It's Never too Early to Learn About Code Quality: A Longitudinal Study of Code Quality in First-year Computer Science Students

Linus Östlund
KTH Royal Institute of Technology
Stockholm, Sweden
linusost@kth.se

Niklas Wicklund
KTH Royal Institute of Technology
Stockholm, Sweden
niwi@kth.se

Richard Glassey
KTH Royal Institute of Technology
Stockholm, Sweden
glassey@kth.se

ABSTRACT

Low code quality incurs a significant cost upon the software industry. Despite this, little serious effort has been devoted to the topic at the most basic levels of computing science education. Where studies have been conducted, the results are disappointing in terms of code quality. In this work, four iterations of a medium to large CS1 and CS2 course (n = 200 students) were analyzed through the lens of code violations to better understand (1) what code violations occur most frequently, (2) how does their occurrence vary throughout the course, and (3) to what extent do teaching assistants have an effect upon code quality. Results showed a statistically significant improvement in code quality over 19 assignments in all four course iterations. In particular, when a single violation was the focus of a learning outcome, the effects were both a dramatic fall and sustained low occurrence. However, this improvement was mostly focused on the three most frequently occurring violations (> 70%), which masked an increase in lesser occurring violations throughout the course. Finally, the effects from different teaching assistants were found to be random at best, contradicting expectations that their influence would be easily detected. Given that explicit learning outcomes targeted at code quality had an effect and teaching assistant influence had no effect, a path forward to improving code quality might combine these forces without creating too many additional demands upon the already stretched teaching resources with CS1 and CS2 courses.

CCS CONCEPTS

• **Social and professional topics → CS1**.

KEYWORDS

CS1, Programming, Code Violations, Teaching Assistants

ACM Reference Format:
Linus Östlund, Niklas Wicklund, and Richard Glassey. 2023. It's Never too Early to Learn About Code Quality: A Longitudinal Study of Code Quality in First-year Computer Science Students. In *Proceedings of the 54th ACM Technical Symposium on Computing Science Education V. 1 (SIGCSE 2023), March 15–18, 2023, Toronto, ON, Canada.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3545945.3569829

SIGCSE 2023, March 15–18, 2023, Toronto, ON, Canada

ACM ISBN 978-1-4503-9431-4/23/03.
https://doi.org/10.1145/3545945.3569829

1 INTRODUCTION

It is generally acknowledged that low-quality code harms software. Errors, maintenance, and refactoring (usually referred to as the *technical debt*) of poor quality code in the US alone was estimated to be $2.87 trillion in 2018 [11]. Truong et al. found that more than 50% of a software project's budget is spent on improving code quality [33]. Although this is a well-known and widespread problem, a study by Kirk et al. from 2020 found that only 30% out of 141 CS1 courses focus on code quality [17].

Programming is a complex intellectual activity, and designing a CS1 course is a task associated with many challenges (scale, resources, staffing, variation in student ability and so on). As one comprehensive literature study puts it, "*Novice programmers suffer from a wide range of difficulties and deficits*" [23], an observation further supported by Toll et al., who emphasize that important code quality aspects like maintainability and readability end up in the shadow of students struggling with basic syntax and programming concepts [3]. Beginner students are more concerned with producing a solution to pass the assignment requirements, rather than focusing on good-quality code.

The absence of code quality as an articulate learning goal in education is concerning. There should be a combined effort of all software educators to curb the rampant cost of technical debt. Reports were found that suggest good code quality be considered a key competency amongst engineer prospects, both by practitioners and educators [3, 17, 37]. McGill et al. have research showing a strong relationship between the quality of students' algorithms and the quality of their final programs [25], findings that suggest best code practices should be taught early in introductory courses.

With little spare time or resources, this situation is entirely understandable and to be expected. Beyond learning goals, another source of influence, positive or negative, upon a student's awareness of code quality could be their teaching assistant (TA). TAs are commonly employed to handle the scale of CS1 courses [23], but also raise concerns about levels of quality in education, as they lack the training and experience [26] and are often overloaded and underappreciated [29]. Due to their close relationship to their students, there is the possibility that they impart their own awareness and attitude about code quality (or not) upon their students.

In response to these challenges and uncertainties, this work focuses upon four years of assignment data collected in a CS1 and CS2 course with an enrollment of roughly 200 students. The research aim is to use code violations as a lens upon this dataset to reveal the prevalence and progression of code quality throughout the course, whilst also determining the extent of influence that different teaching assistants might have upon their respective sub-groups

of students. The following research questions will be addressed in the remainder of this work:

- **RQ1**: Is there a noticeable improvement in code quality during a course offering?
- **RQ2**: Is there an observable difference in code quality between the different TA groups?

2 BACKGROUND

2.1 Code quality metrics

Understanding good quality code is a well-researched, yet elusive concept. Frameworks have been developed to identify software quality, e.g., the ISO/IEC 9126, which aims to address human biases that affect software development [13]. Breuker et al. highlight that not all characteristics are immediately meaningful in an educational context [4]. Related research has approached measuring code quality in various ways, but the most common denominators are *readability* and *understandability* as the foremost criteria [3, 16, 17, 19]. However, there is no widely accepted approach, and other work has focused instead on code complexity [12].

In the educational context, Brown et al. found no consensus on educators' perception of which student mistakes are most common and most serious [5]. Usually, this relies on some kind of human expertise to decide upon, thus introducing an element of subjectivity [3–5]. This is also reported by Roy et al., who point out that the existing metrics are questionable, as there is an apparent discrepancy between what state-of-the-art tools can measure and what developers deem as good code quality [10]. The disclosure of such inconsistencies needs to be preceded by careful consideration and discussion amongst and between both communities.

In this work, the language of instruction under study is Java. Since Java was introduced in 1995, many coding conventions have been developed, and collated [21, 34, 38], and Oracle's official style guide remains an important and influential reference [28]. An anecdotal work often cited in programming literature is *Clean Code* by Robert C. Martin, which includes concepts like best commenting practices and ideal method lengths [24].

Whilst diverting from these guidelines still produces code that compiles, it is considered to lower the overall quality that ultimately leads towards technical debt [1, 18] - a term coined to describe the often hidden costs incurred when software developers take shortcuts, skip best practices, avoid the intricate (architecture) or dull problems (readable code), for some other pressing factor (budget constraints, tight deadlines, lack of skills).

2.2 Student code quality

A common angle of approach when measuring code quality is to utilize tools that analyze code without actually executing it, so-called Static Code Analysis (SCA) [22, 27]. Such tools are commonly used in development and production to spot coding flaws and keep coding styles consistent, in order to minimize the technical debt [37]. These analyzers check code against a set of rules and produce a comprehensive report of violations. The number of code violations is then used as a proxy for code quality.

SCA is intended primarily for experienced programmers, so a majority of the code violations are too advanced for beginners to comprehend and correct. Related work in computer science education has approached this issue with SCA tools by using experts to select the quality violations that will be most explainable and actionable for students [4, 8, 15, 33] as well as investigating the effects that SCA usage might have upon student code quality [15].

Keuning et al. conducted a code quality analysis on the Blackbox dataset, which contains over 2.6 million snapshots from users of the popular BlueJ educational IDE [15]. They found that there were substantial code quality issues and that students did not address them even when provided with tool support and a curated list of violations relevant to their level of ability.

Breuker et al. investigated code quality improvement between first and second years students with 20 code properties that are represented by 20 measurable rules [4]. A sample of these rules include number of classes in a package, number of lines of code in a class and, the number of hard-coded constants in expressions (so called *Magic Numbers*). They use various metric tools like PMD and ANTLR and conclude that "*measuring static quality of code written by student teams is a hard job producing interesting results*".

Troung et al. developed a framework that uses both software engineering metrics and relative comparison to judge the quality of students' programs [33]. They claim their framework is especially useful for "fill in the gap" style exercises. The framework was designed by carrying out a literature review that aimed to find common Java errors students made.

Effenberger and Pelánek analyzed over 100,000 solutions to 161 short coding problems in Python and derived a catalog of 32 defects [8]. The aim of the catalog is for educators to use it as "*an inspiration for what to discuss and highlight in lectures*". In particular, they make the observation that correct solutions often contain defects, yet these go unnoticed as an opportunity for feedback.

2.3 Teaching assistants

One common resource in universities worldwide is Teacher Assistants (TAs). A literature study from 2018 by Luxton-Reilly et al. conducted a systematic review to survey introductory programming literature, categorizing TAs somewhat appropriately as *university infrastructure*, underlying their pervasiveness and importance. They concluded, "*[this] topic has too few papers to warrant a plot of their distribution over time*" [23] and encourages "*further research that considers the interplay of different infrastructures*".

In the same report, the impact of TAs in CS1 courses was found to be *extremely beneficial*. Citing two independent reports from different institutions [7, 35], they concluded that TAs are a great support when it comes to students' learning, engagement, and identity building[23]. Whilst other reports agree on the benefits; employing TAs benefits the institution as it is relatively cheap, flexible, and part-time labor; it is usually a (possible) trade-off at the expense of maintaining quality in teaching [26]. A lack of adequate subject knowledge and pedagogical skills might affect grading and learning outcome [36]. As an example, the study from the University of Helsinki reports they choose TA depending on grades, attitude, and "*with the hope that they are good at teaching as well*" [35]. In perhaps the most challenging case study, TAs in [29] were found to be mostly overloaded, underappreciated, and underpaid.

A New Zealand case study found most TAs understand that students develop their skills when receiving formative feedback, but

their primary concern was to justify the grades that they awarded rather than producing such feedback [20]. According to Price et al., assessment is probably the single biggest influence on how students approach their learning [31]. TAs face many challenges; inexperience and their own studies to name a few, which might take their toll on the students' learning experience.

Whilst there is literature focused on the use of SCA in automatic assessment of students code submissions in order to give formative feedback on code quality [9, 14, 16, 33], few mentions on TA's impact on students overall code quality. Given the noted problems in students' code quality and the near-ubiquitous presence of TAs' and their influence upon their students, both topics merit further study in isolation and in combination.

3 METHOD

3.1 Context of study

This work focuses upon the Introduction to Computer Science (INDA) course at KTH Royal Institute of Technology in Stockholm, Sweden. At the start of each course offering, there are approximately 200 enrolled students. The course focuses first on imperative and object-oriented programming skills with Java as the language of instruction (following typical CS1 topics), before transitioning into an introduction to algorithms, with topics such as implementing common data structures and analysis of complexity and correctness (following typical CS2 topics). The course runs for 21 weeks during academic terms from September through to February.

All of the course assessment is formative, based on weekly programming tasks. Tasks 1-9 closely follow the path laid out by *Objects First with Java: A Practical Introduction Using BlueJ* [2]. These are intended to teach the students basic syntax and programming concepts. Tasks 10-19 build upon these skills with students learning to use recursion, analysis of complexity and implement common algorithms and datastructures. The final task is worth two course points and includes a report on the performance of Quicksort, its strengths and weaknesses and its various improvements.

Depending on course iteration, the enrolled students are partitioned amongst 12-14 TAs. Each TA has a group of about 15 students and is responsible for grading each student submission. Teaching assistants are typically second and third year students who have already completed the course. In line with [35], the TAs are not expected to have pedagogical experience, but are selected based upon their enthusiasm and endorsements from current TAs. However, in recent iterations, all TAs have taken mandatory training sessions during the initial weeks of the semester, which provides a partial solution for awareness of pedagogical concerns, but still can be no substitute for practical experience and reflection.

The students are expected to solve the weekly task and present their solution in the scheduled two-hour seminar with the rest of their group. The presentation usually occupies the first hour of the seminar, where the TA plans the remaining time. The material has little to no focus on teaching good code quality, so the remedy to poor programming practices relies on the TAs capturing and communicating them to each student. Assessing 15 student assignments every week is cumbersome and time-consuming, so finding and communicating all possible code errors is not a realistic nor consistent expectation for all the TAs.

Table 1: The number of valid repositories that constitute the dataset.

Year	Total Repos	Submitted Repos	Valid Repos	Teaching Assistants	Students
2018	2790	2288	2133 (76%)	11	151
2019	2412	2044	1941 (80%)	9	134
2020	2270	1963	1905 (84%)	9	123
2021	1986	1668	1620 (82%)	8	110

Since the course has no mandatory prerequisites, there is a wide variance in coding experience among the students. During their first week, students participate in a *Sorting Hat*[1] exercise to determine which level of group they will join. There are four options: students may choose to join a beginner, regular, plus or plus-plus group. There is no pre-test on aptitude, so the partitioning is purely subjective self-assessment. The plus- and plus-plus-groups are for those who consider themselves experienced and involve modified assignments and deeper discussions on the topics. The material and submissions are the same throughout the remaining two groups and the teaching team does not distinguish between beginner and regular groups, as most often this is more a reflection of self-confidence and humility than raw skill.

Since 2015, the course migrated to using version control and GitHub Enterprise for all student code submissions. Whilst this was deemed a high-risk decision at the time, given the well-known technical challenge that Git/GitHub use can be, the years of positive experience and few concerns have justified the decision. Each task is modeled as a Git repository (repo) and each student receives a clone of the task that is private to them and the teaching team. Initially, the task repos contain a lot of scaffolding and code templates. As the course progresses, the character of the exercises changes. Students receive diminishing code templates and exercise 18 and 19 consists of plain method headers. This approach has led to an extensive dataset of approximately 20,000 repositories that capture the student commits, contributions, corrections, communication with TAs and so on.

3.2 Data preparation and processing

The data set studied consists of Git/GitHub repositories containing student code submissions from 2018 to 2021. All code files modified by the students' were included whether the student had passed or failed an assignment. A summary of the dataset is shown in Table 1.

3.2.1 Exclusions. The plus and plus-plus groups were excluded as they involve students who consider themselves experienced, and those weekly assignments, to some extent, deviate from the regular learning goals. The course introduces the concept of unit testing, and from task-9 and beyond, each template includes a test suite. These test suites have been excluded. There is also a surge of students from various programs who join midway for algorithms and datastructures. As this study aims to find progress over the entire course, these students are also excluded. Finally, Task 1 is excluded since it does not involve any programming exercises.

[1] See: Harry Potter and the Philosopher's Stone

3.2.2 Identification of student code. The course uses a separate GitHub Organization each year to deploy all weekly assignments. Each assignment has a corresponding student repository *student-task-xx*, which contains a *README.md* with instructions and a *src-folder* with potential code templates. Not all weeks include templates, and not all code templates are intended for students to change. In order to target code modified by the student and reduce noise from irrelevant code, the git function `git blame` was used. This approach enabled us to focus on student code and exclude all code templates provided by the teaching team.

3.2.3 SCA tools and rule selection. Two well-known SCA tools, PMD [30] and SonarSource [32], were selected to analyze student code and use the number of reported violations as a proxy for code quality. At the time of writing, PMD has 326 available Java rules, and SonarSource has 627 rules. All rules were examined and categorized as either relevant or not in the educational context and cross-referenced with the work of [33] and [4]. As the tools have some overlap, PMD was the primary choice (33 rules) and the remainder were complemented by SonarSource (12 rules) [2]. To ensure the selected rules cohered with the course objectives they were then empirically assessed by running an initial test on the course iteration of 2021. Rules with zero violations were also excluded. The student repositories for each course iteration were cloned locally and analyzed with PMD and SonarSource corresponding to the final rule set. The recorded data was then gathered in a CSV file. Repositories containing non-compilable files were excluded since PMD could not analyze such files.

3.2.4 Normalisation. The 18 tasks spanned various learning objectives, and each task had its own characteristics. Thus, there is an expected variance in code templates, lines of codes for student solutions, and concentration of violations depending on task. Instead of measuring an absolute value of violations, the metric used in this study is `V/KLOC` (Violations per thousand Lines Of Code). In combination with the use of `git blame` in the data processing, this normalization is a fair representation of how error-prone the code submitted by the student is. This procedure is found in the related work discussed in section 2 [15] and is intended to be a more accurate metric, especially for task transcending comparisons.

3.3 Statistical tests

The data were analyzed using the statistical tests presented in this section. A low value of `V/KLOC` is interpreted as better code quality.

3.3.1 RQ1: Is there a noticeable improvement in code quality during a course offering? As the first research question regards any possible improvement in the violations measured by the study, the normalized statistics `V/KLOC` will be used in a linear regression to test if any significance exists between it and the independent variable `Task`. The outcome is dependent on the p-value of the linear regression, where $p < 0.05$ is an indication that the model describes a significant relationship between `Task` and `V/KLOC`.

3.3.2 RQ2: Is there an observable difference in code quality amongst the different TA groups? A common approach when analyzing differences between groups is studying the behavior of their distribution, mean and median. The Kruskal-Wallis test, a non-parametric test that does not require normal distribution, was used to compare the TA groups. A rejected null hypothesis ($p < 0.05$) indicates that the groups' medians are unequal, and suggest some of those students underperform compared to their peers. Conover's test was used to identify which groups had a significant difference [6].

[2]Table of complete rule set: https://github.com/NiklasWicklund/never-too-early

4 RESULTS

The frequency of all violations found in student submissions from 2018 to 2021 is presented in Figure 1. The first part of the figure shows the ranking of most common violations found in the dataset. Most notably, three rules (*MagicNumber*, *SystemPrintln* and *CommentRequired*) account for over 70% of all reported violations. The second part of the figure shows a heatmap of where violations are occurring most frequently across the series of tasks.

Figure 2 shows a regression analysis of the sum of each student's `V/KLOC`. A negative trend with a p-value < 0.05 was found for each course iteration, implying that code quality improves as the course progresses. However, when the three most commonly occurring violations are omitted the trend was reversed and code quality was found to be gradually becoming worse.

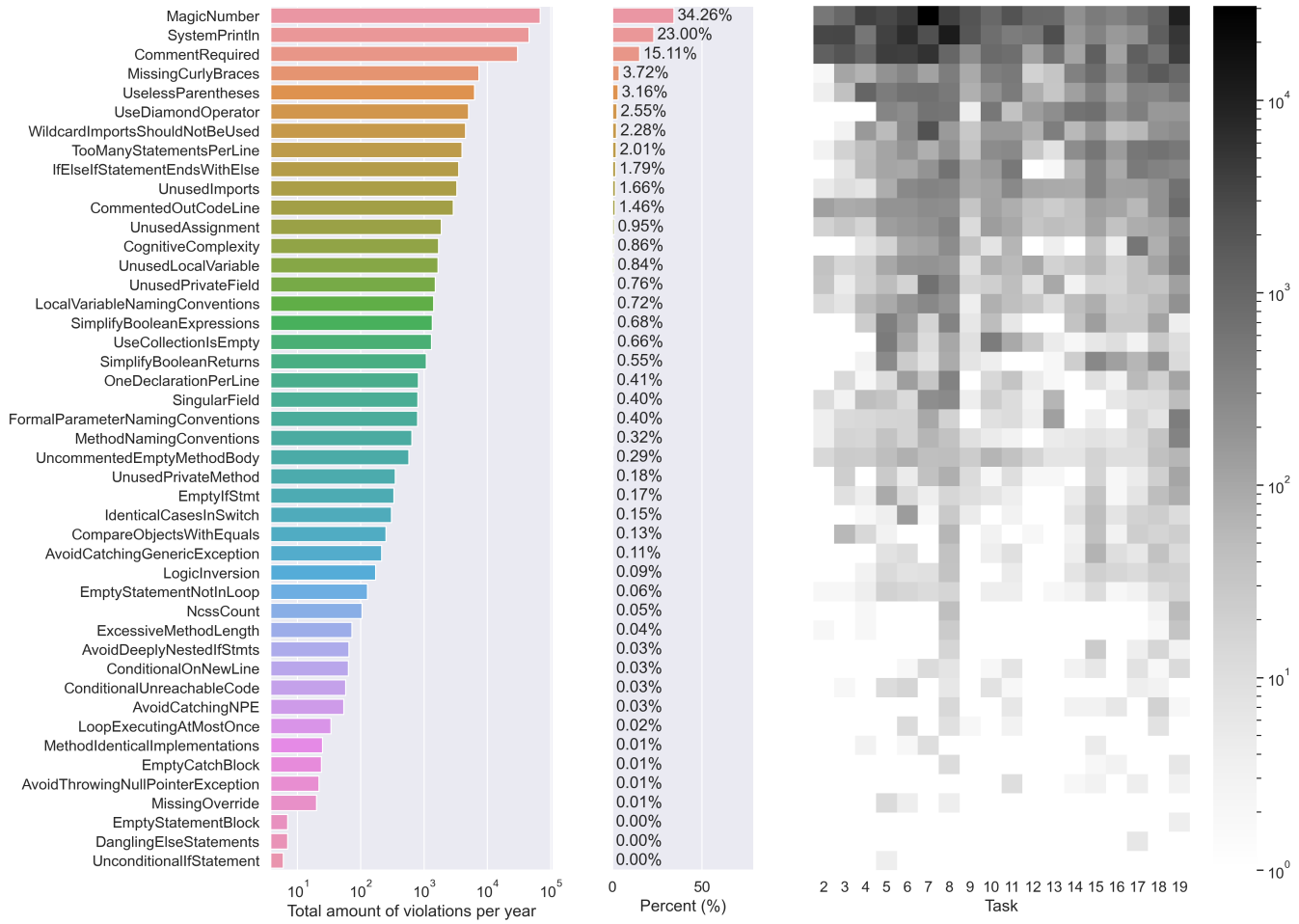
For this study, *CommentRequired* is of particular interest. *CommentRequired* denotes if JavaDocs are present in the analyzed code. From Task 8 and onwards, documentation and JavaDocs are mandatory for a passing grade. *CommentRequired* is the only violation of the rule set that has such requirements and is explicitly part of the learning objectives of the course. Figure 3 shows this violation and how it progresses over each course iteration. There is a clear drop after Task 8 and then a spike at Task 19. This spike can be explained by a multiplier of 4 at play, as students submit four variations of the Quicksort algorithm and readily copy and paste reusable code without improvement.

Finally, no statistically significant difference between TAs could be found in terms of the median of `V/KLOC`. A Kruskal-Wallis test was used as the groups were not normally distributed. A p-value < 0.05 would indicate a significant difference between the groups' median values for `V/KLOC`. Out of four course iterations and 18 analyzed tasks, no iteration has more than six assignments with a significant p-value. However, further post-hoc tests showed no common outlier.

5 DISCUSSION

In response to the first research question, *is there a noticeable improvement in code quality*, we find that the violations at first glance appeared to reduce across the course. On second glance, by removing the most dominant and arguably low-hanging fruit violations, the more serious and insidious violations in fact increased as the course progressed. As the tasks become more complex, one expects an upward trend. However, the results here indicate that simply looking at the totality of violations might lead to a misunderstanding and also that the nature, severity, and progression of violations should be taken into consideration. Consequently, more work is required to create a reusable, community-sourced index of violations that teachers can make use of. Prior works [4, 8, 15] have contributed curated lists of violations, and the ranking shown in Figure 1 adds a further perspective from a longitudinal dataset, with

Figure 1: The frequency and distribution of violations found in student submissions from 2018 to 2021. The left figure ranks violations in terms of their frequency. The right figure plots the distribution of violation intensity across the series of tasks.



violations from both PMD and SonarSource. In contrast, we have attempted to plot the progression of violations over time/task and in their totality to better reveal the behaviour amongst students.

In response to the second research question, *is there an observable difference in code quality amongst the TA groups*, no such difference could be found. The difference between TA groups was essentially random, and this was interpreted as a rather good thing. We had the expectation that with little training and only personal programming experience, TAs would cast considerable influence over their students' code quality. It must be remembered that we excluded the advanced groups from the analysis for practical reasons, however, we still included the majority of TAs. At the risk of furthering the burden of TAs, as reported in [29], and perhaps part of the solution to the lack of TA training [35], TAs might be able to focus some of their efforts on code quality and finding ways to incorporate SCA tools into their workflow. A group-level visualisation of violations, as shown in Figure 1 could act as a dashboard to drive the discussion and engagement with their students.

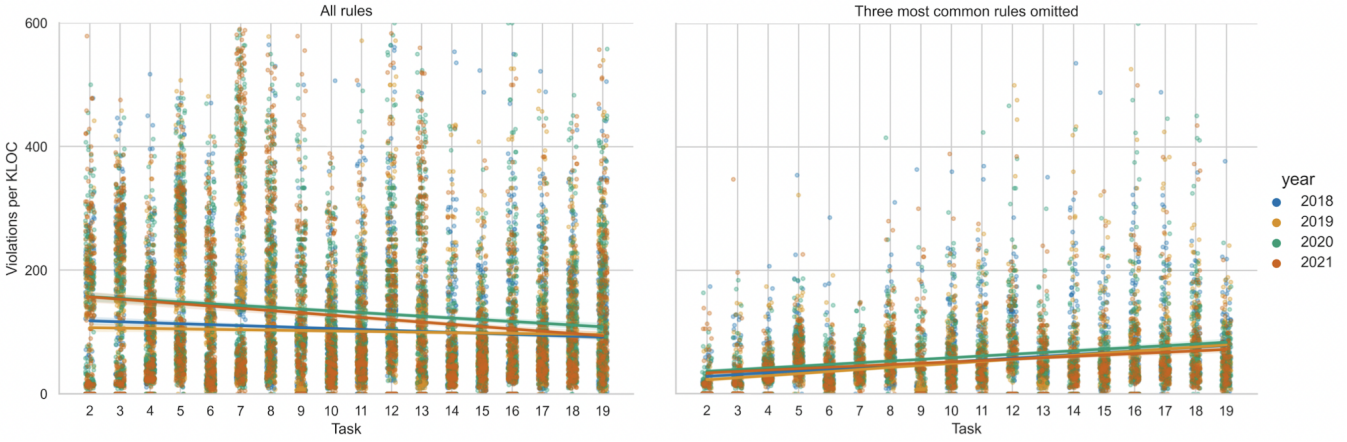
5.1 Threats to validity

Our work is subject to various threats to validity. First, we face the established shortcomings of SCA [10]. As mentioned in section 2, metrics used as a proxy for code quality may not necessarily correspond to what professionals consider "best practices".

Second, the dataset represents the real-world running of a course over four years and there are inevitable changes to the series of tasks that occurred due to course maintenance. Furthermore, some tasks encourage students to use `System.out.println()` which contributes to overrepresentation in the dataset.

Third, there is uncertainty present in the statistical analysis. Given the size of the TA groups ($n < 20$), a normal distribution could not be assumed. For instance, one student may be accountable for most violations in any group. Further work is required to delve into each group to reveal whether such outliers are having a dominant effect, or if the group converges to a reasonable distribution. We also acknowledge the poor sensitivity of RQ2; if the outcome would

Figure 2: The result of regression analysis over the students for each course iteration. When analyzing the 42 less prominent rules, students have an increase of `V/KLOC`. Thus, the measurable improvement is accredited to the three most common rules, *MagicNumber*, *CommentRequired* and *SystemPrintln*.



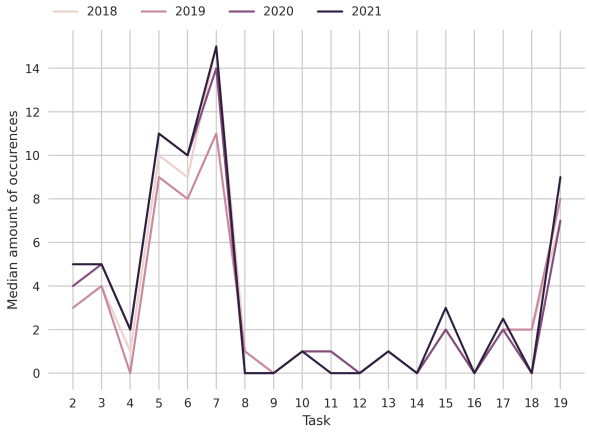
have been different, it might be wrong to infer that TAs affect code quality. At best, it should encourage further research in the field.

Finally, this work is also specific to the tasks and formative course design. An interesting comparison would be with courses that favour summative assessment or tasks that take longer than one week to complete. It is hoped that more courses at the introductory level draw inspiration from this and related works and focus on code quality at the earliest stages of a programmer's education.

5.2 Implications

Returning to the motivation behind this work, addressing technical debt as early as is feasible, we find the origins already manifesting in our own courses each year. Action is required, and we offer some practicable suggestions: (1) Teachers need to work with researchers of SCA tools and work towards pedagogical versions that are explainable to and actionable by students. (2) Teachers then need to weave code quality into their learning objectives. (3) TAs could be the key to unlocking this challenge, as they could incorporate (1) and (2) into their practice and lift some burden from the teacher whilst motivating students to care about this problem.

Figure 3: The median occurrences of the rule *CommentRequired*. Commenting code is mandatory after Task 8. The spike in Task 19 is due to duplication within the task.



6 CONCLUSIONS

The distance between an introductory programming course and a software developer's career may seem large, however from the earliest stages, the potential for future technical debt is already starting to emerge and grow. It is within these nascent stages that preventative action should be taken and targeting code quality is one potential starting point. Code quality analysis is essentially automatic and student code is increasingly archived in version control repositories, which lowers the threshold for easily accessing and reviewing student work. What remains is to figure out how best to blend code quality into the earliest stages of a programmer's education in a complementary and meaningful way; both reducing the load for teachers and TAs through automatic tool support and increasing the awareness of this important topic for students and their future careers.

ACKNOWLEDGMENTS

We would like to thank Professor Martin Monperrus at KTH Royal Institute of Technology, and his research engineers Sofía Bobadilla and Aman Sharma for their valuable input on this work. Thanks also to Trenton McKinney for providing excellent feedback on how to handle data frames in Python. Finally, many thanks to all the wonderful students and teaching assistants from INDA for all their hard work.

REFERENCES

[1] Eric Allman. 2012. Managing Technical Debt. *Commun. ACM* 55, 5 (May 2012), 50–55. https://doi.org/10.1145/2160718.2160733

[2] David J. Barnes and Michael Kölling. 2011. *Objects First with Java: A Practical Introduction Using BlueJ* (5th ed.). Prentice Hall Press, USA.

[3] Jürgen Börstler, Harald Störrle, Daniel Toll, Jelle van Assema, Rodrigo Duran, Sara Hooshangi, Johan Jeuring, Hieke Keuning, Carsten Kleiner, and Bonnie MacKellar. 2018. "I Know It When I See It" Perceptions of Code Quality: ITiCSE '17 Working Group Report. In *Proceedings of the 2017 ITiCSE Conference on Working Group Reports* (Bologna, Italy) *(ITiCSE-WGR '17)*. Association for Computing Machinery, New York, NY, USA, 70–85. https://doi.org/10.1145/3174781.3174785

[4] Dennis M. Breuker, Jan Derriks, and Jacob Brunekreef. 2011. Measuring Static Quality of Student Code. In *Proceedings of the 16th Annual Joint Conference on Innovation and Technology in Computer Science Education* (Darmstadt, Germany) *(ITiCSE '11)*. Association for Computing Machinery, New York, NY, USA, 13–17. https://doi.org/10.1145/1999747.1999754

[5] Neil C.C. Brown and Amjad Altadmri. 2014. Investigating Novice Programming Mistakes: Educator Beliefs vs. Student Data. In *Proceedings of the Tenth Annual Conference on International Computing Education Research* (Glasgow, Scotland, United Kingdom) *(ICER '14)*. Association for Computing Machinery, New York, NY, USA, 43–50. https://doi.org/10.1145/2632320.2632343

[6] W Conover and R Iman. 1979. *Multiple-comparisons procedures*. Technical Report LA-7677-MS. Los Alamos National Laboratory (LANL), Los Alamos, NM (United States). https://doi.org/10.2172/6057803

[7] Adrienne Decker, Phil Ventura, and Christopher Egert. 2006. Through the looking glass. *ACM SIGCSE Bulletin* 38 (3 2006), 46–50. Issue 1. https://doi.org/10.1145/1124706.1121358

[8] Tomáš Effenberger and Radek Pelánek. 2022. Code Quality Defects across Introductory Programming Topics. In *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education V. 1* (Providence, RI, USA) *(SIGCSE 2022)*. Association for Computing Machinery, New York, NY, USA, 941–947. https://doi.org/10.1145/3478431.3499415

[9] Khashayar Etemadi, Nicolas Harrand, Simon Larsén, Haris Adzemovic, Henry Luong Phu, Ashutosh Verma, Fernanda Madeiral, Douglas Wikström, and Martin Monperrus. 2021. Sorald: Automatic Patch Suggestions for SonarQube Static Analysis Violations. *CoRR* abs/2103.12033 (2021). https://arxiv.org/abs/2103.12033

[10] Sarah Fakhoury, Devjeet Roy, Adnan Hassan, and Vernera Arnaoudova. 2019. Improving Source Code Readability: Theory and Practice. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. IEEE, USA, 2–12. https://doi.org/10.1109/ICPC.2019.00014

[11] Consortium for IT Software Quality (CISQ). 2018. The Cost of Poor Quality Software in the US: A 2018 Report. Retrieved 2022-04-10 from https://www.it-cisq.org/the-cost-of-poor-quality-software-in-the-us-a-2018-report/The-Cost-of-Poor-Quality-Software-in-the-US-2018-Report.pdf

[12] Petri Ihantola and Andrew Petersen. 2019. Code complexity in introductory programming courses. (2019).

[13] ISO/IEC. 2001. *ISO/IEC 9126. Software engineering – Product quality*. ISO/IEC, Geneva, Switzerland.

[14] Oscar Karnalim and Simon. 2021. Promoting Code Quality via Automated Feedback on Student Submissions. In *2021 IEEE Frontiers in Education Conference (FIE)*. IEEE, 1–5. https://doi.org/10.1109/FIE49875.2021.9637193

[15] Hieke Keuning, Bastiaan Heeren, and Johan Jeuring. 2017. Code Quality Issues in Student Programs. In *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education*. ACM, New York, NY, USA, 110–115. https://doi.org/10.1145/3059009.3059061

[16] Hieke Keuning, Johan Jeuring, and Bastiaan Heeren. 2016. Towards a Systematic Review of Automated Feedback Generation for Programming Exercises. In *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education*. ACM, New York, NY, USA, 41–46. https://doi.org/10.1145/2899415.2899422

[17] Diana Kirk, Tyne Crow, Andrew Luxton-Reilly, and Ewan Tempero. 2020. On Assuring Learning About Code Quality. In *Proceedings of the Twenty-Second Australasian Computing Education Conference*. ACM, New York, NY, USA, 86–94. https://doi.org/10.1145/3373165.3373175

[18] Philippe Kruchten, Robert L Nord, and Ipek Ozkaya. 2012. Technical debt: From metaphor to theory and practice. *Ieee software* 29, 6 (2012), 18–21.

[19] Björn Latte, Sören Henning, and Maik Wojcieszak. 2019. Clean Code: On the Use of Practices and Tools to Produce Maintainable Code for Long-Living. In *Proceedings of the Workshops of the Software Engineering Conference 2019*, Vol. Vol-2308. CEUR-WS, University of Aachen, Germany, 96–99.

[20] Jinrui Li and Roger Barnard. 2011. Academic tutors' beliefs about and practices of giving feedback on students' written assignments: A New Zealand case study. *Assessing Writing* 16 (4 2011), 137–148. Issue 2. https://doi.org/10.1016/J.ASW.2011.02.004

[21] Fred Long, Dhruv Mohindra, Robert C Seacord, Dean F Sutherland, and David Svoboda. 2013. *Java coding guidelines: 75 recommendations for reliable and secure programs*. Addison-Wesley.

[22] Panagiotis Louridas. 2006. Static code analysis. *Ieee Software* 23, 4 (2006), 58–61.

[23] Andrew Luxton-Reilly, Simon, Ibrahim Albluwi, Brett A. Becker, Michail Giannakos, Amruth N. Kumar, Linda Ott, James Paterson, Michael James Scott, Judy Sheard, and Claudia Szabo. 2018. Introductory programming: a systematic literature review. In *Proceedings Companion of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education*. ACM, New York, NY, USA, 55–106. https://doi.org/10.1145/3293881.3295779

[24] Robert Cecil Martin. 2009. *Clean Code: A Handbook of Agile Software Craftsmanship - Robert C. Martin* (2 ed.). Prentice Hall, Upper Saddle River, New Jersey, USA.

[25] Tanya McGill and Simone Volet. 1995. An investigation of the relationship between student algorithm quality and program quality. *ACM SIGCSE Bulletin* 27 (6 1995), 44–48. Issue 2. https://doi.org/10.1145/201998.202012

[26] Valbona Muzaka. 2009. The niche of Graduate Teaching Assistants (GTAs): perceptions and reflections. *Teaching in Higher Education* 14 (2 2009), 1–12. Issue 1. https://doi.org/10.1080/13562510802602400

[27] Jernej Novak, Andrej Krajnc, et al. 2010. Taxonomy of static code analysis tools. In *The 33rd international convention MIPRO*. IEEE, 418–422.

[28] Oracle. 1997. Java Code Conventions. Retrieved 2022-05-11 from https://www.oracle.com/technetwork/java/codeconventions-150003.pdf

[29] Chris Park and Marife Ramos. 2002. The donkey in the department? Insights into the graduate teaching assistant (GTA) experience in the UK. *Journal of Graduate Education* 3, 2 (2002), 47–53.

[30] PMD. 2022. Source Code Analyzer. Retrieved 2022-05-11 from https://pmd.github.io/

[31] Chris Rust, Berry O'Donovan, and Margaret Price. 2005. A social constructivist assessment process model: how the research literature shows us this could be best practice. *Assessment & Evaluation in Higher Education* 30 (6 2005), 231–240. Issue 3. https://doi.org/10.1080/02602930500063819

[32] SonarSource. 2022. Java Code Quality and Code Security. Retrieved 2022-05-10 from https://www.sonarsource.com/java/

[33] Nghi Truong, Paul Roe, and Peter Bancroft. 2004. Static Analysis of Students' Java Programs. In *Proceedings of the Sixth Australasian Conference on Computing Education - Volume 30* (Dunedin, New Zealand) *(ACE '04)*. Australian Computer Society, Inc., Australia, 317–325. https://doi.org/10.5555/979968.980011

[34] Allan Vermeulen, Scott W Ambler, Greg Bumgardner, Eldon Metz, Trevor Misfeldt, Patrick Thompson, and Jim Shur. 2000. *The elements of Java (tm) style*. Number 15. Cambridge University Press.

[35] Arto Vihavainen, Thomas Vikberg, Matti Luukkainen, and Jaakko Kurhila. 2013. Massive increase in eager TAs. In *Proceedings of the 18th ACM conference on Innovation and technology in computer science education - ITiCSE '13*. ACM Press, New York, New York, USA, 123. https://doi.org/10.1145/2462476.2462508 [683] i den stora litteraturstudien av Luxton..

[36] Navé Wald and Tony Harland. 2020. Rethinking the teaching roles and assessment responsibilities of student teaching assistants. *Journal of Further and Higher Education* 44 (1 2020), 43–53. Issue 1. https://doi.org/10.1080/0309877X.2018.1499883

[37] Dawei Yang, Daojiang Wang, Dongming Yang, Qiwen Dong, Ye Wang, Huan Zhou, and Daocheng Hong. 2020. DevOps in Practice for Education Management Information System at ECNU. *Procedia Computer Science* 176 (2020), 1382–1391. https://doi.org/10.1016/j.procs.2020.09.148 Knowledge-Based and Intelligent Information & Engineering Systems: Proceedings of the 24th International Conference KES2020.

[38] Marsha Zaidman. 2004. Teaching defensive programming in Java. *Journal of Computing Sciences in Colleges* 19, 3 (2004), 33–43.