Common Code Quality Issues of Novice Java Programmers: A Comprehensive Analysis of Student Assignments

Christina Julia Kohlbacher[1], Michael Vierhauser[2][a] and Iris Groher[1][b]
[1]*Institute of Business Informatics - Software Engineering, JKU Business School, Johannes Kepler University, Linz, Austria*
[2]*LIT Secure and Correct Systems Lab, Johannes Kepler University, Linz, Austria*

Keywords: Novice Programmers, Code Quality, Best Practices, Static Code Analysis.

Abstract: Starting to learn programming is often perceived as being quite tedious by students at the bachelor level. Many programming courses thus face high drop-out rates and moderate results for those who pass. This problem is exacerbated when teaching programming to students enrolled in non-computer science curricula. To overcome these issues, we have developed a novel didactic concept based on peer learning, tutoring, dedicated teaching, and learning material that supports individuality and competency-based learning. Our current focus lies on teaching basic programming principles, but to further support our students and foster a positive learning experience, we want to learn more about the difficulties they are facing, particularly with respect to best practices, coding conventions, and code quality. We, therefore, performed a static code analysis of homework assignments of students participating in our introductory programming course for two consecutive years. We analyzed over 13,000 Java files and more than 400,000 lines of Java code to identify common code quality issues faced by our students. Our analysis shows that the majority of rule violations are related to coding style. The violations do not differ much with respect to the topics covered in the homework assignments, and hardly change over time. The more lines of code the students write, the more rules are violated. Based on our findings we present concrete recommendations on how to support novice programmers in improving their code quality.

1 INTRODUCTION

Introductory programming classes are, for most computer science students – and studies from related fields such as Business Informatics – the first time they encounter programming, gain experience with programming languages, and need to understand the concept of algorithmic thinking. Many first-semester introductory programming courses, however, face significant drop-out rates and moderate results of those who pass (Lahtinen et al., 2005; Milne and Rowe, 2002; Watson and Li, 2014). To address this issue, and to provide students with a better learning experience, in recent years, several didactic concepts, specifically targeting first-semester programming, have been introduced with the goal of fostering collaborative learning and teamwork (Sabitzer et al., 2020; Krusche et al., 2020; Williams and Upchurch, 2001; McDowell et al., 2002).

At our university, for the past decade, we have been teaching introductory programming classes for Business Informatics, and recently also Business Administration students, and have experienced similar issues, as students are typically quite diverse, with different educational and cultural backgrounds. To overcome these issues we recently adopted a novel didactic concept based on peer learning, tutoring, dedicated teaching, and learning material that supports individuality and competency-based learning.

In general, the results have improved, with fewer drop-outs and more students passing the course, while the students' experience appears to be positive. However, based on the feedback we received, the focus so far has mostly been on teaching basic programming concepts while putting little emphasis on overall code quality aspects. As this class is a precursor to advanced computer science classes in later semesters, this has turned out to be a pain point for students later on. While basic programming skills are no doubt important, if students are not guided to write "good" code in an early stage, this might result in additional time and effort students have to invest in later

[a] https://orcid.org/0000-0003-2672-9230
[b] https://orcid.org/0000-0003-0905-6791

Kohlbacher, C., Vierhauser, M. and Groher, I.
Common Code Quality Issues of Novice Java Programmers: A Comprehensive Analysis of Student Assignments.
DOI: 10.5220/0011715400003470
In *Proceedings of the 15th International Conference on Computer Supported Education (CSEDU 2023) - Volume 2*, pages 349-356
ISBN: 978-989-758-641-5; ISSN: 2184-5026

classes (Keuning et al., 2017). Moreover, knowing which mistakes novices are likely to make also helps when designing teaching and learning material or educational tools (Altadmri and Brown, 2015; Brown and Altadmri, 2017).

In order to gain insights into the code quality of student assignments, and the difficulties and challenges students face, we have conducted a study analyzing homework assignments with regards to code quality and coding best practices. We have collected data from the past two years of our course and analyzed the assignments of 284 students. We performed a static code analysis of more than 13,000 Java files comprising over 400,000 lines of code. In total, we checked 108 different rules with respect to best practices, code style, design, error-prone code, and performance.

We have observed that the majority of rule violations are related to coding style. The violations do not differ much with respect to the topics covered in the homework assignments and do not change over time. The more lines of code the students write, the more rules are violated.

The remainder of this paper is structured as follows. We first discuss related work in Section 2. In Section 3, we then provide a brief overview of our course setup and the teaching strategies. In Section 4, we describe our research method, data collection, and research questions and report on the results of our data analysis. We then discuss implications, lessons learned, and recommendations in Section 5. We finally present threats to validity in Section 6.

2 RELATED WORK

Different authors have performed static analysis of source code written by students as part of a university course. Edwards *et al.* (Edwards et al., 2017) performed a static analysis of Java code of student groups with different levels of experience. The authors used Checkstyle and PMD to identify the most frequent rule violations of four different programming courses. The data set comprised nearly ten million static analysis errors, made by more than 3,000 students over a five-semester time period. The analysis revealed that formatting and documentation issues were most common. Furthermore, the results showed that, with more experience, fewer errors are made. However, the most frequent errors were consistent across all analyzed experience levels.

Albluwi and Salter (Albluwi and Salter, 2020) analyzed the source code of 968 students in a Java course for beginners from a three-semester period. With Checkstyle, PMD, and FindBugs, more than one million issues were identified. Similar to the results of Edwards *et al.* (Edwards et al., 2017), the results show that *Formatting* and *Documentation* were the most frequent error categories. Their overall results showed that those two categories account for 60% of the total errors found. The authors also provided static analysis tools to the students and monitored the difference in error numbers between the initial and the final submissions. Most of the issues got fixed before the final submission. More specifically, only 20,3% of the total number of issues occurred in the final submissions.

Delev and Gjorgjevikj (Delev and Gjorgjevikj, 2017) present their static analysis results of *C* source code from novice programmers. The data set comprised about 14,000 exam submission files from a three-year period. The authors found that the most common flaws in the submission files were issues like uninitialized or unused variables, as well as logical operation errors.

Keuning *et al.* (Keuning et al., 2017) analyzed the type and frequency of code quality issues that occur in two million Java programs of novice programmers. They also investigate whether students are able to solve these issues and whether solving code quality issues is improved when students have code analysis tools installed. Interestingly, the study by Keuning *et al.* revealed that most issues are rarely fixed, especially when they are related to modularization. Another finding was that the use of tools has little effect on issue occurrence.

Brown and Altadmri (Brown and Altadmri, 2017) investigate in their study whether educators have an accurate understanding of the mistakes that students often make. The study reveals that educators have only a weak consensus about the frequency of errors novice programmers make. Also, the beliefs of the educators are not in line with the analysis results of the Java programs. This mismatch makes more studies on programming issues necessary to help educators understand where students need better support.

3 COURSE SETUP

In this section, we briefly describe the main concepts, as well as the structure of our introductory programming course. The course is mandatory in the first semester of the Business Informatics bachelor program at our university. Each year approximately 150 to 200 students are attending the course. The goal of the course is to teach basic programming principles and to further introduce students to software development with the Java programming language. Java was chosen deliberately, as subsequent courses, building on top of this course, rely on and require students to have basic programming knowledge in Java. The course starts with a basic introduction to programming principles (statements, conditions, loops, data types), covers the foundations of Java programming (methods and arrays), and concludes with object orientation, and basic inheritance principles.

With 6 ECTS credits (corresponding to approximately 150 hours of work), the course consists of two separate parts. First, a weekly lecture, where attendance is optional (but highly recommended), and second, a weekly exercise with mandatory attendance.

Lecture: This part consists of a 90-minute slide-based lecture, covering one topic per week, over the course of 14 weeks. The slides are augmented with live-coding sessions in which the lecturer presents programming examples, supporting students with hands-on experience. Slides and programming examples are provided to the students, alongside an optional textbook. We encourage students to read the book chapter and/or familiarize themselves with the topic that is covered before the lecture and prepare questions.

Excercise: The exercise is synchronized with the lecture and takes place in the same week (typically one or two days after the lecture). Students are divided into several smaller exercise groups of approximately 30 people. In order to provide an optimal learning experience, we have recently adopted a new teaching concept specifically designed for teaching STEM classes at the university level. It has a strong focus on discovery and cooperative learning, and instead of solely relying on front-of-class teaching, the exercise is split into three parts.

In the first part (*Repetition and Questions*) the lecturer summarizes the most important concepts of the previous lecture and provides additional examples and code snippets. During this time, students are encouraged to ask clarification questions. The second part (*Discovering*) is dedicated to self-learning. Students have time to take a look at what we call "Reading Corners", where we provide sample solutions, step-by-step exercises, and examples related to the topic. The third part (*Pair Programming*) of the 90-minute exercise is dedicated to teamwork and pair programming. Students work together in groups of two or three on their weekly assignments.

Beginning with 2020, due to the COVID pandemic, lectures and exercises were both held online. Despite not having students physically present in the classroom, we tried to follow the same teaching concepts previously introduced as close as possible. Pair programming sessions were performed in breakout Zoom rooms, and the lecturer was called for help when needed. Students shared their screens or used the collaborative online editor to work together on the assignments. Additionally, a lecturer visited each team at least once during the pair programming session.

Homework Assignments: The weekly homework assignments are composed of several smaller examples, typically five to six individual tasks, including different types of tasks. Besides "traditional" programming tasks, we also include tasks such as reading, understanding, and describing code snippets. Each student has to submit the weekly homework assignment tasks electronically within one week.

During the semester, we hand out ten assignments out of which a minimum of eight must be submitted. If more are submitted, we only consider the best eight assignments when calculating the final grade. With regards to grading, assignments are manually graded by a tutor, typically a student in a higher semester, that has Java programming experience. The tutors are provided with grading guidelines provided by the lecturer to ensure consistent grading. As part of the feedback, students not only receive points for each assignment, but tutors provide detailed feedback about the errors made by the students as well as the quality of the code and the way the solution is implemented.

Additionally, complementing the lectures and exercises, a weekly tutorial is offered. Attendance to the tutorial is optional and run by a tutor where support is provided to students who experience problems or have questions about the assignment.

Teaching Materials: Besides lecture slides and homework assignments, the students receive different learning materials for studying the course contents. They are provided with supporting literature in the form of books, summary slides of the exercises with additional examples, and a weekly Reading Corner that contains sample solutions and step-by-step examples to foster pattern recognition and discovery learning. In addition, links to videos are provided that contain further examples and coding sessions. While the teaching material does include guidelines, recommendations, and best practices for Java programming, it is part of the lecture and we do not present this as a separate topic, but blend these into the first three lectures and exercises. We present best practices and guidelines regarding variable, method, (and later on) class names as well as braces in control structures.

Exam: Students have to take an exam at the end of the semester. The final grade is a result of both the assignment and exam results. In order to pass the course, students have to (1) receive at least 50% of the total points for the exam, and (2) hand in at least eight out of the ten homework assignments, with at least 50% of the available points for the homework assignments.

4 STUDY

In order to gain insights into common mistakes students make, we collected and analyzed data from homework assignments of the last two years of our introductory programming course. We performed a static code analysis of all submitted assignments and analyzed the results with respect to mistakes and violations of practices and the development of these violations over the course. Based on this data, we answer the following two research questions:
RQ1: What are the most common issues with regard to code quality? With this first research question, we investigated which best practice and code quality rules were violated most by the students of our course.
RQ2: How do the quality issues develop over the course and are they related to different assignment topics? For the second research question, we investigated the different types of rules and the number of violations with respect to the different topics that were part of the assignments. We were particularly interested in whether students learn to follow best practices and whether rule violations differ between the assignment topics.

4.1 Data Collection and Analysis

We collected data from the assignments submitted by our students from the winter terms of 2019 and 2020. Therefore, we downloaded all submitted Java files from the course management platform of our university and anonymized the data for privacy protection. We removed the student names and IDs, but retained information regarding the year, the topic, and the assignment the files belong to.

For the analysis, we used the static code analysis tool PMD[1] with a custom set of rules. Starting from the 445 rules included in the default PMD rule set, we removed rules that were not deemed relevant for the analysis, based on the following exclusion criteria: we excluded (1) all non-Java rules. We further eliminated (2) all rules that cover aspects not relevant for our course (e.g., rules from categories such as *Documentation*, *Multithreading*, and *Security*) as these are not part of the topics covered in introductory programming. The final rule set contained 108 rules from five different categories: 15 rules from the *Best Practices* category, 21 rules from *Code Style*, 20 from *Design*, 38 from *Error Prone*, as well as 14 from *Performance* (see Figure 1a).

We developed several Python scripts for automatically extracting data and analyzing the Java files for rule violations. We further calculated the lines of code for each submitted assignment. The result analysis and visualization were performed with the data analytics platform *Qlik Sense*[2].

4.2 Results

Table 2 provides an overview of the collected data. In total, we analyzed data from homework assignments of 284 students participating in our introductory programming course for two consecutive years. We had around 20% more participants in 2020 compared to 2019 which lies within the typical limits of variation we see in our first-semester courses. In total, we analyzed more than 13,000 Java files with a total of over 400,000 lines of Java code for all assignments. In 2019, we had on average, 113 submissions per assignment with a maximum of 127 submissions (assignment 2) and a minimum of 73 submissions (assignment 10). In 2020, we had 129 submissions on average, with a maximum of 154 (assignment 1) and a minimum of 85 (assignment 10).

In total, the submitted homework assignments contained over 60,000 rule violations, corresponding to roughly 14.5 violations per 100 LLOC. The violations per 100 LLOC are slightly higher for 2019, even though we observed a larger number of rules violated in 2020 (62 PMD rules in 2020 compared to 56 in 2019).

4.2.1 Common Code Quality Issues

To identify common violations of best practices and code quality rules in our course, we analyzed the distribution of rule violations among the five categories.

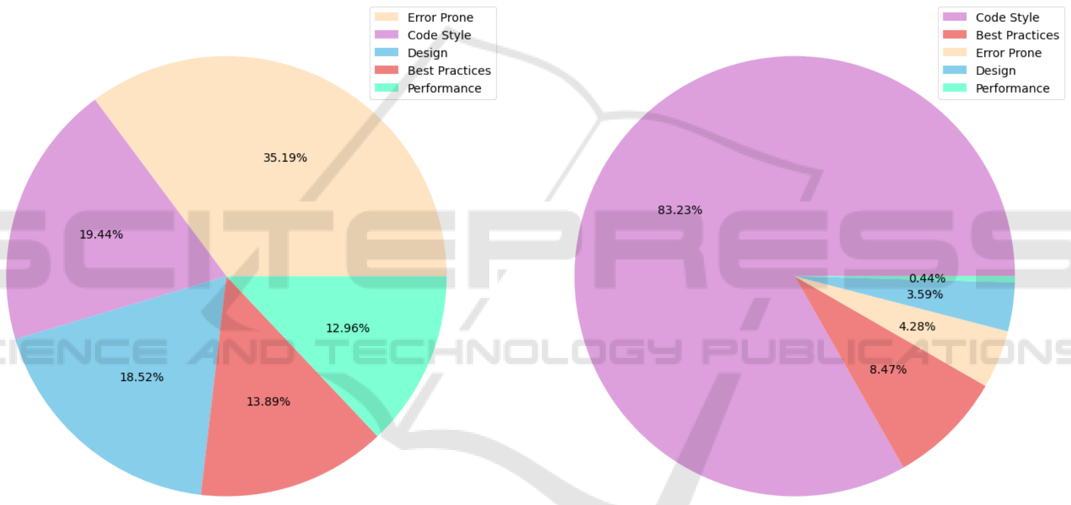
Figure 1a provides an overview of the distribution of the analyzed rules among the five rule categories. With over one third (35.2%), *Error Prone* is the most examined category. This category includes rules such as empty statements or missing breaks in switch statements. *Code Style* and *Design*, make up 19,4% and 18,5% of the rule set. Code Style rules include naming conventions or duplicate imports. Design rules include deeply nested if-statements, or excessive method or class lengths. *Best Practices* and *Performance* are the least represented categories with less than 15% each. Examples of Best Practice rules include rules checking for unused imports, or unused

[1]https://pmd.github.io

[2]https://www.qlik.com/de-de/products/qlik-sense

Table 1: Overview of top 10 violated PMD rules (PMD, 2021) and their categories (CS=Code Style, BP=Best Practices, D=Design).

Rule	Cat.	Description
ShortVariable	CS	Checks minimum length of field, local variable, or parameter names
ControlStatement Braces	CS	Checks braces on 'if … else' statements and loops
Useless Parentheses	CS	Checks for superfluous parentheses
ClassNaming Conventions	CS	Checks for standard Java naming conventions of a class (camel case, no special characters, …)
LocalVariable NamingConvents	CS	Checks for standard Java naming conventions of local variables
AvoidReassigning Parameters	BP	Checks if a parameters of methods are reassigned
UnusedLocalVariable	BP	Checks if a variable is declared and/or assigned, but not used
UnnecessaryLocal BeforeReturn	CS	Checks for unnecessary local variables in methods
SimplifyBoolean Expression	D	Checks unnecessary comparisons in boolean expressions
ConfusingTernary	CS	Checks for confusing (negated) if expressions with an "else" clause



(a) Distribution of Checked Rules among Categories (b) Distribution of Rule Violations among Categories

Figure 1: Rule and Violation Distribution for Checked PMD Rules.

methods and local variables. Performance rules include instantiations of objects and inefficient checks (cf Table 1).

By far the most violations we observed fall into the Code Style category (83,2%), followed by Best Practices (8,5%) and Error-Prone (4,3%). Only few rule violations belong to the categories Design (3,6%) and Performance (0,4%). Figure 1b shows the distribution of rule violations among these categories in all assignments of 2019 and 2020.

Figure 2 presents the top ten violated rules of all homework assignments for the years 2019 and 2020. The calculation is based on the number of rule violations per 100 LLOC. Colors represent the categories (cf. Figure 1a) the rules belong to. Seven out of ten, including the top five, rule violations belong to the *Code Style* category. Two rules belong to the *Best*

Table 2: Overview of the collected data and static code analysis results.

Year	2019	2020	Total
Nr. of Students	127	157	284
Java Class Files	6,426	7,239	13,665
Total LLOC	162,611	252,091	414,702
Violated PMD Rules	56	62	63
Rule Violations	25,626	34,631	60,257
Violations/100 LLOC	15.76	13.74	14.53
Avg. Violation/Ass.	20.18	22.06	21.22
Avg. LLOC/Ass.	128.04	160.57	146.02

Practices category and one to *Design*. Most rule violations are related to short variable names. Class and local variable naming conventions are the fourth and fifth of all rule violations. Problems related to parentheses are second and third of all violations. The other rule violations among the top ten are reassigning parameters, unused local variables, unnecessary local variable before return, complex boolean expressions, and confusing ternaries.

With regards to RQ1 and the most common issues we have observed, simple code style/coding convention violations were by far the most common violations. This includes naming conventions for variables, methods, and classes, as well as structuring loops and conditions. This is not surprising, as the topics covered by our course mostly deal with simple programs, control structures, and methods. All of these guidelines were part of the course material, and were also mentioned by the tutors during assignment grading. However, while we did encourage students to follow them, we did not strictly enforce them, e.g., by deducting points.

4.2.2 Development Throughout the Course

In a second step, we were interested if, and to what extent, quality issues changed over the course of the semester. Therefore, we analyzed the different types of rules and the number of violations with regards to the different topics that were part of the assignments. We were further interested in the change of violated rules to uncover any connections between topics and specific rules.

For this, we examined the top ten violated rules for each of the ten assignments. This analysis revealed that the most violated rule over all assignments, short variable name, is the top violated rule in seven out of the ten assignments. In the remaining three assignments it was the second most violated rule. In the first assignment, in which students had to develop simple programs without any control structures, local variable naming conventions were violated the most. In assignments three (covering the topics of if-statements and loops) and four (covering the topic of methods), we observed that students had the most difficulties with parentheses. Overall, there are only slight variations with regard to the violated rules among the assignments. For example, rules related to switch statements mostly occur in assignment three, as this is the assignment that explicitly covers this topic. Generally, the number of violations related to naming conventions grows over the course of the semester, which means that students tend to put less emphasis on naming conventions as assignments get more complex and comprehensive.

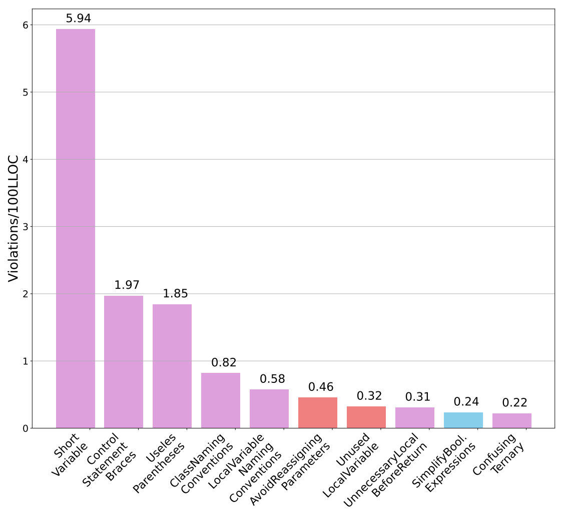


Figure 2: Top Ten Violated Rules Across all Assignments (purple: Code Style, red: Best Practices, blue: Design).

Figure 3 provides an overview of the number of rule violations per 100 LLOC for each of the ten assignments separately for 2019 and 2020. Overall, the number of rule violations in relation to the lines of code students write grows over the course. The more complex the assignments get, the more rule violations are detected. The numbers are quite consistent for 2019 and 2020, except for assignment 5 in which the 2020 students performed better, and assignments 8 and 9 in which the 2019 students performed better.

Over the ten assignments, we are facing a drop out of students in our course. Also, to pass the course, students have to hand in at least eight out of the ten assignments. While the first eight assignments are typically handed in by the majority of the students, this number typically drops for assignments nine and ten. In 2019, for example, assignments 9 and 10 were only handed in by 92 and 73 students (out of 127 students that were part of the course at the beginning), respectively. We can observe a similar drop in 2020.

With regards to RQ2, how quality issues develop over the course, if they are related to different assignment topics, we can conclude that the types of rule violations do not change significantly, and the number grows with the size of the programs. As discussed above, the fact that the types of rule violations stay rather consistent over the course is not surprising. Still, we were hoping that basic code style would improve since we covered the rules in the lecture and the tutors continuously made remarks. Unfortunately, we could not observe a learning effect with respect to code quality in our course.

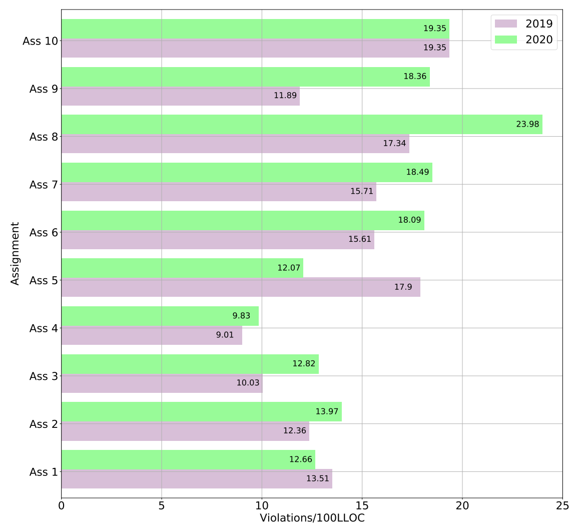


Figure 3: Violations/100LLOC per Assignment for 2019 and 2020.

5 DISCUSSION

In this section, we discuss the results and findings from our analysis and present three recommendations intended to establish best practices and improve code quality.

While we did introduce a small set of Java programming practices and guidelines early on in our course, and tutors repeatedly reminded students during the assignments, this was not the main focus of the lectures and assignments. Thus, students did pay scant attention to these guidelines which is consistent with the results we observed for the different assignments. While some guidelines, especially those pertaining to object-oriented programming, are only important in later parts of the course, fundamental ones such as naming conventions affect even simple programs. One concrete improvement we are planning in our course is to gradually introduce guidelines throughout the semester and repeat them after each lecture, not just as a side note, but as an integral part of the teaching material that will also be enforced more rigorously during assignment and exam grading. We also plan to include best practices in the weekly Reading Corner as part of the assignment.

Repetition: Introduce Best Practices and guidelines early on and keep repeating them throughout the course. Provide example solutions that focus on code quality to foster pattern recognition and discovery learning.

The weekly homework assignments in our course include different types of tasks. They contain programming tasks as well as tasks such as reading, understanding, and describing code snippets. The different tasks support individual learning paths, needs, and interests (Sabitzer and Strutzmann, 2013). One concrete step we plan to address these code quality issues in the future is to include new types of tasks in homework assignments. These tasks could include code snippets in which the students have to detect and fix code quality issues.

Specific Tasks: Provide tasks that explicitly focus on code quality.

Analyzing the top ten rule violations throughout the course revealed that students have problems with naming conventions and parentheses. There are only slight variations among the assignments and the topics covered. Existing work shows that novice programmers hardly fix code quality issues reported by code analysis tools (Keuning et al., 2017) even when explicitly presented. We thus suggest limiting the analyzed rules to the ten most common ones. Reports created by code analysis tools might be too complex and detailed for novice programmers. One way to address this could be customized rule violation messages that are easy to understand for beginners. In the future, we will use a static code analysis tool such as PMD but limit the rule set and also present the results to the students in an easy-to-understand manner.

Focus: Limit the rules to the ten most important ones and provide automated feedback to the students. Present simple and easy-to-understand rule violation messages.

6 THREATS TO VALIDITY

Our research is subject to a number of threats to validity. Our study is limited to one university and the number of students participating in the course, which is a threat to external validity. The data we analyzed covers two years of the course. As their results only differ slightly, we are confident that the results would be similar for further years. Moreover, as our findings yield similar results as other studies, we are confident that some observations are generalizable for other introductory programming courses.

The number of submitted assignments is inconsistent throughout the semester. This is partly because only eight out of ten assignments have to be submitted in order to pass the course. Also, some students quit the course and only submit the first few assignments. Our analysis does not confirm that the first assignments, which include most students, contain more rule violations than the latter ones.

We selected a specific subset of the PMD default rule set for our analysis (108 out of 445), posing a threat to construct validity. We only chose rules suitable for the Java programming language and excluded rules that cover topics not relevant to our course.

7 CONCLUSION

In this paper, we report on a study analyzing code quality in an introductory programming course. The main goal of this work gain insights into difficulties our students are facing, particularly with respect to best practices, coding conventions, and code quality. We performed a static code analysis of homework assignments of the previous two iterations of our course. The findings show that the majority of rule violations are related to coding style without much difference with respect to the topics of our course. Also, the rule violations do not change much over time.

Based on our analysis we discuss lessons learned and recommendations for improving the code quality of novice programmers. We suggest repetitively including best practices in teaching and learning materials and presenting only a very limited set of rules to the students in order to foster deeper understanding, without creating unnecessary confusion. Assignments explicitly focusing on code quality might further improve the learning outcomes.

REFERENCES

Albluwi, I. and Salter, J. (2020). Using Static Analysis Tools for Analyzing Student Behavior in an Introductory Programming Course. *Jordanian Journal of Computers and Information Technology (JJCIT)*, 6.

Altadmri, A. and Brown, N. C. (2015). 37 million compilations: Investigating novice programming mistakes in large-scale student data. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, SIGCSE '15, page 522–527, New York, NY, USA. Association for Computing Machinery.

Brown, N. C. C. and Altadmri, A. (2017). Novice java programming mistakes: Large-scale data vs. educator beliefs. *ACM Transactions on Computing Education*, 17(2).

Delev, T. and Gjorgjevikj, D. (2017). Static analysis of source code written by novice programmers. In *Proceedings of the 2017 IEEE Global Engineering Education Conference*.

Edwards, S. H., Kandru, N., and Rajagopal, M. B. (2017). Investigating Static Analysis Errors in Student Java Programs. In *Proceedings of the 2017 ACM Conference on International Computing Education Research*. ACM.

Keuning, H., Heeren, B., and Jeuring, J. (2017). Code quality issues in student programs. In *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education*, pages 110–115. ACM.

Krusche, S., von Frankenberg, N., Reimer, L. M., and Bruegge, B. (2020). An interactive learning method to engage students in modeling. In *Proceedings of the ACM/IEEE 42nd Int'l Conference on Software Engineering: Software Engineering Education and Training*, pages 12–22.

Lahtinen, E., Ala-Mutka, K., and Järvinen, H.-M. (2005). A study of the difficulties of novice programmers. *SIGCSE Bull.*, 37(3):14–18.

McDowell, C., Werner, L., Bullock, H., and Fernald, J. (2002). The effects of pair-programming on performance in an introductory programming course. *SIGCSE Bull.*, 34(1):38–42.

Milne, I. and Rowe, G. (2002). Difficulties in Learning and Teaching Programming—Views of Students and Tutors. *Education and Information Technologies*, 7(1):55–66.

PMD (2021). Ruleset. https://pmd.github.io/latest/pmd_rules_java.html. [Last accessed 01-01-2022].

Sabitzer, B., Groher, I., Sametinger, J., and Demarle-Meusel, H. (2020). Cool programming: Improving introductory programming education through cooperative open learning. In *Proceedings of the 2020 9th International Conference on Educational and Information Technology*, ICEIT 2020, page 95–101, New York, NY, USA. ACM.

Sabitzer, B. and Strutzmann, S. (2013). Brain-based programming: A new concept for computer science education. In *Proceedings of the 18th ACM Conf. on Innovation and Technology in Computer Science Education*, ITiCSE '13, page 345, New York, NY, USA. ACM.

Watson, C. and Li, F. W. (2014). Failure rates in introductory programming revisited. In *Proceedings of the 2014 Conf. on Innovation & Technology in Computer Science Education*, ITiCSE '14, page 39–44, New York, NY, USA. Association for Computing Machinery.

Williams, L. and Upchurch, R. L. (2001). In support of student pair-programming. *SIGCSE Bull.*, 33(1):327–331.