IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 29 July 2024, accepted 14 August 2024, date of publication 19 August 2024, date of current version 27 August 2024.

Digital Object Identifier 10.1109/ACCESS.2024.3445432

RESEARCH ARTICLE

Does ChatGPT Help Novice Programmers Write Better Code? Results From Static Code Analysis

PHILIPP HAINDL AND GERALD WEINBERGER
Department for Computer Science and Security, St. Pölten University of Applied Sciences, 3100 St. Pölten, Austria
Corresponding author: Philipp Haindl (philipp.haindl@fhstp.ac.at)

ABSTRACT In the realm of AI-enhanced programming education, there is growing interest in using such tools to help students understand good coding principles. This study investigates the impact of ChatGPT on code quality among part-time undergraduate students in introductory Java programming courses, who lack prior Java experience. The source code of 16 students from the control group (without ChatGPT) and 22 students from the treatment group (with ChatGPT) who completed identical programming exercises focused on coding conventions was analyzed. Static code analysis tools assessed adherence to a common coding convention ruleset and calculated cyclomatic and cognitive complexity metrics. The comparative analysis shows that the ChatGPT-assisted group significantly improved code quality, with fewer rule violations and reduced cyclomatic and cognitive complexities. The treatment group adhered more closely to coding standards and produced less complex code. Violations primarily occurred in line length, final parameters, and the extensibility of object-oriented programming (OOP). These findings suggest that ChatGPT can be beneficial in programming education by helping students write cleaner, less complex code and adhere to coding conventions. However, the study's limitations, such as the small sample size and novice status of participants, call for further research with larger, more diverse populations and different educational contexts.

INDEX TERMS Programming education, ChatGPT large language models, static code analysis.

I. INTRODUCTION

ChatGPT, an advanced Large Language Model (LLM) developed by OpenAI [1], has emerged as a significant tool capable of generating human-like text based on the input it receives. Since its introduction, ChatGPT has been leveraged across various domains, including coding and programming education. The capacity of ChatGPT to understand and generate code snippets in response to natural language prompts signifies a transformative potential for educational practices, especially in enhancing learning experiences and outcomes in programming courses. As the era of ChatGPT unfolds, programming education must identify the most effective ways to integrate and apply chatbots like ChatGPT in the classroom. It is essential that students possess the ability to evaluate the quality of AI-generated code while also enhancing their programming fluency and critical thinking skills [2].

Code quality is a critical aspect of software development, encompassing properties like code structure, code layout, and statement quality. Previous studies have analyzed the static code quality of student code [3], [4], [5], common mistakes made by students when learning to program [6], and the semantic style of code [7].

This study examines the impact of ChatGPT on the quality of code produced by students in an introductory Java course. It complements our previous qualitative study regarding the suitability of ChatGPT for programming exercises and typical application scenarios from students' perspective [8]. In this survey-based study, we also asked students about the additional required effort to adapt ChatGPT's generated code to the concrete programming task of an exercise, which they predominantly assessed as either high or very high. Hence, we can conclude that even when students use ChatGPT in an assistive manner, the quality of a student's submitted code

The associate editor coordinating the review of this manuscript and approving it for publication was Alba Amato.

for an exercise is an amalgamation of ChatGPT's generated code and the student's own work. In the presented study we employed static code analysis to detect rule violations to Java coding standards as well as cyclomatic and cognitive complexity in the source code of two groups and analyzed the deviations in the measurements. One group of students employed ChatGPT to implement the course's programming assignments, whereas the other group was devoid of access to ChatGPT. Static code analysis tools such as Checkstyle [9] and PMD [10] have also been used in previous studies to ensure code in a project conforms to a defined coding standard style and to identify flaws that could manifest themselves as bugs.

The motivation behind this study is to provide insights into the implications of ChatGPT on code quality, particularly in an educational setting. Our findings may help educators tailor teaching materials to promote students' awareness of code quality and code understandability when using ChatGPT. To investigate the effect of ChatGPT onto code quality and code understandability among students in an introductory Java course, we formulated the following three research questions:

- **RQ1.** What violations to coding conventions [11] can be observed in the source code if students use or not use ChatGPT for the course's programming exercises?
- **RQ2.** What are the most violation-dense topics in the programming exercises if students use or not use ChatGPT for implementing them?
- **RQ3.** How does using ChatGPT influence the cyclomatic and cognitive complexity of the students' programming exercises' source code?

This paper is organized as follows: Section II delves into the related studies, while Section III outlines the methodology employed in our research. Afterwards, in Section IV, the results of our study are presented and specifically their implications for programming education discussed in Section V. Consequently, in Section VI, the potential threats to the validity of our study are addressed, before concluding our work and suggesting avenues for future research in Section VII.

II. RELATED WORK

We identified four streams of related research. The first stream involves investigating the quality of ChatGPT-generated code. In a study conducted by Li et al. [12], the authors examined the differences between human-authored code and code generated by ChatGPT. They developed a discriminative feature set and a dataset cleansing technique to differentiate between the two sources and achieved high accuracy in distinguishing ChatGPT-generated code from human-authored code. In another study by Guo et al. [13], the potential of ChatGPT in automating the code review process and improving the quality of student-generated code was evaluated. The results showed that ChatGPT outperforms *CodeReviewer* in code refinement tasks, but the authors also identified challenges and limitations, such as refining documentation and functionalities due to a lack of domain knowledge, unclear location, and unclear changes in the review comments. Liu et al. [14] systematically studied the quality of ChatGPT-generated code and found that although it can generate correct code for a significant number of tasks, there are still issues with wrong outputs, compilation or runtime errors, and maintainability. The study also revealed that ChatGPT's performance drops significantly on new programming tasks. The authors concluded that ChatGPT can partially address these challenges and improve code quality by more than 20%. Idrisov and Schlippe [15] evaluated the performance of generative AIs, including ChatGPT, BingAI Chat, GitHub Copilot, StarCoder, Code Llama, CodeWhisperer, and InstructCodeT5+, for six LeetCode problems of varying difficulty in Java, Python, and C++ based on metrics such as correctness, efficiency, and maintainability. Their analysis revealed that Github Copilot generated the most accurate Java and C++ code, while BingAI excelled in Python. The authors concluded that AI-generated code can be beneficial but often requires minor corrections, and AI tools can still significantly speed up the coding process by providing a solid starting point.

Studies examining the code quality of Github Copilot are in the center of the second stream of research. Nguyen and Nadi [16] assessed GitHub Copilot's code recommendations' accuracy and comprehensibility across different programming languages using LeetCode questions. They evaluated Copilot's suggestions in Python, Java, JavaScript, and C, measuring accuracy with LeetCode test cases and comprehensibility with SonarQube's complexity metrics. Results indicated Java had the highest accuracy at 57%, while JavaScript had the lowest at 27%. Although Copilot's code was generally comprehensible and low in complexity, issues included generating redundant code and relying on undefined helper methods. The study's limitations, such as a small sample size and Copilot's closed-source nature, suggest the need for further research under varied conditions. Yetistiren et al. [17] used the HumanEval dataset with 164 coding problems to evaluate GitHub Copilot's code generation based on validity, correctness, and efficiency. They found that 91.5% of the generated code was valid, 28.7% correct, and 51.2% partially correct, with efficiency comparable to human-written code in 87.2% of cases. Removing docstrings reduced correctness from 28.7% to 19.5%. The study concluded that while Copilot can generate valid and partially correct code, its accuracy is significantly influenced by input quality, such as docstrings and function names. The research highlights the need for further improvements in AI code generation tools for better accuracy and reliability.

The third stream of research examines the code quality of novice programmers. Studies by Izu and Mirolo [4] and Börstler et al. [18] revealed disparities in the definition and prioritization of code quality among novice programmers, with aspects such as performance, structure, conciseness, and comprehensibility being evaluated differently. Izu and

Mirolo [4] explored novice computer science students' perceptions of code quality, finding that students prioritize performance, structure, conciseness, and comprehensibility. The authors suggested that teaching should encourage discussion of best practices and personal preferences, rather than simply proposing a set of rules. Börstler et al. [18] investigated perceptions of code quality among students, educators, and professional developers. The authors observed that there was no common definition of code quality among the groups, and that readability was the most frequently named indicator of code quality.

Several studies have investigated the nature of common coding mistakes and quality issues among novice programmers. Brown et al. [19], Karnalim et al. [20], and Keuning et al. [21], [22] provide insights into the frequent types of errors and code quality problems students encounter, including neglect of coding conventions and difficulties with program flow and modularization. These studies collectively underscore the importance of targeted instruction and feedback in helping students overcome these challenges and highlight the potential role of automated tools in educating students about code quality issues. Östlund et al. [5] and Brown and Altadmri [23] point to the impact of teaching assistants and changing instructional practices on improvements in code quality. However, Breuker et al. [3] revealed that despite these efforts, there is no significant difference in code quality between first-year and second-year students, suggesting a plateau in learning that needs to be addressed through more effective teaching strategies. Brown and Altadmri [23] conducted a study to determine the frequency of common mistakes made by students learning Java programming and to assess the accuracy of educators' estimations of these mistakes. By analyzing data from nearly 100 million compilations across over 10 million programming sessions from novice Java programmers, as well as a survey among educators, the researchers discovered a discrepancy between the estimates of educators and the actual mistakes made by students. This finding implies that educators may need to enhance their understanding of common students' mistakes to improve their teaching effectiveness.

The last stream of related research explores the potential of LLM-based chatbots such as ChatGPT in programming education. In a recent study, Kazemitabaar et al. [24] examined the suitability of *OpenAI Codex* on supporting introductory programming for novice programmers. The research aimed to determine whether novices could comprehend, modify, or extend code generated by these tools without developing a reliance on technology. The study, which involved young learners with no prior experience in text-based programming, found that LLM-based code generators significantly improved code-authoring performance, suggesting that such tools do not necessarily result in reliance on technology for coding tasks. Jacques [25] investigated the influence of ChatGPT on the quality of student-generated code in an introductory course in computer science. Using a qualitative approach, Jacques demonstrated how these tools could enhance students' understanding of coding concepts and critical thinking skills by providing multiple solutions to problems, thereby increasing their engagement with the programming language. The work concluded that LLM-based coding tools could serve as valuable resources in improving programmingStudents' experiences of using ChatGPT in an undergraduate programming course education. Daun and Brings [26] identified both potential benefits and drawbacks of using generative AIs chatbots ChatGPT in software engineering education, noting that while ChatGPT performed well in providing answers to software engineering questions and literature references, its unsupervised use could be detrimental. They recommended supervised integration of these tools in educational practices to leverage their benefits while mitigating potential risks. Ali et al. [27] conducted a study in an educational context using quantitative testing on 460 Google-certified Python problems to evaluate the performance of various LLMs, including GPT-3.5, GPT-4, Claude, Bard, and Bing. The objective was to assess the efficiency and challenges of ChatGPT in code generation tasks. Results showed GPT-4 with the highest score at 87.51%, followed by GPT-3.5 at 83.18%, and Bing at 81.96%, while Claude and Bard scored 71.43% and 76.16%, respectively. GPT-based models generated more concise and efficient code. The study concluded that GPT-4 is the most efficient coding assistant but noted that LLMs require human feedback to ensure accuracy and face compatibility issues with existing code. Chen et al. [28] employed a mixed-method approach combining surveys, interviews, and performance metrics to evaluate ChatGPT's efficacy in higher education. The study aimed to assess ChatGPT's benefits and drawbacks in enhancing learning and teaching. Findings showed that ChatGPT provided instant feedback and explanations, boosting student engagement and understanding. However, issues such as information accuracy and potential over-reliance on AI, which might hinder critical thinking development, were noted. The study concluded that ChatGPT offers valuable educational support but should be integrated with caution to avoid negative impacts. Limitations included a small sample size, short intervention duration, and a focus on higher education, limiting generalizability to other educational levels. Hartley et al. [29] assessed ChatGPT's role in supporting independent coding learning, examining its effectiveness in delivering instructional materials, feedback, and planning. Through an evaluative case study, they queried ChatGPT on self-regulated programming learning and analyzed the responses. Findings indicated that ChatGPT provided comprehensive, personalized guidance on programming concepts, integrated multimodal information, and offered detailed planning schedules. However, it lacked interactivity and assessment capabilities. The study concluded that ChatGPT holds significant potential for personalized learning akin to one-on-one tutoring, contingent on learners' metacognitive skills. Limitations

included ChatGPT's current capabilities, focus on Python programming, and the scope of the analysis. Rahman and Watanobe [30] examined the use of ChatGPT in programming education and research, conducting coding experiments to assess its capabilities in code generation, pseudocode creation, and code correction, validated by an online judge system. Surveys revealed that students and teachers found ChatGPT beneficial for personalized feedback and solutions in programming education. However, concerns arose about content accuracy and over-reliance on AI, which might impair critical thinking. The study concluded that ChatGPT can enhance programming education but must be carefully integrated to prevent issues like diminished critical thinking and cheating. Prather et al. investigated the effects of generative AI tools on novice programmers, using a lab study with 21 participants. The study observed students' interactions with tools like GitHub Copilot and ChatGPT during a programming task. While some students benefitted from these tools by accelerating their learning, others faced persistent metacognitive challenges, such as misunderstanding problems and overestimating their skills. Generative AI occasionally worsened these issues or introduced new ones, like reliance on incorrect suggestions. Limitations include a small sample size and focus on specific AI tools, potentially affecting the generalizability of the findings.

III. METHODOLOGY

We conducted our study with two separate groups of part-time undergraduate students in an introductory Java programming courses. Each group's course lasted five weeks and followed a Python programming course. The students, who had no prior experience with Java, were expected to attend on-campus lectures, complete programming exercises at home, and pass a written exam at the end of the course. The programming exercises, which were identical in both groups, covered a wide range of Java programming concepts, including fundamentals, loops, object-oriented programming, interfaces, collections, file handling, lambda expressions, and multithreading. Both courses also emphasized the importance of code quality, i.e., the adherence to coding conventions, by encompassing regular training sessions designed to instruct students on avoiding code smells. The lecturer provided individualized feedback and graded the students' programming exercises, which were submitted on a weekly basis through the university's online teaching system. Although not mandatory, submission of programming exercises was strongly encouraged to enable students to benefit from the lecturer's feedback. For a positive grade of an exercises it was necessary that its code compiled. Uncompilable code resulted in zero points. Incorrect implementation of the programming task resulted in point deductions.

The first group of student (n = 16, control group) attended the course during the 2022 summer term, thus before the public availability of ChatGPT. The second group of students (n = 22, treatment group) attended it during the 2023 summer term. In this course, students could choose for each programming exercise whether they wanted to use ChatGPT. The decision to use or not use ChatGPT needed to be given during submission of their programming exercise. Table 1 shows the number of exercise submissions among the two groups in chronological sequence of the exercises. In the treatment group, with 63.2% GPT-4 was the predominant version used compared to 36.8% for GPT-3.5. All students used the free version of ChatGPT without subscription.

TABLE 1. Students' submissions of programming exercises.

		Control Group	Treatment Group	
#	Exercise Focus		●	○
Fu	Fundamentals	16	19	3
Lo	Loops	15	19	3
OO	Object-Oriented Programming	15	19	3
In	Interfaces, Exception Handling	15	19	3
Co	Collections	15	15	6
Fi	File, IO Streams	14	16	3
La	Lambda Expressions, Multithreading	13	12	6

● Students using ChatGPT, ○ Students not using ChatGPT

A. STATIC CODE ANALYSIS

We used *Checkstyle* [9] to perform static analysis of the source code submitted by students and employed a predefined ruleset [32], [33] to detect violations of "Oracle's Code Conventions for Java" [11]. The ruleset encompasses rules specifying the correct implementation of established coding conventions. These data were utilized to answer the first two research questions (RQ1 and RQ2).

Furthermore, to address the third research question (RQ3), we also calculated the cyclomatic and cognitive complexity of the submitted exercise source code. *Cyclomatic complexity* [34] is a quantitative measure that assesses the number of linearly independent paths through a program's source code, reflecting its complexity and potential for modification. It is often employed to predict a program's maintainability, with higher values indicating more intricate code that could be harder to understand and modify. We determined this metric using *Checkstyle*.

Similarly, *cognitive complexity* [35] is a quantitative measure that evaluates the ease of comprehending code, diverging from cyclomatic complexity by focusing on the mental effort required to comprehend code rather than just control flow complexity. Its calculation involves evaluating elements such as the depth of nesting in control structures, the complexity added by logical operators, and the presence of multiple conditions within statements, all contributing to the overall score. This metric was determined using the *SonarQube* platform [36] for static code analysis.

B. STATISTICAL TESTS

Upon an initial assessment of the data, we discovered that the distribution of rule violations and complexity measures

TABLE 2. Definitions of the checked coding rules [31].

Rule	Definition
LineLength	Checks for long code lines.
FinalParameters	Checks that parameters for methods, constructors, catch and for-each blocks are marked `final`.
HiddenField	Checks that a local variable or a parameter does not shadow a field that is defined in the same class.
MissingSwitchDefault	Checks that each `switch` statement has a `default` clause.
DesignForExtension	Ensures that classes are structured in a way that facilitates their extension through subclassing.
MagicNumber	Checks that there are no "magic numbers", i.e., a numeric literal not defined as a constant.
VisibilityModifier	Enforces encapsulation by requiring that public class members must be either `static final`, immutable or annotated by specific annotations.
RightCurly	Checks the placement of right curly braces ('}') for code blocks.
NeedBraces	Checks for braces around code blocks.
LocalVariableName	Checks that local, non-final variable names conform to Java naming conventions.
MethodParamPad	Evaluates the spacing between method and constructor definitions, calls, and invocations, and the opening parentheses of parameter lists.
AvoidNestedBlocks	Identifies instances where blocks of code are nested within other blocks.

was right-skewed, which suggested a concentration of lower values with a long tail extending towards higher values. This skewness indicated that a considerable number of students had fewer violations, while a smaller subset had a higher number of violations. As the distribution of the data was non-normal, traditional parametric tests, which presume normality of the data distribution, were deemed inappropriate for statistical analysis of the data. Parametric tests rely on the assumption that the data are normally distributed within each group being compared. However, the right-skewed nature of our data violated this assumption, which could lead to erroneous conclusions if such tests were applied.

To address this issue, we opted for the *Wilcoxon Rank Sum Test* with continuity correction for our comparative analysis between the two groups. This non-parametric test does not assume normal distribution of the data and is particularly well-suited for analyzing differences in median values between two independent groups when the data are skewed. It evaluates whether the distribution of rule violations or complexity measures significantly differs between students who used ChatGPT for the course' programming exercises (treatment group) and those who did not (control group). The statistical significance was set to $p \leq 0.005$ and we formulated the following null hypotheses $H_{1\text{-}2,0}$ for research questions RQ1 and RQ2:

There is no correlation between the usage of ChatGPT by students to implement programming exercises and

- $H_{1,0}$: *the number of rule violations (to code conventions).*
- $H_{2.1,0}$: *the cyclomatic complexity of the source code.*
- $H_{2.2,0}$: *the cognitive complexity of the source code.*

As a result, the corresponding alternative hypotheses $H_{1\text{-}2,a}$ can be accepted with a *p-value* of less than 0.05, indicating that there is a statistically significant correlation between the usage of ChatGPT by students to implement programming exercises and rule violations ($H_{1,a}$), and cyclomatic or cognitive code complexity ($H_{2.1,a}$ and $H_{2.2,a}$ respectively).

IV. RESULTS

The following section outlines the findings of our study in sequential manner with the research questions.

A. MOST FREQUENT RULE VIOLATIONS (RQ1)

In our initial analysis, we assessed the quality and distribution of the acquired rule violations and complexity measures. Subsequently, we normalized the data to account for the varying number of students in the control and treatment groups, which was essential for maintaining the integrity of our study. After preparing the data, we calculated the key statistical properties for both groups. We focused on assessing the likelihood of observing particular rule violations under the null hypothesis $H_{1,0}$ (see Section III), which assumes no influence of ChatGPT on the probability of specific rule violations. If the *p-value* of a rule violation is smaller than or equal 0.005, the null hypothesis $H_{1,0}$ for this rule can be rejected, and the alternative hypothesis $H_{1,a}$ can be embraced. This hypothesis signifies that the use of ChatGPT has an effect on the likelihood of violations of this rule.

We ranked the rules in ascending order of their *p-value*, i.e., with decreasing statistical significance. To focus our discussion, we concentrate on the top 12 rule violations with the highest significance. The results from these calculations are presented in Table 3. Also, we give the Pearson correlation coefficient between the groups for each rule. An asterisk (*) in the first column of the table marks rule violations that have been identified as statistically relevant, i.e., for which the alternative hypothesis $H_{1,a}$ is applicable. Figure 1 shows the rule violations for each programming exercise among the two groups.

The treatment group demonstrated greater adherence to rules *LineLength*, *FinalParameters*, *HiddenField*, *MissingSwitchDefault*, *DesignForExtension*, *MagicNumber*,

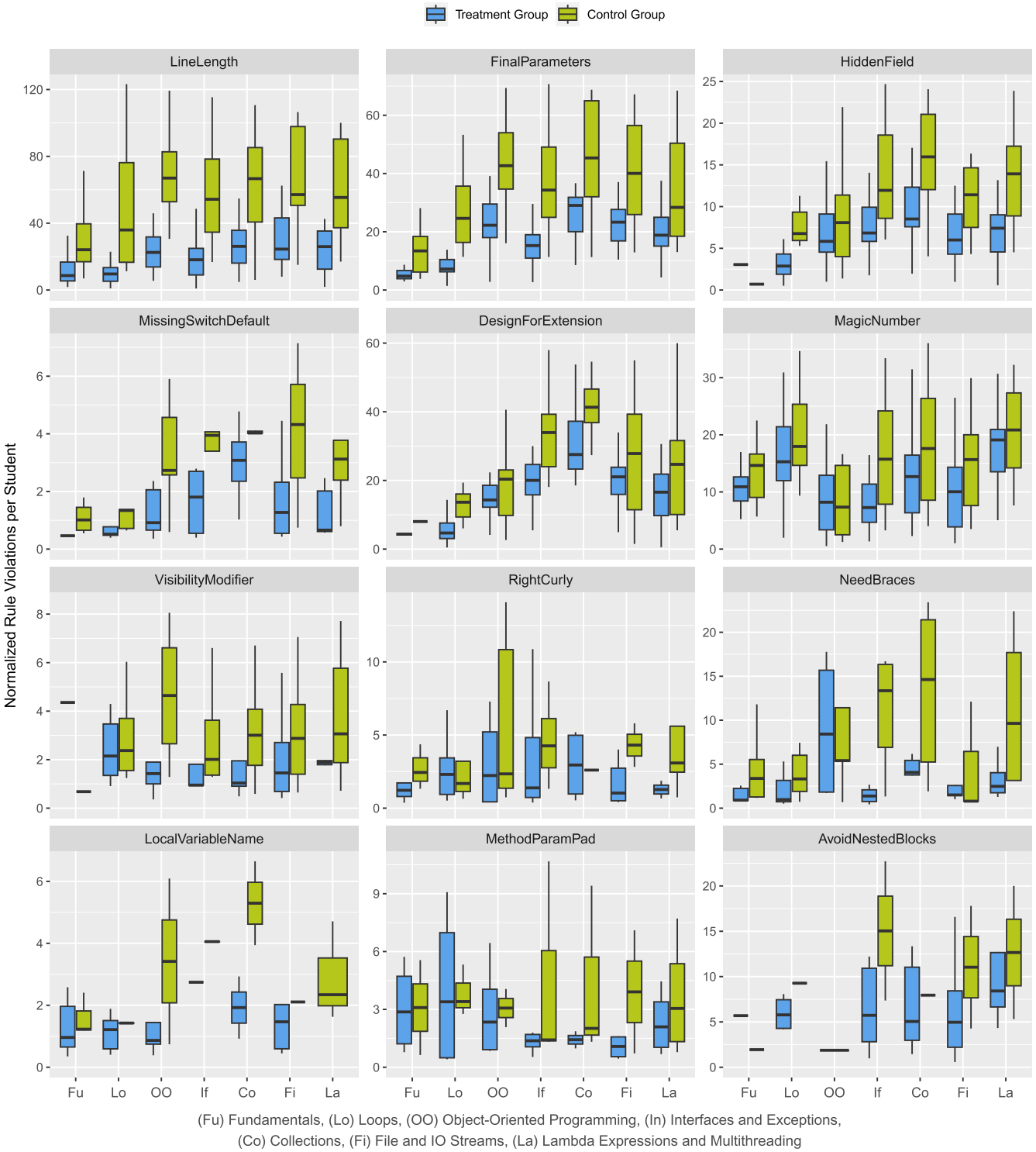


FIGURE 1. Most frequent rule violations across the different programming exercises. The treatment group (blue) shows students using ChatGPT, the control group (green) students not using ChatGPT.

VisibilityModifier, *RightCurly*, *NeedBraces*, *LocalVariableName* across all programming exercises with statistical significance. In the treatment group rules *LineLength* and *FinalParameters* had even less than half of median violations of the control group across all exercises. However, the tests of rules *MethodParamPad* and *AvoidNestedBlocks* did not produce statistically significant results.

TABLE 3. Static code measures, *p-values* and Pearson correlation for $H_{1,0}$ (rule violations ~ ChatGPT usage).

	Rule	p-value	Correlation Coefficient	Control Group			Treatment Group		
				Std.Dev.	Median	Mean	Std.Dev.	Median	Mean
*	LineLength	< 0.005	0.0051	32.71	50.3	54.36	15.91	16.65	21.18
*	FinalParameters	< 0.005	0.0126	19.22	28.57	33.24	10.62	16.23	17.13
*	HiddenField	< 0.005	0.0275	6.24	9.91	11.61	3.96	6.32	6.81
*	MissingSwitchDefault	< 0.005	0.358	2.02	2.80	3.13	1.43	0.92	1.59
*	DesignForExtension	< 0.005	0.0118	14.95	23.51	25.19	10.94	18.10	17.54
*	MagicNumber	< 0.005	0.0254	9.48	15.42	16.28	7.46	11.42	12.05
*	VisibilityModifier	< 0.005	0.146	2.20	2.97	3.49	2.02	1.69	2.28
*	RightCurly	< 0.005	0.188	3.92	2.74	4.05	2.81	1.55	2.59
*	NeedBraces	< 0.005	0.152	7.01	5.37	7.99	3.91	2.69	3.83
*	LocalVariableName	< 0.005	0.0798	1.88	2.37	2.90	1.33	1.03	1.60
	MethodParamPad	0.057	0.162	3.07	2.66	3.79	2.27	1.31	2.41
	AvoidNestedBlocks	0.154	0.0332	7.49	8.07	10.76	5.19	5.57	6.85
	Complexity Measure								
*	CyclomaticComplexity	< 0.005	0.0232	8.67	9.17	12.2	4.80	7.27	8.82
*	CognitiveComplexity	< 0.005	0.0002	10.4	1.50	3.30	6.78	1.34	2.88

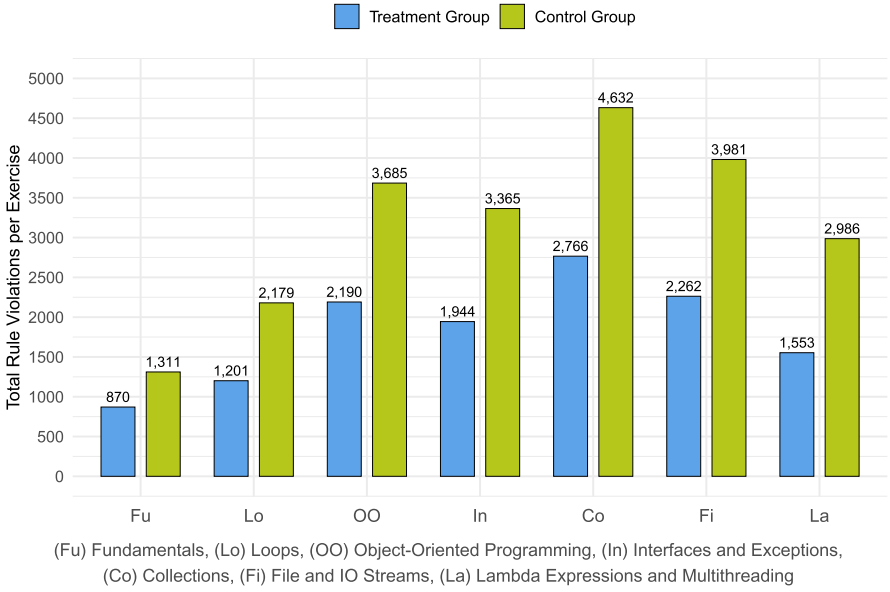


FIGURE 2. Distribution of rule violations across the different programming exercises. The treatment group (blue) shows students using ChatGPT, the control group (green) students not using ChatGPT.

B. MOST VIOLATION-DENSE TOPICS (RQ2)

Figure 2 shows the total number of rule violations per exercise and group. For the sake of brevity, we only present the top three exercises with the highest frequency of rule violations and the leading five rules that were violated most frequently across all exercises for each group. We identified *"Collections"*, *"File IO and Streams"*, and *"Object-Oriented Programming"* as the exercises with the highest number of rule violations for both groups.

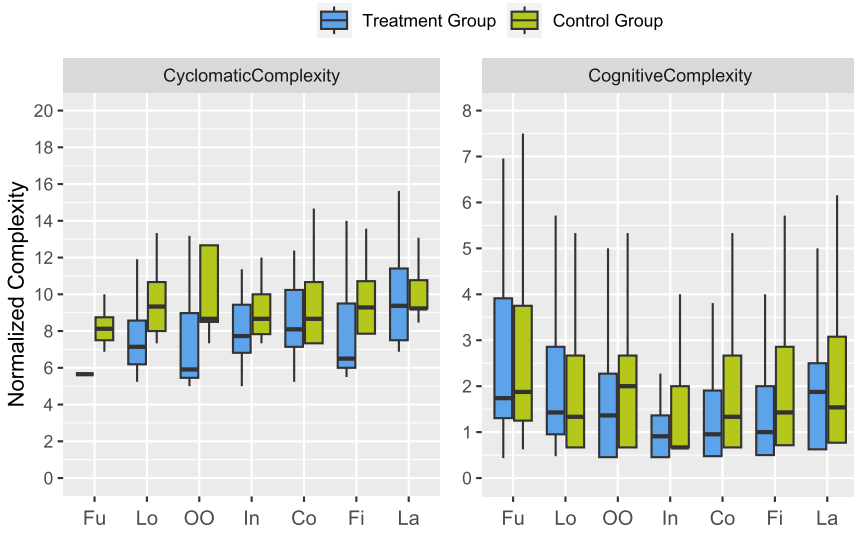
The concrete number of rule violations for each exercise and the top three most violated rules among control and treatment group are given in Table 4. As can be depicted from this table, exercises *"Collections"* (Co), *"File IO and Streams"* (Fi), and *"Object-Oriented Programming"* (OO) are the most violation-dense exercises and rules *LineLength*, *FinalParameters*, and *DesignForExtension* are the most frequently violated rules among both groups. Also, the Pearson correlation coefficient shows a differently manifestations of linear correlation with the highest being for *MissingSwitchDefault*, *RightCurly*, and *NeedBraces*.

C. INFLUENCE ON CYCLOMATIC AND COGNITIVE COMPLEXITY (RQ3)

Figure 3 shows the distribution of cyclomatic and cognitive complexity of the students' implementation of the programming exercises. The statistical analysis concerning

TABLE 4. Total number and mean ($\bar{x}$) of rule violations among students not using ChatGPT (control group) and students using ChatGPT (treatment group) per exercise in descending order.

Exercise	Total Violations		$\bar{x}$	Rule	Violation/Rule	
	Control Group	Treatment Group			Control Group	Treatment Group
Co	4,632	2,766	3,699	LineLength	563	560
				FinalParameters	494	546
				DesignForExtension	455	597
Fi	3,981	2,262	3,121.5	LineLength	587	628
				FinalParameters	490	458
				DesignForExtension	349	397
OO	3,685	2,190	2,937.5	LineLength	845	557
				FinalParameters	555	511
				DesignForExtension	221	289
In	3,365	1,944	2,654.5	LineLength	684	445
				FinalParameters	521	344
				DesignForExtension	424	397
La	2,986	1,553	2,269.5	LineLength	659	385
				FinalParameters	383	315
				DesignForExtension	316	286
Lo	2,179	1,201	1,690	LineLength	725	213
				FinalParameters	400	181
				DesignForExtension	259	344
Fu	1,311	870	1,090.5	LineLength	504	283
				FinalParameters	217	247
				DesignForExtension	215	127



(Fu) Fundamentals, (Lo) Loops, (OO) Object-Oriented Programming, (In) Interfaces and Exceptions, (Co) Collections, (Fi) File and IO Streams, (La) Lambda Expressions and Multithreading

FIGURE 3. Complexity measures of students' code when using (treatment group) or not using ChatGPT for implementing programming exercises (control group).

the cyclomatic complexity of the students' code showed a *p-value* below 0.005, indicating a significant difference and thus supporting the alternative hypothesis $H_{3.1,a}$. This notable discrepancy underscores a trend to lower cyclomatic complexity of the code in the treatment group, evident across various parts of the exercise. For instance, in

"Object-Oriented Programming" exercise, the treatment group reported a median complexity of 5.91, substantially lower than the control group's 8.67. Such patterns are consistent across all exercises, with the treatment group frequently demonstrating lower median complexity measures. This suggests that ChatGPT potentially aids in producing code with reduced cyclomatic complexity.

Similarly, the statistical analysis of the cognitive complexity revealed a *p-value* below 0.005, thus supporting the alternative hypothesis $H_{3.2,a}$. This significant finding indicates lower cognitive complexity in the treatment group across various programming exercises. For instance, in the *"Object-Oriented Programming"* exercise, the treatment group's median complexity was 1.36, compared to 2 in the control group. This trend of lower median complexity values in the treatment group suggests that ChatGPT slightly assists students in writing code that is easier to understand. However, given the only small difference in the median values we regard this effect as rather negligible and requiring further research with larger student populations for further evaluation. Finally, the Pearson correlation coefficient shows slight linear correlation among the groups for both complexity metrics.

V. DISCUSSION

The results of our study indicate a consistent pattern, wherein the treatment group that utilized ChatGPT displayed significantly fewer violations across several Java programming rules compared to the control group. This pattern was particularly evident in rules pertaining to code structure and syntax, such as *LineLength*, *FinalParameters*, and *MissingSwitchDefault*. This aligns with findings from previous studies, such as those by Guo et al. [13] and Liu et al. [14], which demonstrated ChatGPT's capability to improve code quality and perform well in code refinement tasks, despite certain limitations.

One of the most noteworthy observations was the treatment group's ability to more closely adhere to Java programming best practices, as evidenced by their lower median violations in rules like *DesignForExtension* and *VisibilityModifier*. These results suggest that incorporating ChatGPT into programming education can not only assist students in writing syntactically correct code but also in understanding and applying good design principles, which are essential for creating maintainable and scalable software. This supports the findings of Jacques [25] and Daun and Brings [26], who found that LLM-based tools can enhance students' understanding of coding concepts and design principles.

Furthermore, the reduction in cyclomatic and cognitive complexities in the treatment group's code underscores another vital educational benefit. Lower complexity measures often correlate with simpler, more readable code, which not only reduces the cognitive load on programmers but also potentially decreases the likelihood of bugs and errors. This finding is crucial for educators as it emphasizes the potential of tools like ChatGPT to enhance students' ability to write efficient, understandable, and less complex code, thereby improving their overall coding proficiency. This observation is in line with research by Hartley et al. [29], which highlighted ChatGPT's role in providing comprehensive guidance on programming concepts, leading to more efficient and readable code.

Our research further revealed that violations of specific coding rules, including *MethodParamPad* and *AvoidNestedBlocks*, did not result in statistically significant differences between the groups. This suggests that while ChatGPT can be beneficial for many coding aspects, there may be nuances in coding style and structure that may not be adequately addressed if students rely solely on assistance from an LLM like ChatGPT. This aligns with findings from Chen et al. [28] and Rahman and Watanobe [30], which highlighted potential drawbacks of over-reliance on AI tools, such as reduced critical thinking and the risk of not developing independent problem-solving skills.

The trend of lower median complexity values in the treatment group suggests that ChatGPT slightly assists students in writing code that is easier to understand. However, given the only small difference in the median values, we regard this effect as rather negligible and requiring further research with larger student populations for further evaluation. This cautionary note is consistent with findings by Idrisov and Schlippe [15] and Ali et al. [27], who noted that while AI tools can be beneficial, their impact varies and must be evaluated in broader contexts.

Given the limited sample size, these results should be interpreted cautiously. To delve deeper into the underlying factors contributing to these disparities, additional research with larger sample sizes is necessary. This echoes the sentiments of the study of Nguyen and Nadi [16], who highlighted the need for further research to validate findings under varied conditions and larger datasets.

VI. THREATS TO VALIDITY

We acknowledge several potential threats to the validity of our study that could impact the interpretation and generalizability of our findings.

Regarding **internal validity**, one major concern is the level of experience of the participants. Given that the students were novices with no prior experience in Java programming, their learning curve over the course could affect their ability to adhere to coding conventions and manage code complexity, regardless of the assistance provided by ChatGPT. Additionally, the individualized feedback provided by the lecturer may have varied in its effectiveness across students, potentially confounding results related to improvements in code quality.

In terms of **external validity**, the small sample size poses a significant threat, limiting the generalizability of the results to a broader population of programming students. Moreover, the specific educational context of part-time undergraduate students may not represent the diverse backgrounds and educational settings in which programming is taught,

affecting the applicability of the findings to other learning environments.

The measurement of code quality through static code analysis tools and predefined rulesets to detect violations may raise concerns about **construct validity**. The reliance on these tools and rulesets to quantify code quality and complexity may not fully capture the nuances of what constitutes high-quality, maintainable code, potentially overlooking aspects of code quality not encompassed by static code analysis per se.

Similarly, concerns have been expressed about **statistical conclusion validity** of the non-parametric *Wilcoxon Rank Sum Test*, which is often used in non-normal distributions. Given the right-skewed distribution of rule violations and complexity measures, there is a risk that the test may not have the power to detect subtle differences between groups due to outliers or the distribution shape.

VII. CONCLUSION

Our research systematically investigated the impact of ChatGPT on code quality among undergraduate students through a detailed analysis of rule violations and complexity metrics. Subsequently, we present the key findings in alignment with our research questions.

- **Most Frequent Rule Violations:** Our investigation revealed statistically significant differences in rule adherence between students using ChatGPT (treatment group) and those not using it (control group). Specifically, the treatment group showed fewer violations across several key coding conventions. Notably, rules verifying the length of code lines (*LineLength*), the declaration of parameters for methods, constructors, catch blocks, and for-each loops as `final` (*FinalParameters*) or verifying code extensibility (*DesignForExtension*) showed marked improvements, with $p\text{-values} < 0.005$, indicating a statistically significant difference in adherence between the two groups.
- **Most Violation-Dense Topics**: We observed exercises related to *"Collections"*, *"File IO and Streams"*, and *"Object-Oriented Programming"* as topics with the highest incidence of rule violations. Despite a significant overall reduction in violations among the treatment group, these areas remained challenging, underscoring the need for targeted educational interventions. The rules *LineLength* and *FinalParameters* were among the most frequently violated, highlighting specific areas where ChatGPT's guidance notably improved student performance.
- **Influence on Cyclomatic and Cognitive Complexity:** Our study further explored the cyclomatic and cognitive complexity of student submissions. We observed that code from the treatment group displayed lower complexity levels across both metrics, notably in cyclomatic complexity ($p\text{-value} < 0.005$). These findings imply that ChatGPT contributes not merely to simplifying code complexity but also to enhancing code comprehensibility, as evidenced by the statistically significant difference in cognitive complexity ($p\text{-value} < 0.005$).

The outcomes of our study offer novel insights into the impact of ChatGPT on code quality among novice programming students. Nevertheless, it is crucial to exercise caution when interpreting the results, taking into account the identified threats to validity. Expanding on these findings requires future research involving larger, more diverse samples, considering various educational contexts and more refined measures of code quality and ChatGPT usage. In addition, it is essential to explore the experiences of a larger number of students, possibly also assessing ChatGPT's suitability for supporting students in programming courses beyond Java. A qualitative analysis of the students' prompts to ChatGPT could help examine the structure and quality of their input, enabling educators to better instruct students on how to tailor their input to ChatGPT to improve code quality.

REFERENCES

[1] OpenAI. *OpenAI*. Accessed: Aug. 15, 2024. [Online]. Available: https://openai.com/

[2] I. Ozkaya, "Application of large language models to software engineering tasks: Opportunities, risks, and implications," *IEEE Softw.*, vol. 40, no. 3, pp. 4–8, May 2023.

[3] D. M. Breuker, J. Derriks, and J. Brunekreef, "Measuring static quality of student code," in *Proc. 16th Annu. Joint Conf. Innov. Technol. Comput. Sci. Educ.*, Jun. 2011, pp. 13–17.

[4] C. Izu and C. Mirolo, "Exploring CS1 student's notions of code quality," in *Proc. Conf. Innov. Technol. Comput. Sci. Educ.*, Jun. 2023, pp. 12–18.

[5] L. Östlund, N. Wicklund, and R. Glassey, "It's never too early to learn about code quality: A longitudinal study of code quality in first-year computer science students," in *Proc. 54th ACM Tech. Symp. Comput. Sci. Educ. V.1*, Mar. 2023, pp. 792–798.

[6] N. C. C. Brown and A. Altadmri, "Investigating novice programming mistakes: Educator beliefs vs. student data," in *Proc. 10th Annu. Conf. Int. Comput. Educ. Res*, Jul. 2014, pp. 43–50.

[7] G. De Ruvo, E. Tempero, A. Luxton-Reilly, G. B. Rowe, and N. Giacaman, "Understanding semantic style by analysing student code," in *Proc. 20th Australas. Comput. Educ. Conf*, Jan. 2018, pp. 73–82.

[8] P. Haindl and G. Weinberger, "Students' experiences of using ChatGPT in an undergraduate programming course," *IEEE Access*, vol. 12, pp. 43519–43529, 2024.

[9] *Checkstyle—Checkstyle 10.14.1*. Accessed: Aug. 15, 2024. [Online]. Available: https://checkstyle.sourceforge.io/

[10] (2024). *PMD*. [Online]. Available: https://pmd.github.io/

[11] Oracle. *Code Conventions for the Java Programming Language: Contents*. Accessed: Aug. 15, 2024. [Online]. Available: https://www.oracle.com/java/technologies/

[12] K. Li, S. Hong, C. Fu, Y. Zhang, and M. Liu, "Discriminating human-authored from ChatGPT-generated code via discernable feature analysis," in *Proc. IEEE 34th Int. Symp. Softw. Rel. Eng. Workshops (ISSREW)*, Oct. 2023, pp. 120–127.

[13] Q. Guo, J. Cao, X. Xie, S. Liu, X. Li, B. Chen, and X. Peng, "Exploring the potential of ChatGPT in automated code refinement: An empirical study," in *Proc. IEEE/ACM 46th Int. Conf. Softw. Eng.*, Feb. 2024, pp. 1–13.

[14] Y. Liu, T. Le-Cong, R. Widyasari, C. Tantithamthavorn, L. Li, X.-B.-D. Le, and D. Lo, "Refining ChatGPT-generated code: Characterizing and mitigating code quality issues," *ACM Trans. Softw. Eng. Methodol.*, vol. 33, no. 5, pp. 1–26, Jun. 2024.

[15] B. Idrisov and T. Schlippe, "Program code generation with generative AIs," *Algorithms*, vol. 17, no. 2, p. 62, Jan. 2024.

[16] N. Nguyen and S. Nadi, "An empirical evaluation of GitHub copilot's code suggestions," in *Proc. IEEE/ACM 19th Int. Conf. Mining Softw. Repositories (MSR)*, May 2022, pp. 1–5.

[17] B. Yetistiren, I. Ozsoy, and E. Tuzun, "Assessing the quality of GitHub copilot's code generation," in *Proc. 18th Int. Conf. Predictive Models Data Analytics Softw. Eng.*, Nov. 2022, pp. 62–71.
[18] J. Brstler, H. Strrle, D. Toll, J. van Assema, R. Duran, S. Hooshangi, J. Jeuring, H. Keuning, C. Kleiner, and B. MacKellar, "'I know it when I see it' perceptions of code quality: ITiCSE'17 working group report," in *Proc. Conf. Work. Group Rep.*, Jan. 2018, pp. 70–85.
[19] N. C. C. Brown, P. Weill-Tessier, M. Sekula, A.-L. Costache, and M. Kölling, "Novice use of the Java programming language," *ACM Trans. Comput. Educ.*, vol. 23, no. 1, pp. 1–24, Mar. 2023.
[20] O. Karnalim, Simon, and W. Chivers, "Work-in-progress: Code quality issues of computing undergraduates," in *Proc. IEEE Global Eng. Educ. Conf. (EDUCON)*, Mar. 2022, pp. 1734–1736.
[21] H. Keuning, B. Heeren, and J. Jeuring, "Code quality issues in student programs," in *Proc. ACM Conf. Innov. Technol. Comput. Sci. Educ.*, Jun. 2017, pp. 110–115.
[22] H. Keuning, J. Jeuring, and B. Heeren, "A systematic mapping study of code quality in education," in *Proc. Conf. Innov. Technol. Comput. Sci. Educ. V.1*, Jun. 2023, pp. 5–11.
[23] N. C. C. Brown and A. Altadmri, "Novice Java programming mistakes: Large-scale data vs. Educator beliefs," *ACM Trans. Comput. Educ.*, vol. 17, no. 2, pp. 1–21, Jun. 2017.
[24] M. Kazemitabaar, J. Chow, C. K. T. Ma, B. J. Ericson, D. Weintrop, and T. Grossman, "Studying the effect of AI code generators on supporting novice learners in introductory programming," in *Proc. CHI Conf. Hum. Factors Comput. Syst.*, Apr. 2023, pp. 1–23.
[25] L. Jacques, "Teaching CS-101 at the dawn of ChatGPT," *ACM Inroads*, vol. 14, no. 2, pp. 40–46, Jun. 2023.
[26] M. Daun and J. Brings, "How ChatGPT will change software engineering education," in *Proc. Conf. Innov. Technol. Comput. Sci. Educ. V.1*, Jun. 2023, pp. 110–116.
[27] D. Ali, Y. Fatemi, E. Boskabadi, M. Nikfar, J. Ugwuoke, and H. Ali, "ChatGPT in teaching and learning: A systematic review," *Educ. Sci.*, vol. 14, no. 6, p. 643, Jun. 2024.
[28] J. Chen, Z. Zhuo, and J. Lin, "Does ChatGPT play a double-edged sword role in the field of higher education? An in-depth exploration of the factors affecting student performance," *Sustainability*, vol. 15, no. 24, p. 16928, Dec. 2023.
[29] K. Hartley, M. Hayak, and U. H. Ko, "Artificial intelligence supporting independent student learning: An evaluative case study of ChatGPT and learning to code," *Educ. Sci.*, vol. 14, no. 2, p. 120, Jan. 2024.
[30] M. M. Rahman and Y. Watanobe, "ChatGPT for education and research: Opportunities, threats, and strategies," *Appl. Sci.*, vol. 13, no. 9, p. 5783, May 2023.
[31] *Checkstyle—Checks*. Accessed: Aug. 15, 2024. [Online]. Available: https://checkstyle.sourceforge.io/checks.htm
[32] *Checkstyle—Sun's Java Style*. Accessed: Aug. 15, 2024. [Online]. Available: https://checkstyle.sourceforge.io/
[33] (2024). *Checkstyle/src/main/resources/sun.xml At Master*. [Online]. Available: https://github.com/checkstyle/checkstyle/
[34] T. J. McCabe, "A complexity measure," *IEEE Trans. Softw. Eng.*, vol. SE-2, no. 4, pp. 308–320, Jun. 1976.
[35] G. A. Campbell, "Cognitive complexity—An overview and evaluation," in *Proc. IEEE/ACM Int. Conf. Tech. Debt (TechDebt)*. New York, NY, USA: Association for Computing Machinery, May 2018, pp. 57–58.
[36] (2024). *Code Quality Tool & Secure Analysis With SonarQube*. [Online]. Available: https://www.sonarsource.com/products/sonarqube/

PHILIPP HAINDL received the Ph.D. degree in computer science from Johannes Kepler University Linz, in 2021. He is currently with St. Pölten University of Applied Sciences, as a Lecturer of software engineering, and has more than 15 years of practical experience in industrial and research-focused software projects, as a Software Engineer and an Architect. His research interests include empirical software engineering, software security, and software quality operationalization. He regularly functions as a reviewer of international software engineering conferences and journals.

GERALD WEINBERGER received the B.Sc. and Dipl.-Ing. (M.Sc. equivalent) degrees in IT security from St. Pölten University of Applied Sciences. He is currently a Lecturer with the Department of Informatics and Security, St. Pölten University of Applied Sciences.