knowledge gap in related work. A study more similar to our research was conducted by Brooks Jr [15], who concluded that the more people touch the same code, the higher the probability of defects is. Brooks Jr [15] explained this relationship by suggesting that if more developers have to collaborate, there may be miscommunication between engineers, leading to design and implementation failures. As code quality is a broader concept than defects only, a gap in knowledge regarding the effect of team size on the code quality of a codebase can be demonstrated.

2.2 Version Control Systems and Code Quality

Version control software (VCS) is essential for collaboration in software development projects. VCS tools facilitate collaboration by sharing files, tracking changes, flagging conflicts, restoring old versions, describing changes and more. Two types of version control software can be distinguished, the first being a centralized version control system having just one copy of a central repository where users need to be connected directly to this repository through a network. The second is decentralized version control software, where each user has a local copy of the repository, allowing for offline work. A network connection is only required when sharing a repository with other collaborators [16]. The latter is primarily used in Open-Source software, with Git being the prominent software used [2]. Earlier research found that the usage of VCS influences code quality. One study found that decentralized VCS contain commits of higher quality compared to centralized VSC [17].

An important benefit of DVCS is the ability to easily control local branches. Developers can cheaply create isolated branches that correspond to distinct tasks, such as development of a new features or bug fixes [18]. If done properly, branching facilitates faster development, and product quality [19]. However, research regarding the influence of the usage of branches on code quality return contradicting conclusions, as described next.

Several studies present the negative impacts of branches. An empirical study by Premraj et al. looked at best-practices for branching and concluded that incorrect usage results in high costs for propagating and merging changes [20]. A case study by Shihab, Bird, and Zimmermann analyzed branch usage during the development of Windows Vista and Windows 7. They found that branch activity and branch scatter both have a negative impact on post-release failures [21]. Bird and Zimmermann published a second paper the same year with a what-if analysis, concluding that a sub-optimal branch structure caused delays during the development of Windows' components [20]. These three studies are however all conducted in a closed-source development project, so these finding might not translate to open-source projects. All three papers hypothesize that the negative impacts of branches are caused by issues that arise during merging and discussed that these could occur due to organisational structures. Due to the nature of open-source projects these results can differ for open-source repositories

Two other studies that are both done in an education setting show a significant positive impact in code quality after using branches with a DVCS. A small-scale study took a traditional Git branching model used during software development and applied it in a university education setting to student development projects, which showed to decrease code quality issues [22]. Krusche, Berisha, and Bruegge [23] also did a study in a comparable setting with a larger sample size (300 students), they found that using branches to review code leads to higher code quality.

2.3 Code Quality Metrics

In assessing code quality of projects, different tools exist [24], [25]. A common factor in these is that they evaluate smells. Code smells (CSs) can be defined as "certain structures in the code that indicate violation of fundamental design principles and negatively impact design quality" [25]. CSs are not bugs per se, but indicate weaknesses that could hinder development progression or increase the risk of bugs later [26]. Labeling code smells is subjective, and may vary per language and development methodology [27]. Sharma and Kessentini [28] discerns three types of smells: architectural, design, and implementation. Manifold categories of CSs and CSs are put forth over the last 25+ years, as demonstrated by Sharma and Spinellis [29]. The most notable categories are architecture, design, and implementation. A subset of examples (most frequently studied [29]) of CSs are given in Table 1

Table 1: Common code smells, taken from Sharma and Spinellis [29].

Code Smell / Reference	**Description**
God class [30]	The god class smell occurs when a huge class which is surrounded by many data classes acts as a controller (i.e., takes most of the decisions and monopolises the functionality offered by the software). The class defines many data members and methods and exhibits low cohesion.
Feature envy [27]	This smell occurs when a method seems more interested in a class other than the one it actually is in.
Shotgun surgery [27]	This smell characterizes the situation when one kind of change leads to a lot of changes to multiple different classes. When the changes are all over the place, they are hard to find, and it's easy to miss an important change.
Data class [27]	This smell occurs when a class contains only fields and possibly getters/setters without any behavior (methods).
Functional decomposition [31]	This smell occurs when the experienced developers coming from procedural languages background write highly procedural and non-object-oriented code in an object-oriented language.

Smell detection tools or mechanisms can be categorized into five groups:

metrics, rules/heuristics, history, machine learning, and optimization-based detection. JSpIRIT [32], JDeodorant [33], and Designite [34] are examples of tools that automate smell detection. Detection of smells is, however, merely an operationalization of code quality with its shortcomings. In systematic reviews on tools [35] and smells in general, it can be synthesized that there is a "proneness to false-positives and poor coverage of smells detectable" [29]. Moreover Pantiuchina, Lanza, and Bavota [36] found that, after refactoring, quality metrics are frequently unable to capture the quality improvement as perceived by developers, limiting their accuracy.

2.4 Impacting Factors of Code Quality

Various existing studies have examined what factors have an impact on code quality. Bavota and Russo [37] for example, found that performing code reviews halves the chance of introducing bugs and overall, results in more readable code. Kochhar, Wijedasa, and Lo [38] have shown that adding new languages to a project significantly increases the change of bug fix commits and they identified specific languages that are more prone to bugs when used in multi-language projects. Other research by Butler, Wermelinger, Yu, *et al.* [39] suggests that not following identifier name guidelines increases the change of code quality issues. Lastly, Santos and Hindle [40] found that the "unusualness" of a commit message significantly correlated with a code's quality (more unusual implying worse quality).

3 Methodology

The goal of this section is to further describe our methodology, discussing the chosen goals, hypotheses, variables and data collection procedure.

3.1 Goals, Hypotheses and Variables

The aim of our study is to investigate how factors regarding software development influence CQ. To achieve this goal, two knowledge goals should be satisfied. First, The discovery of a set of factors related to the software development method which influence code is needed. Second, we require a subjective quantification of a software project's CQ such that we can compare the effect of individual independent variables.

The first knowledge question has already been answered when evaluating related work. As a result, the set of independent variables for this study will consist of the software production team size, Git usage, and lines of code (LoC) per commit. The list of independent variables, together with their definitions and scale are listed in Table 2. Git usage is represented as an ordinal variable instead of a ratio variable to combat extreme influences by outliers. The variable has three categories: 1 branch, 2 or 3 branches, and 4 or more branches.

Additionally, the categories better represent the Git usage, representing respectively: only working on the master branch (no other branch usage), working with feature and/or release branches, and having even more branches.

Table 2: List of independent variables

Variable	Definition	Scale
NoC	**Number of Contributors:** Number of GitHub users who have made a commit to the repository.	Ratio
Gu	**Git Usage:** Defined by active branches of a GitHub repository divided into three categories [1, 2-3, 4+].	Ordinal
LoCpC	**Lines of Code per Commit:** Average sum of additions and deletions per commit.	Ratio

The second knowledge goal dictates that the study objectives can only be attained through a measurement that is capable of capturing the code quality of a software project. Thakur, Kessentini, and Sharma [41] have developed an open platform called *QScored* that allows for code quality analysis. It has a large dataset of over 11 thousand repositories and ranks these based on the given code quality score. It takes in analyzed information generated by existing code smell detection tools Designite and DesigniteJava and determines the number of architectural, design and implementation smells. Based on this number it then calculates the code quality score by adding the total number of smells of each of the three categories in a weighted sum. The lower the quality score the better the code quality.

We have chosen to use QScored over other code quality estimation approaches because it is a free and open platform that enables us to analyze the large number of repositories in their dataset (over 109.000 repositories as of January 2022). Unlike FindBugs, a code quality analysis tool used by related work, QScored allows us to analyze both Java and C# based projects. And by looking at architectural, design and implementation smells it incorporates a wide variety of code quality indicators.

After elaborating on our study's knowledge goals, a total of three hypotheses can be presented. Table 3 describes the null hypotheses, whereas Table 4 lists the alternative hypotheses.

3.2 Data Collection Procedure

As described in Section 3.1, the QScored dataset will be chosen as a measurement for the dependent variable CQ. The QScored dataset only accommodates open-source repositories written in either C# or Java, meaning that the scope and context of our work will be limited to software projects with these characteristics. Furthermore, three additional criterion are established to select objects

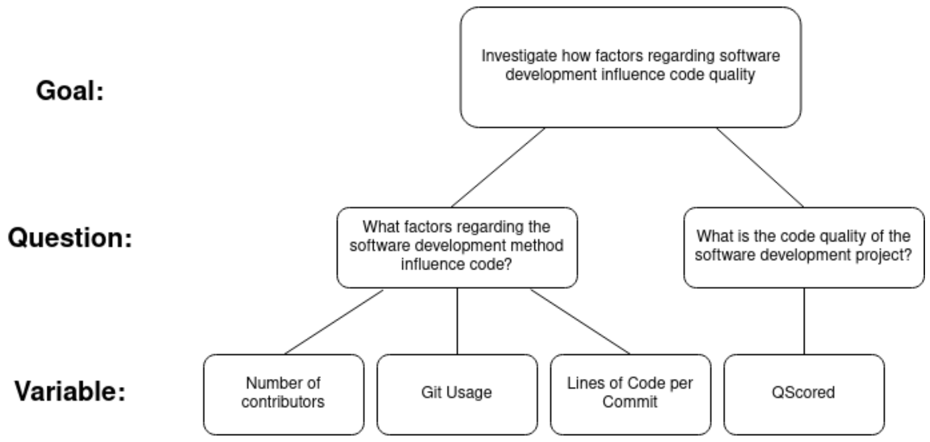

Goal:
Investigate how factors regarding software development influence code quality
Question:
What factors regarding the software development method influence code?
What is the code quality of the software development project?
Variable:
Number of contributors
Git Usage
Lines of Code per Commit
QScored

Figure 1: The GQM-model

Table 3: List of null hypotheses

Index	**Hypothesis**
$H1_0$	Software production team size is not a predictor of code quality
$H2_0$	Branch usage is not a predictor of code quality
$H3_0$	Lines of code per commit is not a predictor of code quality

Table 4: List of alternative hypotheses

Index	**Hypothesis**	**Source**
$H1_a$	Software production team size is a predictor of code quality	[13], [15]
$H2_a$	Branch usage is a predictor of code quality	[20]–[23]
$H3_a$	Lines of code per commit is a predictor of code quality	Proposed

eligible for analysis. First, only repositories that have created at least one release are considered. A release is a bundle of the source code used to notify users of changes in the software product incorporated as a new increment [42]. Filtering out repositories without releases ensures that all the examined objects are actively using Github as a means of managing the different iterations of their software product. Second, data will not be collected for repositories with only one week of commits, as this makes it likely that Github is not actively used for tracking increments to the software product. For the same reason, repositories that have less than seven commits per week are also descoped.

To enrich the QScored dataset with the variables described in Table 2, a Python API consumer was developed as an instrument to request data from the Github REST API [43]. Github provides dedicated endpoints for retrieving the number of contributors, branches and lines of code per commit. For tracking the lines of code per commit, additions and deletions are stored in separate variables. The acquired data will be stored in tabular format using pandas, a Python library used to create structured datasets [44].

The list of fields captured in the resulting dataset is shown in Appendix A.The source code used for producing the enriched dataset can be found on the Github page of the researchers which is presented in Appendix B.

4 Results

In this section, we will report the data accumulated in this study. In the first subsection, the dataset is introduced by providing descriptive statistics. All posed hypotheses are tested using an appropriate statistical test. The outcomes of these hypothesis tests are presented in the second subsection. Before analysis, records where the LoCpC variable was unknown were dropped.

4.1 Descriptive Statistics

In total, the QScored quality score was recorded for 2422 open-source Java projects, and 3192 open-source C# projects. The projects report an average QScored quality score of 15.21 (SD = 12.04). Figure 2 shows the distribution of QScored quality scores in a histogram.

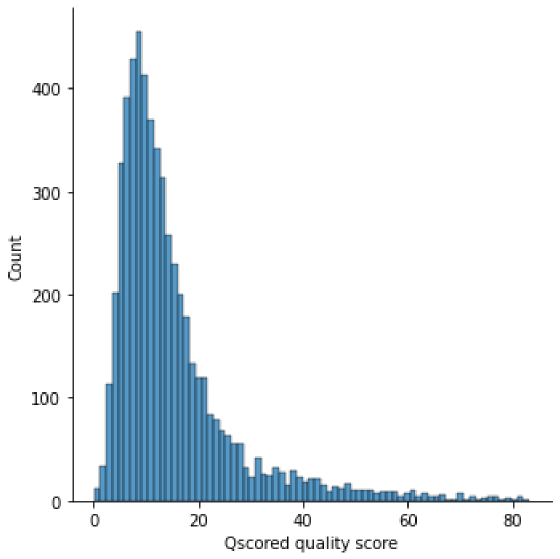


Figure 2: Distribution of QScored quality scores

Descriptive statistics of the two independent variables using a ratio scale (NoC and LoCpC) are presented in Table 5. As both variables do not follow a normal distribution, the mean and standard deviation are not representative of our data [45]. To provide a more meaningful descriptive analysis on the independent variables, the median, minimum, maximum, skewness, kurtosis and listed.

Table 5: Descriptive statistics for ratio independent variables

	Noc	**LoCpC**
M	9.72	3717.98
SD	26.61	41622.25
MDN	2	418.66
MIN	1	5.14
MAX	402	2346961.84
Skewness	6.79	38.61
Kurtosis	60.81	1912.20
N	5614	5614

Furthermore, the project's Git usage is quantified using an ordinal scale. Table 6 describes the reported QScored quality score aggregated on Git usage.

Table 6: Descriptive statistics for CQ, aggregated on Git usage

Git usage	**CQ Mean**	**CQ Standard Deviation**	**N**
1	16.80	13.44	1986
2	14.71	11.87	1742
3	14.00	10.36	1886

Lastly, to show there is no interaction between the independent variables, Table 7 demonstrates minimal correlations between them.

Table 7: Correlations between study variables

	CQ	**NoC**	**Gu**	**LoCpC**
CQ	1			
NoC	-0.041	1		
Gu	-0.100	0.260	1	
LoCpC	0.033	-0.020	-0.032	1

4.2 Inferential Statistics

For testing the hypotheses, multiple statistical tests were considered. First, a Pearson correlation test was considered to test whether the ratio variables are significantly correlated with the dependent variable [46]. For the ordinal variable Git usage, we considered using an ANOVA test, an appropriate test when testing between an ordinal independent variable with three or more groups and one continuous dependent variable [47]. When using a separate hypothesis test for each independent variable, the assumption is made that only the tested independent variable is manipulated. As this is an incorrect assumption, we examined tests that focused on comparing multiple independent variables with one dependent variable. A multiple logistic regression test is appropriate when testing on multiple continuous independent variables and one dependent variable following a categorical scale [48]. Moreover, we evaluated the multiple linear regression test, a suitable test for testing on multiple continuous independent variables and one continuous dependent variable [49]. As this test fits our variable model, we carried out a multiple linear regression test for hypothesis testing.

The conducted test was used to predict CQ based on NoC, Gu and LoCpC. The fitted regression model is given by the following function.

$$CQ = 17.878 - 1.325 \cdot Gu + .009 \cdot LoCpC - .008 \cdot NoC$$

In this equation, Gu is encoded as 1 for having a single branch, 2 for having 2 or 3 branches, and 3 for having 4 or more branches. One addition in LoCpC translates to 1000 line changes in the source code. NoC represents the total number of project contributors. The model explained 1% of its variance ($R^2 = .010$), resulting in a very weak effect size [50] and low predictive power [51]. The overall regression was significant ($F(3, 5614) = 19.806$, $p < .001$). The hypotheses were tested with a significance level of $\alpha = .05$. It was found that bot Gu and LoCpC significantly predicted CQ. The found beta coefficients and results of hypothesis testing are presented in Figure 3. The complete output of the linear regression model can be found in Figure A1.

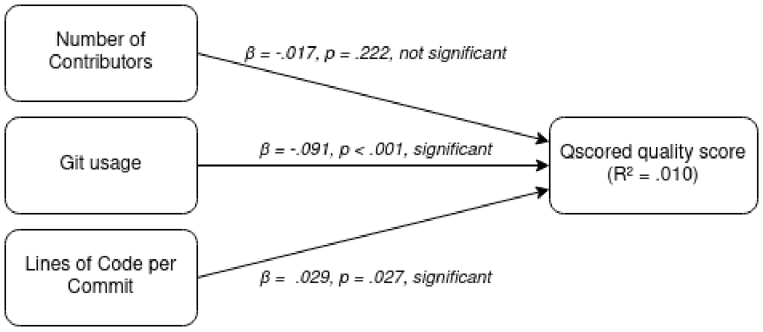

Number of Contributors
β = -.017, p = .222, not significant
Git usage
β = -.091, p < .001, significant
Qscored quality score (R² = .010)
Lines of Code per Commit
β = .029, p = .027, significant

Figure 3: The linear regression model results

5 Interpretation

The research conducted analyzes the collected dataset. This section describes the meaning of the results and possible issues regarding the validity of the presented conclusion. Beta coefficients in the model represent the change in CQ when one of the independent variables changes by one unit. The significant relationships suggest that Git usage is related to code quality, using more branches increases code quality. According to the model, lines of code per commit do contribute to code quality. Synthesizing this result with that of Santos and Hindle [40], a reason could be that smaller commits make commit messages more meaningful and therefore improve collaboration quality. Common sense suggests that better collaboration results in a better end product, a piece of software in our case. Furthermore, the low effect size implies that there are probably many other variables (not measured here) influencing code quality.

5.1 Limitations

The primary component that influences the experiment's validity is related to the used and collected data. The concept of team size is represented by the NoC of a GitHub repository. There is no threshold set for this variable on what extent each GitHub user had to contribute to the repository to be counted, which means that users who contributed just a few lines of code are counted the same as a full-time developer responsible for maintaining the repository. Additionally, there was no condition set about which files had to be modified to be counted as a contributor. Our research makes conclusions about the number of contributors influencing code quality, but some might have only contributed to documentation or contributed a long time ago. Lastly, a subgroup of GitHub repositories analyzed had a contributor that was an automated continuous integration bot. These bots are also counted as a collaborator. Second, the variable git usage (GU) is represented by the number of branches used in the repository. A concern with this variable is that this value represents the number of branches during the time of scraping the data. We thus do not have access to the maximum number of concurrent branches used during development. This might incorrectly represent repositories who have merged their branches and removed them afterwards. Third, LoCpC can likewise be incorrectly influenced by automated CI bots. Additionally, changes to documentation can have an unwanted influence on this averaged variable. Fourth, our analysis of software development factors implementing code quality is limited to the three mentioned independent variables. In accordance with extant literature, we expected to see an influence of time-to-market on the code quality. However, TTM cannot be measured for the open-source projects in our dataset. We still see the possibility of this (and possible other) variable(s) influencing the code quality since our multiple regression model has a very weak effect size ($R^2 = .010$). Lastly, the QScored dataset also poses some threats to validity. The QScored score is, though scientifically grounden, an imperfect representation of code quality. QScored does not provide a precise timestamp about when the code quality has been recorded in their dataset. QScored mentions their data is updated monthly, implying our enrichment of their dataset can be off by one month at maximum (such that the score could be different on the date that our tool was set to download (e.g. NoC).

5.2 Generalizability

Even though the threats to validity mentioned above, we argue that these findings can be generalized to other actively maintained open-source projects. This is supported by our large sample size of $N = 5614$. The main difference between our chosen sample size and the general population of active open-source projects is the programming-language. We argue that it is not probable that a relation exists between programming language and our independent variables. Also, existing literature describes no significant impact between programming languages and code quality [52].

5.3 Reflection

One of the main challenges in this study has been the availability and accessibility of the QScored dataset. The dataset could be accessed through an HTTP API, but posed two major issues. Responses were relatively slow, some taking longer than two seconds. Also, responses from the QScored API do not contain the repository link described in QScored's documentation. This resulted in our data collection tool having to run for several days to gather all data. Furthermore, our stated minimum requirements for active repositories caused over 90% of the repositories to be excluded from our data. Both issues made it a challenge to scale data collection to over 35 thousand records from the QScored dataset. To mitigate this in the future, more preliminary work could be put in to investigate the fitness of the dataset. Hardware or software could then be chosen and configured accordingly (e.g., distributing the requests over several computers or setting up a cloud computing solution that can run automatically full-time). Nonetheless, alongside this research paper we provide a tool and our generated dataset (Appendix B), meaning that future work can circumvent these issues by downloading the set directly.

6 Conclusion

The main research question of this paper was as follows. *What easily manipulable factors related to software development influence code quality?* To answer this, we have conducted a data analysis experiment to investigate the influence of factors relating to the software development process on code quality. First, a literature review was conducted to find related variables that could affect QC. In this phase three independent variables were chosen to be analyzed for the next experiment. An API consumer was built to query additional publicly available data from the GitHub repositories in the QScored dataset. This retrieved data represents the number of contributors of a project, git usage and lines of code per commit. Lastly, the data was analyzed as a multiple regression problem to test our hypotheses. The data indicates that there is no significant evidence for number of contributors as a predictor of code quality. Git usage and lines of code per commit, however, were found to be significant predictors of code quality.

In line with other work [20], [21] using more branches does increase the CQ score, which indicates lower quality. It could also be argued that our work conforms to earlier work with respect to team size as a predictor for code quality. Namely, Brooks Jr [15] suggests the task at hand, communication within a team and division of tasks and contributors' qualities should be guidelines for the size of a team. These variables were not taken into account in our work due to feasibility issues, meaning our measured variable NoC is not a reliable predictor for code quality (hence the insignificant coefficient).

Our work provides practical directions for further investigating code quality predictors. As mentioned in the limitations, this work currently sees all contributors as equal, regardless of their contributions and does not filter out bots. Future work could focus on building more advanced scrapers that can better specify information about repositories. This could lead to more insightful variables. Our conclusion could be addressed in further studies by looking at further optima of the discussed variables. Additionally, future studies could implement improved operationalizations of the variables used here to find more definite support for the relationships hypothesized in this work. Lastly, future work could investigate whether there are more (hidden) variables that might affect code quality.

7 Acknowledgements

We would like to give acknowledgement to the collaborators of all open-source projects that were studied. Via the GitHub API [43] they were accessible for our purposes. Credit is also due to QScored [28] for providing a dataset with projects scored on their code quality. They made their work available through an API [53], which gave our research viability.

References

[1] G. S. Matharu, A. Mishra, H. Singh, and P. Upadhyay, "Empirical study of agile software development methodologies: A comparative analysis," *ACM SIGSOFT Software Engineering Notes*, vol. 40, no. 1, pp. 1–6, 2015.

[2] R. Baggen, J. P. Correia, K. Schill, and J. Visser, "Standardized code quality benchmarking for improving software maintainability," *Software Quality Journal*, vol. 20, no. 2, pp. 287–307, 2012.

[3] E. Von Hippel, "Learning from open-source software," *MIT Sloan management review*, vol. 42, no. 4, pp. 82–86, 2001.

[4] Kubernetes, *Kubernetes*, `https://github.com/kubernetes/kubernetes`, 2021.

[5] Y. Perez-Riverol, L. Gatto, R. Wang, *et al.*, *Ten simple rules for taking advantage of git and github*, 2016.

[6] J. Loeliger and M. McCullough, *Version Control with Git: Powerful tools and techniques for collaborative software development.* " O'Reilly Media, Inc.", 2012.

[7] Krasner and CISQ, "The cost of poor quality software in the us: A 2020 report," Jan. 2020. [Online]. Available: `https://www.it-cisq.org/pdf/CPSQ-2020-report.pdf`.

[8] Y. . Guo, R. O. Spínola, and C. . Seaman, "Exploring the costs of technical debt management – a case study," *Empirical Software Engineering*, vol. 21, no. 1, pp. 159–182, 2014. DOI: `10.1007/s10664-014-9351-7`.

[9] J. . Bohnet and J. . Döllner, "Monitoring code quality and development activity by software maps," *Proceedings of the 2nd Workshop on Managing Technical Debt*, pp. 9–16, 2011. DOI: `10.1145/1985362.1985365`.

[10] D. I. Sjoberg, A. . Yamashita, B. C. Anda, A. . Mockus, and T. . Dyba, "Quantifying the effect of code smells on maintenance effort," *IEEE Transactions on Software Engineering*, vol. 39, no. 8, pp. 1144–1156, 2013. DOI: `10.1109/tse.2012.89`.

[11] A. . Yamashita and L. . Moonen, "Do code smells reflect important maintainability aspects?" *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, pp. 306–315, 2012. DOI: `10.1109/icsm.2012.6405287`.

[12] ——, "Exploring the impact of inter-smell relations on software maintainability: An empirical study," *2013 35th International Conference on Software Engineering (ICSE)*, pp. 682–691, 2013. DOI: `10.1109/icse.2013.6606614`.

[13] L. Fried, "Team size and productivity in systems development bigger does not always mean better," *Information System Management*, vol. 8, no. 3, pp. 27–35, 1991.

[14] A. Mundra, S. Misra, and C. A. Dhawale, "Practical scrum-scrum team: Way to produce successful and quality software," in *2013 13th International Conference on Computational Science and Its Applications*, IEEE, 2013, pp. 119–123.

[15] F. P. Brooks Jr, "The mythical," *ManMonth: Essays on Software Engineering,, Anniversary Edition,: Addison-Wesley*, 1995.

[16] N. N. Zolkifli, A. Ngah, and A. Deraman, "Version control system: A review," *Procedia Computer Science*, vol. 135, pp. 408–415, 2018.

[17] C. Brindescu, M. Codoban, S. Shmarkatiuk, and D. Dig, "How do centralized and distributed version control systems impact software changes?" In *Proceedings of the 36th International Conference on Software Engineering*, 2014, pp. 322–333.

[18] D. Spinellis, "Git," *IEEE software*, vol. 29, no. 3, pp. 100–101, 2012.

[19] C. Walrad and D. Strom, "The importance of branching models in scm," *Computer*, vol. 35, no. 9, pp. 31–38, 2002.

[20] R. Premraj, A. Tang, N. Linssen, H. Geraats, and H. van Vliet, "To branch or not to branch?" In *Proceedings of the 2011 International Conference on Software and Systems Process*, 2011, pp. 81–90.

[21] E. Shihab, C. Bird, and T. Zimmermann, "The effect of branching strategies on software quality," in *Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement*, 2012, pp. 301–310.

[22] Y. Lu, X. Mao, T. Wang, G. Yin, Z. Li, and H. Wang, "Continuous inspection in the classroom: Improving students' programming quality with social coding methods," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, 2018, pp. 141–142.

[23] S. Krusche, M. Berisha, and B. Bruegge, "Teaching code review management using branch based workflows," in *Proceedings of the 38th International Conference on Software Engineering Companion*, 2016, pp. 384–393.

[24] M. Hadj-Kacem and N. Bouassida, "A multi-label classification approach for detecting test smells over java projects," *Journal of King Saud University - Computer and Information Sciences*, 2021, ISSN: 1319-1578. DOI: `https://doi.org/10.1016/j.jksuci.2021.10.008`. [Online]. Available: `https://www.sciencedirect.com/science/article/pii/S1319157821002950`.

[25] G. Suryanarayana, G. Samarthyam, and T. Sharma, *Refactoring for Software Design Smells: Managing Technical Debt*, 1st ed. Morgan Kaufmann, 2014.

[26] M. Tufano, F. Palomba, G. Bavota, *et al.*, "When and why your code starts to smell bad," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1, 2015, pp. 403–414. DOI: `10.1109/ICSE.2015.59`.

[27] M. Fowler, K. Beck, J. Brant, W. Opdyke, D. Roberts, and E. Gamma, *Refactoring: Improving the Design of Existing Code*, 1st. Addison-Wesley Professional, 1999.

[28] T. Sharma and M. Kessentini, "Qscored: A large dataset of code smells and quality metrics," in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, 2021, pp. 590–594. DOI: `10.1109/MSR52588.2021.00080`.

[29] T. Sharma and D. Spinellis, "A survey on software smells," *Journal of Systems and Software*, vol. 138, pp. 158–173, 2018, ISSN: 0164-1212. DOI: `https://doi.org/10.1016/j.jss.2017.12.034`. [Online]. Available: `https://www.sciencedirect.com/science/article/pii/S0164121217303114`.

[30] A. J. Riel, "Object-oriented design heuristics, vol. 335," *Reading: Addison-Wesley*, 1996.

[31] W. H. Brown, R. C. Malveau, H. W. S. McCormick, and T. J. Mowbray, *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*, 1st ed. John Wiley & Sons, Inc., 1998.

[32] S. Vidal, H. Vazquez, J. A. Diaz-Pace, C. Marcos, A. Garcia, and W. Oizumi, "Jspirit: A flexible tool for the analysis of code smells," in *2015 34th International Conference of the Chilean Computer Science Society (SCCC)*, 2015, pp. 1–6. DOI: `10.1109/SCCC.2015.7416572`.

[33] N. Tsantalis, D. Mazinanian, M. Fokaefs, *et al.*, *JDeodorant*, version 5.0.81, 2020. [Online]. Available: `https://github.com/tsantalis/JDeodorant`.

[34] T. Sharma, *Designite - A Software Design Quality Assessment Tool*, version 2.5.10, http://www.designite-tools.com, May 2016. DOI: `10.5281/zenodo.2566832`. [Online]. Available: `https://doi.org/10.5281/zenodo.2566832`.

[35] T. Paiva, A. Damasceno, E. Figueiredo, and C. Sant'Anna, "On the evaluation of code smells and detection tools," *Journal of Software Engineering Research and Development*, vol. 5, no. 1, p. 7, Oct. 2017, ISSN: 2195-1721. DOI: `10.1186/s40411-017-0041-1`. [Online]. Available: `https://doi.org/10.1186/s40411-017-0041-1`.

[36] J. Pantiuchina, M. Lanza, and G. Bavota, "Improving code: The (mis) perception of quality metrics," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2018, pp. 80–91. DOI: `10.1109/ICSME.2018.00017`.

[37] G. Bavota and B. Russo, "Four eyes are better than two: On the impact of code reviews on software quality," in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, IEEE, 2015, pp. 81–90.

[38] P. S. Kochhar, D. Wijedasa, and D. Lo, "A large scale study of multiple programming languages and code quality," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, IEEE, vol. 1, 2016, pp. 563–573.

[39] S. Butler, M. Wermelinger, Y. Yu, and H. Sharp, "Relating identifier naming flaws and code quality: An empirical study," in *2009 16th Working Conference on Reverse Engineering*, IEEE, 2009, pp. 31–35.

[40] E. A. Santos and A. Hindle, "Judging a commit by its cover," in *Proceedings of the 13th International Workshop on Mining Software Repositories-MSR*, vol. 16, 2016, pp. 504–507.

[41] V. Thakur, M. Kessentini, and T. Sharma, "Qscored: An open platform for code quality ranking and visualization," in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, IEEE, 2020, pp. 818–821.

[42] Github, *Managing releases in a repository*, `https://docs.github.com/en/repositories/releasing-projects-on-github/managing-releases-in-a-repository`.

[43] ——, *Rest api*, `https://docs.github.com/en/rest`.

[44] W. McKinney *et al.*, "Pandas: A foundational python library for data analysis and statistics," *Python for High Performance and Scientific Computing*, vol. 14, no. 9, pp. 1–9, 2011.

[45] P. Mishra, C. M. Pandey, U. Singh, A. Gupta, C. Sahu, and A. Keshri, "Descriptive statistics and normality tests for statistical data," *Annals of cardiac anaesthesia*, vol. 22, no. 1, p. 67, 2019.

[46] P. Good, "Robustness of pearson correlation," *Interstat*, vol. 15, no. 5, pp. 1–6, 2009.

[47] V. P. Bhapkar, "11 anova and manova: Models for categorical data," *Handbook of statistics*, vol. 1, pp. 343–387, 1980.

[48] D. W. Hosmer and S. Lemesbow, "Goodness of fit tests for the multiple logistic regression model," *Communications in statistics-Theory and Methods*, vol. 9, no. 10, pp. 1043–1069, 1980.

[49] L. S. Aiken, S. G. West, and S. C. Pitts, "Multiple linear regression," *Handbook of psychology*, pp. 481–507, 2003.

[50] D. Moore, W. Notz, and M. Flinger, *The basic practice of statistics*, 6th ed. WH Freeman New York, 2013.

[51] W. Chin, "The partial least squares approach for structural equation modeling," in *Modern Methods for Business Research.* Abingdon, United Kingdom: Lawrence Erlbaum Associates Publishers, 1998, pp. 295–336.

[52] E. D. Berger, C. Hollenbeck, P. Maj, O. Vitek, and J. Vitek, "On the impact of programming languages on code quality: A reproduction study," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 41, no. 4, pp. 1–24, 2019.

[53] QScored, *QScored Docs - APIs.* [Online]. Available: `https://qscored.com/docs/api/`.

Appendix A Enriched dataset metadata

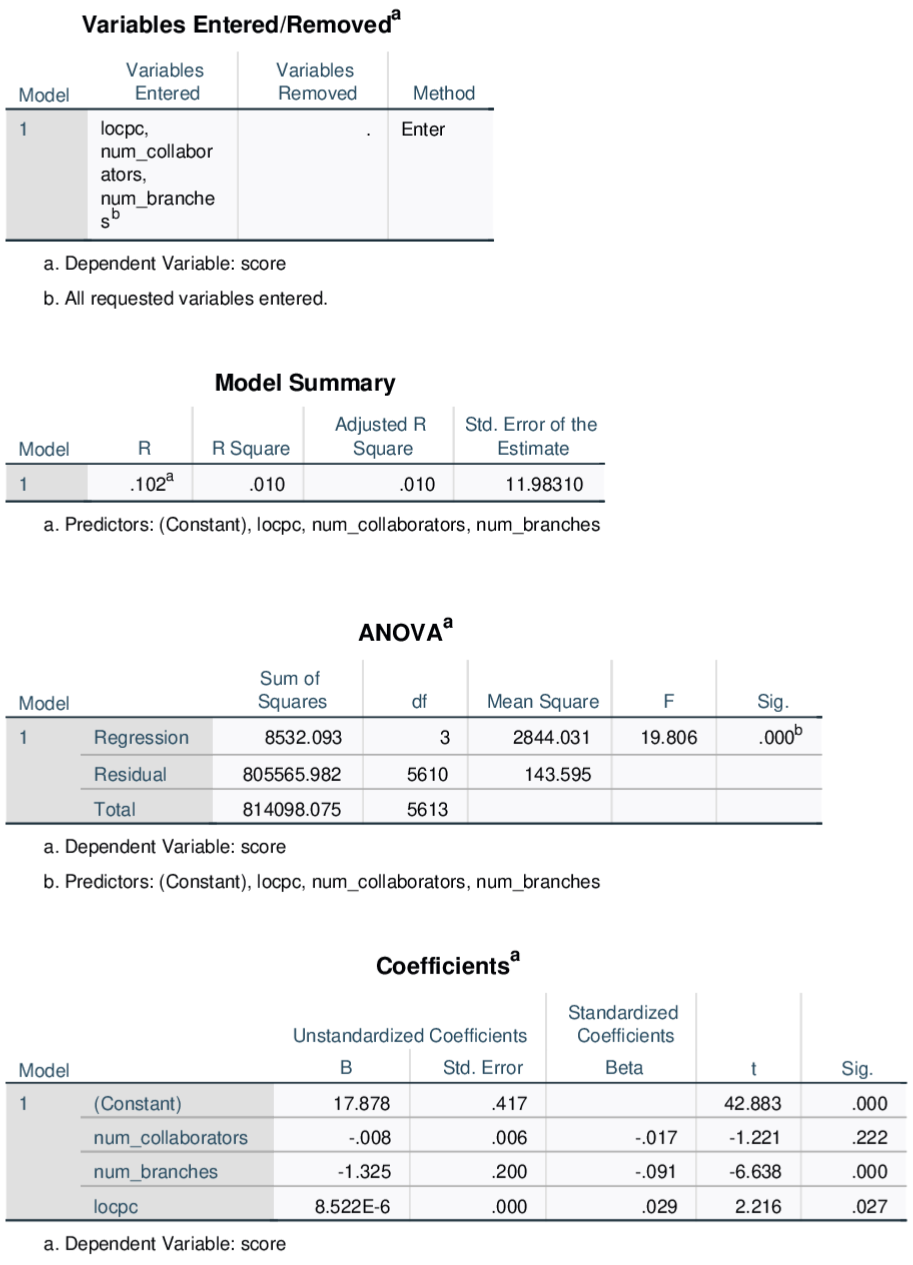
Table A1: Fields captured in enriched dataset

Field name	Description	Example value
ProjectName	Name of the project as specified on Github	HtmlBuilder
RepoLink	HTTPS link to the project's repository on Github	https://github.com/maroontress/HtmlBuilder
Score	QScored code quality score	0.28
Lang	Project language, either cs (C#) or java	cs
NumCollaborators	The number of GitHub users who have contributed to the project	17
NumBranches	The encoded number of concurrent branches the project has	2
MeanLocPCA	Average number of additions per commit	1949.74
MeanLocPCD	Average number of deletions per commit	-37.55
STDLocPCA	Standard deviation for the number of additions per commit	5908.81
STDLocPCD	Standard deviation for the number of additions per commit	65.78

Appendix B Code and Data used for Analysis

The repository containing the dataset and the data collection tool that was coded and used to retrieve the data from the various sources can be found at: `https://github.com/TKForgeron/INFOARM`.

Appendix C Statistical Test Output



Variables Entered/Removed[a]

Model	Variables Entered	Variables Removed	Method
1	locpc, num_collaborators, num_branches[b]	.	Enter

a. Dependent Variable: score

b. All requested variables entered.

Model Summary

Model	R	R Square	Adjusted R Square	Std. Error of the Estimate
1	.102[a]	.010	.010	11.98310

a. Predictors: (Constant), locpc, num_collaborators, num_branches

ANOVA[a]

Model		Sum of Squares	df	Mean Square	F	Sig.
1	Regression	8532.093	3	2844.031	19.806	.000[b]
	Residual	805565.982	5610	143.595		
	Total	814098.075	5613			

a. Dependent Variable: score

b. Predictors: (Constant), locpc, num_collaborators, num_branches

Coefficients[a]

Model		Unstandardized Coefficients		Standardized Coefficients	t	Sig.
		B	Std. Error	Beta		
1	(Constant)	17.878	.417		42.883	.000
	num_collaborators	-.008	.006	-.017	-1.221	.222
	num_branches	-1.325	.200	-.091	-6.638	.000
	locpc	8.522E-6	.000	.029	2.216	.027

a. Dependent Variable: score

Figure A1: Output of Linear Regression Model

Appendix D Author Contributions

Table A2: Project dedication of each team member, represented in hours spend and percentage of participation

	Tasks	**Hours spent**	**Participation (%)**
Matthijs Blaauw	Sections Abstract, 2.2, 5, 6, Coupling Python client to QScored API, running data collection tool	82	25%
Max de Froe	Sections Abstract, 2.4, 5, Coupling Python client to QScored API	79	25%
Kouros Pechlivanidis	Sections 1, 2.1, 3, 4, 5, 6, Building Python client	83	25%
Tim Smit	Sections 1, 2.3, 5, 6, Building Python client, Final spell check and layout corrections	80	25%