Collaboration in Computer Science Education

What Is the Relationship Between Prior Programming Experience and Code Maintainability in Student Software Development Projects?

Egemen Yildiz[1]

Supervisors: Dr. ir. Fenia Aivaloglou[1]**, ir. Merel Steenbergen**[1]

[1] **EEMCS, Delft University of Technology, The Netherlands**

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 21, 2025

Name of the student: Egemen Yildiz
Final project course: CSE3000 Research Project
Thesis committee: Dr. ir. Fenia Aivaloglou, ir. Merel Steenbergen, Dr. ir. Mitchell Olsthoorn

Abstract

Group software projects are widely used in computer science education to teach collaboration and development skills, but students often enter with differing levels of programming experience. This study investigates how prior experience relates to code maintainability in a university-level group project course by analyzing four groups of five individuals each. Experience was measured using a weighted survey-based score, while maintainability was assessed using five metrics: code smells, comment density, code duplication, average file size, and naming quality. Although some metrics showed weaker trends, the results overall suggest that more experienced students tend to produce more maintainable code. These findings offer insights into how individual backgrounds influence collaborative software development in educational settings.

1 Introduction

In computer science education, group software projects are widely used to teach collaboration, software engineering practices, and real-world development skills. These projects often include students with different levels of programming experience, from beginners to those with several years of practice. Prior work has shown that group-based programming tasks can improve learning outcomes and engagement, but also introduce variation in code quality due to differing student backgrounds [3]. Despite this variation, it remains unclear how prior programming experience affects the quality of the code that students produce in such settings.

Code quality is a broad concept that includes aspects such as correctness, performance, and maintainability. This study focuses on maintainability, which refers to how easily code can be understood, modified, and extended by others [10]. Maintainability is especially important in group projects, where code is shared and updated frequently by multiple contributors. Unmaintainable code can slow development, introduce bugs, and reduce the learning value of the project for team members [16].

The goal of this research is to examine the relationship between students' programming experience and the maintainability of their code within the context of a university-level group software engineering project. Prior experience is measured through a structured survey from Feigenspan et al. [5], covering dimensions such as years of programming, language familiarity, and exposure to different paradigms. Maintainability is assessed using a set of widely adopted metrics, including code smells, comment density, code duplication, and file modularity, extracted via static analysis tools and grounded in the review by Ardito et al. [1]. Since the analysis is conducted on group project contributions rather than individual assignments, the results may also reflect the influence of team collaboration, shared code ownership, and group-level practices. These are important contextual factors to consider when interpreting the relationship between experience and maintainability.

This study addresses three research questions:

1. *Does programming experience correlate with code maintainability?*
2. *Which maintainability metric best reflects experience?*
3. *How does maintainability evolve during the project?*

To answer these, experience scores were computed using weighted survey responses, maintainability metrics were extracted per student using SonarQube and manual evaluation, and statistical analyses were used to explore trends and correlations.

Understanding how experience affects maintainability can inform how educators assign students to teams, interpret code quality differences, or structure instruction in project-based learning environments. If a clear link is found, it could justify differentiated support or assessment strategies based on experience. If not, it may point toward the need to emphasize collaborative practices or shared coding standards over individual experience in improving code quality.

2 Background and Related Work

Group-based software development projects are a widely adopted method in computer science education for teaching students essential software engineering practices such as modular design, documentation, and collaboration [3]. These projects simulate real-world engineering environments by requiring students to work in teams, manage shared repositories, and communicate design and implementation decisions effectively [4]. Prior work has highlighted that such collaborative settings often lead to disparities in code quality, influenced by differences in students' prior programming experience and familiarity with development tools [8].

A key dimension of code quality emphasized in both academia and industry is maintainability, defined as the ease with which code can be read, modified, or extended over time [1,10]. In educational group projects, maintainable code is especially important since team members frequently rely on and extend each other's contributions [3]. Clear structure, proper documentation, and modular design not only facilitate collaboration and learning but also mirror best practices in professional software engineering, where maintainability is critical for reducing long-term technical debt [16].

Programming experience has long been studied as a factor shaping students' performance, code comprehension, and development strategies. Feigenspan et al. and Siegmund et al. propose multidimensional models of experience that go beyond years coded, incorporating self-perception, tool familiarity, and exposure to different programming paradigms [5, 14]. Lopes et al.'s recent systematic review further confirms that more experienced developers generally write higher-quality code, although the strength and nature of this relationship vary across contexts [12]. However, much of this research is grounded in individual or professional settings, and it remains unclear whether these findings generalize to collaborative student environments where peer learning and shared ownership complicate the mapping between experience and quality outcomes.

This study addresses that gap by investigating how programming experience correlates with code maintainability in university-level group projects. It tests whether findings from professional contexts also hold in educational team settings. In doing so, it contributes to the growing literature on software engineering education by examining experience as a potential predictor of collaborative code quality in realistic learning environments.

3 Methodology

The research was conducted in the context of a university-level course where students developed a shared software project over several weeks, using version control and regular team and scrum meetings.

3.1 Participants and Data Sources

The participants in this study were second-year undergraduate students enrolled in the "CSE2000 Software Project" course at TU Delft. A total of 20 students took part, divided into 4 groups of five. Each group selected a real-world software development project from the university's "Software Project Portal" and contacted the corresponding client to initiate collaboration. Over a ten-week period, the students worked in teams using shared GitLab repositories for version control. Each student committed code using their own Git account, making it possible to track individual contributions through metadata such as author names and email addresses. By the end of the course, each group had produced a working project, typically consisting of around 10,000 lines of code and ready for deployment by the client.

Two types of data were collected for this research. First, each student completed a structured survey measuring their prior programming experience, including both objective and self-assessed components. Then, Git repository data was collected and processed to isolate each student's individual code contributions. This made it possible to extract maintainability metrics on a per-student basis. To protect privacy, all participants were assigned anonymized identifiers (1A, 2B etc.), which were used consistently across survey data and code attribution.

3.2 Measuring Programming Experience

The programming experience of students was assessed using a survey taken from Feigenspan et al. [5]. The survey covered multiple aspects of experience, including self-estimation, number of programming languages, years of coding, and exposure to paradigms like Object-oriented Programming (OOP) and logic programming.

To create a unified experience score, all responses were normalized and weighted. Furthermore, each item contributed to a weighted formula, based on its relevance to general programming fluency and its likely impact on coding practices such as modularity, readability, and code organization:

- **Years of programming (20%)**: This reflects long-term exposure to programming practice. Students who have coded for several years are more likely to have internalized fundamental principles. We assigned 20% weight because duration strongly correlates with fluency. It is normalized using `min(years, 10) / 10`, capping at 10 years to limit the effect of outliers.
- **Size of past projects (10%)**: Experience with larger codebases often requires more attention to structure and maintainability. This metric captures that dimension. We weighted it at 10% since not all students have access to large-scale projects, but it remains an important factor that can indicate experience. Options were mapped as: `N/A = 0`, `<900 LOC = 3`, `900 to 40000 LOC = 7`, `>40000 LOC = 10`.
- **Self-estimated experience (20%)**: Self-assessment on a scale of 1–10 captures students' confidence and perceived competence. Prior research shows strong correlation with actual performance. We assigned it 20% to balance its subjective but predictive nature. It was normalized as `value / 10`.
- **Number of known programming languages (15%)**: Knowing multiple languages suggests conceptual flexibility and broader exposure. We weighted this at 15% to reflect its relevance to adaptable coding habits. Normalized using `min(languages, 10) / 10`, with a cap at 10 languages.
- **OOP and logical paradigm experience (15% total)**: Experience with multiple paradigms indicates deeper structural understanding. We split this into two 7.5% components: one for object-oriented programming and one for logical programming, both rated on a 1–5 scale. Each was normalized as `value / 5`.
- **Number of programming courses taken (15%)**: Completing formal coding assignments builds structured knowledge and debugging habits. We weighted this 15% for its academic value. Since the number of courses that students could complete until the Software Project in TU Delft is 21, it is normalized as `min(courses, 21) / 21`.
- **Comparison to classmates (5%)**: This is a social reference point that may reflect informal confidence or teamwork dynamics. Although it adds insight into peer-relative experience, the highly subjective aspect of it makes its use limited. We gave it a modest 5% weight and normalized it as `value / 5`.

The aforementioned normalized values, then, were multiplied by their assigned weight and summed to produce a final experience score on a 0 to 100 scale. To support comparative analysis across experience levels, participants were divided into three groups based on the three-way distribution of the actual scores: lower, moderate, and higher experience. This grouping ensures balanced sample sizes and reflects the relative distribution within the studied population. The combined scoring method captures both objective and perceived dimensions of programming background and provides the foundation for experience-based analysis throughout the research.

An important thing to explain is that the weighting scheme used in this study is informed by prior work by Feigenspan et al. [5] and Siegmund et al. [14], who demonstrated that

certain self-estimated experience items, particularly familiarity with the paradigms and self-rated skill compared to peers, were statistically predictive of performance in programming tasks. Using stepwise regression, they assigned specific weights to these items and showed that a linear combination could effectively model experience. Inspired by their findings, this study incorporates this set of experience indicators and applies custom weights reflecting each item's expected influence on coding habits and maintainability. This approach aligns with Feigenspan's earlier thesis work, which aggregated coded responses into a single score, as well as later studies that normalized and averaged multiple self-estimate items to produce a composite experience metric [15]. While not derived from a formal regression model, the weights used here are grounded in domain knowledge and reflect the relative importance of each factor as supported by the literature.

3.3 Measuring Code Maintainability

Maintainability in this study is understood as the degree to which student-written code can be easily understood, modified, and extended. This interpretation is consistent with ISO/IEC 25010 and aligns with definitions from prior literature such as Ardito et al. [1], who describe maintainability as a composite attribute made up of analyzability, modularity, documentation quality, and structural clarity. In the context of student software projects, maintainability becomes especially important due to frequent handoffs between team members, variable experience levels, and the short but intensive nature of group-based development tasks. Measuring maintainability helps uncover not just technical issues, but also collaboration habits, style adherence, and clarity in communication through code.

To evaluate maintainability systematically and quantitatively, we selected the following five metrics:

- **Code Smells:** These refer to stylistic or structural warnings such as duplicated code, overly long methods, or excessive nesting. Code smells are widely accepted indicators of poor design practices and reduced maintainability. For each student, the total number of code smells across their contributions was extracted. To allow cross-student comparison, this count was normalized by dividing by total lines of code, resulting in *code smells per 1,000 LOC*.
- **Comment Density:** This measures the ratio of comment lines to total code lines. Adequate commenting aids comprehension and team communication, especially in educational settings where code needs to be readable not just by compilers but by peers. This metric was normalized as a percentage (*comment lines / LOC* $\times$ *100*).
- **Code Duplication:** The percentage of code blocks that are repeated within a student's codebase. High duplication often signals poor modularization and hinders code reusability and testability. This metric is used in percentage form without further normalization, as it is already relative to the size of the codebase.
- **Average Lines of Code per File:** Average file size in terms of lines of code, used as a rough proxy for modularity and code structure. While not a direct indicator of quality, excessively large files often indicate a lack of decomposition or adherence to the single-responsibility principle.
- **Naming Quality:** Variable and function names are pivotal for code readability and maintainability. Research by Avidan and Feitelson (2017) demonstrates that meaningful identifier names significantly aid in program comprehension, particularly for novice programmers [2]. To evaluate naming quality, we assessed each student's code based on clarity, consistency, and descriptiveness of identifiers. A 1–5 scale was employed, where 1 indicated poor naming practices and 5 represented excellent naming conventions. At least two representative files per student were reviewed to ensure a comprehensive assessment. While subjective, this evaluation captures aspects of code quality not easily measured by automated tools.

When analyzing how code maintainability evolves over time, it becomes necessary to aggregate the selected metrics into a single composite value per student per week, as it is difficult to account for each metric individually when comparing experience level against time. This allows for a clearer visualization of trends across project phases without focusing on individual metric dimensions. To achieve this, an evenly weighted average of the five maintainability metrics is used. Each metric is first normalized to ensure comparability, with inverse scaling applied to metrics such as code smells and LOC per file, where lower values indicate better maintainability. While this temporary aggregation supports longitudinal analysis, the study treats each metric as individually meaningful in other parts of the analysis and does not focus on the overall score created.

Furthermore, to attribute these metrics to individual students, we bundled all Git commits by author using `git log`, `git show`, and `git diff`, excluding the merge commits as they mix the contributions of different students. A custom Python script was used to extract all files modified by each student and organize them into isolated directories, effectively creating personal snapshots of their contributions. Each directory was treated as an independent project and analyzed using a locally deployed instance of SonarQube, an open-source static analysis platform. After the individual bundles were run on the local SonarQube server, the first four metrics could either be obtained through the server's API or could be calculated from the information there, without additional effort. Naming quality was assessed manually by reviewing at least two representative files per student and rating identifier clarity, consistency, and descriptiveness on a 1–5 scale. To ensure fairness and accuracy, formatting-only commits and non-source files were excluded from the analysis, and in cases of shared file authorship, line-level attribution was determined using `git blame`. All steps in the analysis pipeline were recorded for reproducibility and transparency.

3.4 Data Analysis

To investigate the first research question, which explores whether programming experience correlates with code maintainability, Pearson correlation analysis was applied. The

Pearson correlation coefficient (r) quantifies the linear relationship between two continuous variables, ranging from -1 (strong negative correlation) to $+1$ (strong positive correlation), with 0 indicating no linear correlation [6]. For each of the five maintainability metrics, code smells, comment density, code duplication, average lines of code per file, and naming quality, the correlation with the overall experience score was calculated. This provided a metric-by-metric view of how experience levels relate to maintainability. To address the second research question, which investigates which maintainability metric best reflects experience, the relative strength of each correlation was compared. This included not only evaluating the Pearson coefficients, but also computing effect size estimates (using r^2) to determine the proportion of variance in each maintainability metric explained by experience. The third question, concerning how maintainability evolves throughout the project, was answered using a longitudinal analysis. Maintainability scores were averaged by experience group (high, mid, low) across four key project weeks (Week 4 to Week 7). These values were then plotted to visualize how code quality progressed over time within each experience category, allowing for trend comparison between groups. All analyses were performed using the `scipy.stats` module in Python, an open-source scientific computing library. The assumption of normally distributed data, required for the validity of Pearson correlation, was evaluated using the Shapiro-Wilk test.

4 Results

4.1 Programming Experience Scores

Table 1 presents the final experience scores for all participants, expressed on a 0–100 scale. To preserve anonymity, individual survey responses are not disclosed.

Student	Experience Score (%)	Experience Category
1A	85.36	High
1B	71.21	High
1C	64.00	Mid
1D	56.14	Mid
1E	67.57	High
2A	46.50	Low
2B	63.29	Mid
2C	65.00	High
2D	47.71	Low
2E	37.86	Low
3A	49.36	Low
3B	50.14	Mid
3C	73.79	High
3D	41.86	Low
3E	53.57	Mid
4A	53.50	Mid
4B	40.21	Low
4C	62.21	Mid
4D	74.36	High
4E	48.71	Low

Table 1: Overall experience scores and corresponding experience categories

4.2 Maintainability Metric Results

Table 2 summarizes the maintainability metrics extracted from the code contributions of each student. The values represent a combination of automatically computed results from static analysis tools and manually evaluated naming quality scores. Each metric reflects the characteristics of code written by an individual student, allowing us to examine trends and patterns across different levels of programming experience. These metrics form the basis for the correlation and interpretation steps described in the following sections.

Student	Comment	Duplication	Code Smells	LOC per File	Naming Quality
1A	23.73	3.19	8.41	96.12	5
1B	26.52	7.83	9.63	86.69	5
1C	24.59	5.65	16.51	100.13	4
1D	30.42	5.42	19.83	106.34	4
1E	20.34	5.85	11.43	73.32	4
2A	15.53	2.32	18.32	120.23	5
2B	14.78	3.24	16.74	113.65	4
2C	20.91	2.67	15.43	109.84	5
2D	18.71	3.75	16.52	90.83	5
2E	19.13	4.53	19.57	101.68	4
3A	17.02	6.23	19.34	110.35	3
3B	19.58	5.91	16.92	105.73	4
3C	24.54	4.18	11.27	85.25	4
3D	16.57	7.16	20.41	115.91	3
3E	21.83	5.57	14.21	98.33	4
4A	20.30	3.78	15.60	95.24	5
4B	16.70	4.25	16.60	102.24	4
4C	21.40	5.42	14.20	110.24	5
4D	22.50	3.54	12.84	88.24	4
4E	19.20	5.56	16.80	123.24	5

Table 2: Maintainability metrics per student

The dataset encompasses all 20 students and will be used in the correlation analysis and subsequent discussion on how experience may relate to software maintainability in educational settings.

4.3 RQ1: Experience and Maintainability Metrics

Pearson correlation coefficients were computed to examine the relationship between each student's experience score and their corresponding maintainability metric values. Prior to this analysis, the normality of each maintainability variable was assessed using the Shapiro-Wilk test to ensure the assumptions for Pearson's r were met. All metrics except naming quality passed the normality test at $\alpha = .05$, indicating that most variables were suitable for Pearson correlation analysis.

Naming quality, which was manually rated on a discrete 1–5 scale, did not follow a normal distribution. This is likely due to the limited range and clustering of scores near the top end of the scale, resulting in a ceiling effect. Despite this, the metric was retained for completeness, as it represents an important qualitative aspect of maintainability that complements the more objective static analysis metrics.

In terms of effect size, all metrics demonstrated large magnitudes, with Cohen's d ranging from 3.88 to 5.77. These values suggest that the relationships observed in the data are not only statistically noticeable but also potentially meaningful in practical terms, reinforcing the relevance of programming experience as a factor influencing maintainability outcomes.

The strongest correlation was observed between programming experience and the number of code smells per 1,000 lines of code. As shown in Figure 1, this relationship was strongly negative, $r(18) = -.84, p < .001$. Students with higher experience scores tended to produce code with fewer smells, reinforcing code smells as the most sensitive indicator of experience in this dataset.

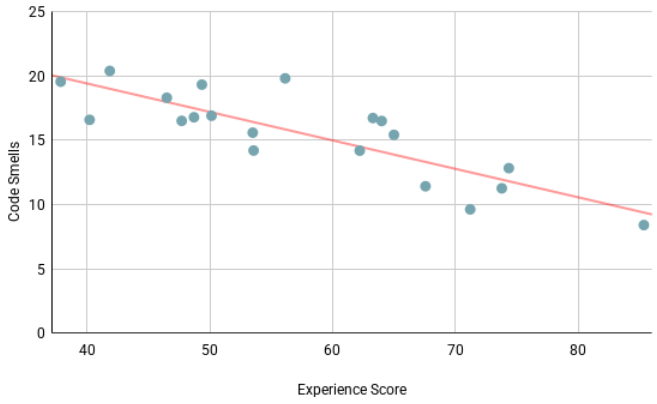

Figure 1: Correlation between programming experience and code smells per 1,000 LOC.

A moderate positive correlation was found between experience and comment density, $r(18) = .55, p = .037$, as presented in Figure 2. The data points show some scatter and outliers, particularly in the mid-to-lower experience range. Nonetheless, the positive trend is evident: students with more experience tend to comment more consistently.

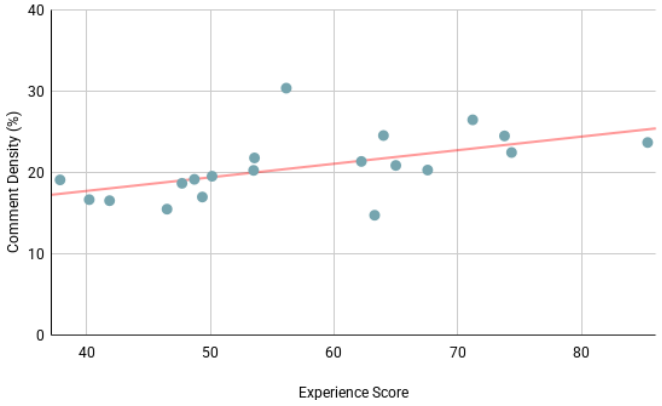

Figure 2: Correlation between programming experience and comment density.

Code duplication showed the weakest correlation, $r(18) = -.14, p = .221$, as shown in Figure 3. The relationship was nearly flat, suggesting little to no association between experience and duplication patterns in this dataset.

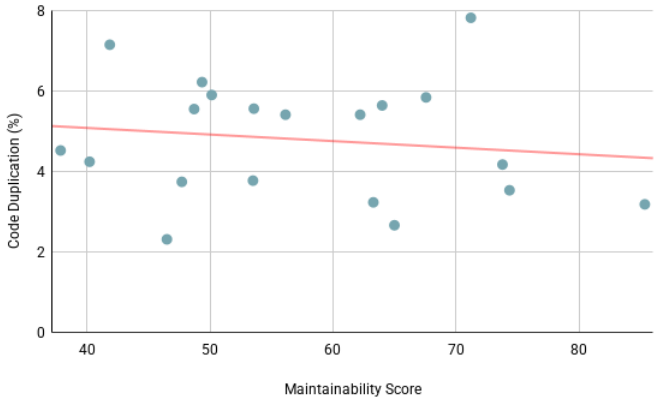

Figure 3: Correlation between programming experience and code duplication percentage.

The relationship between experience and average lines of code (LOC) per file was moderately negative, $r(18) = -.50, p = .060$, as seen in Figure 4. Although some students produced unusually large or small files, the general trend suggests that more experienced students create more modular file structures.

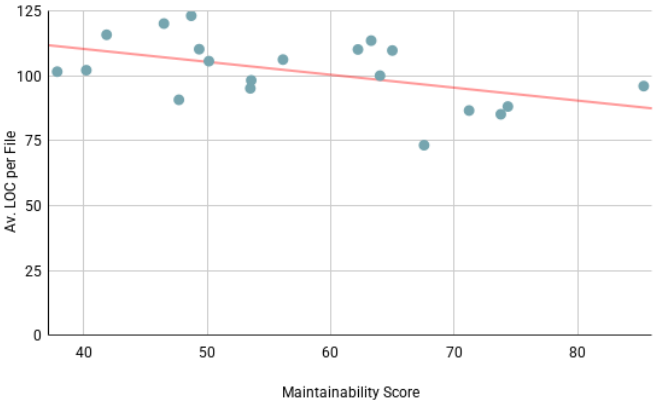

Figure 4: Correlation between programming experience and average LOC per file.

Naming quality was positively but weakly correlated with experience, $r(18) = .27, p = .327$, as shown in Figure 5. The Shapiro-Wilk test indicated non-normal distribution, likely due to scores clustering around 4 and 5. This limited variation may have reduced the observable correlation.

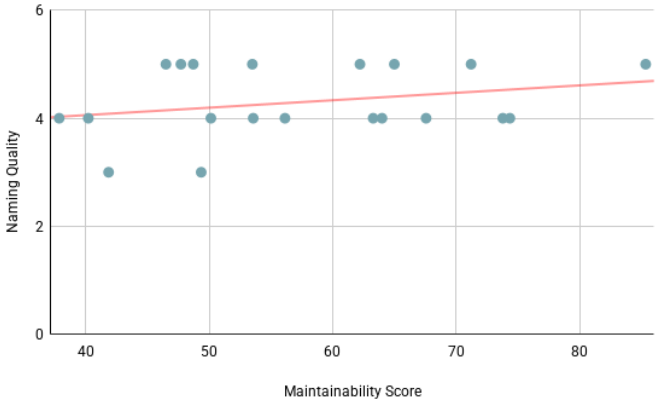

Figure 5: Correlation between programming experience and naming quality.

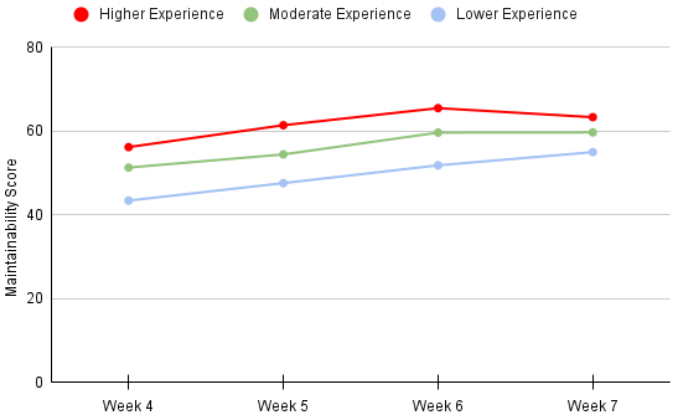

Figure 6: Maintainability score progression by experience group (Weeks 4–7)

In summary, all maintainability metrics showed some level of association with experience. Code smells, comment density, and LOC per file had the strongest correlations, while duplication and naming quality were weaker. The direction of all associations supports the hypothesis that programming experience shapes maintainability practices.

4.4 RQ2: Best Metric for Experience

Among the five maintainability metrics examined, code smells per 1,000 lines of code showed the strongest correlation with experience, $r(18) = -.84, p < .001$, as it can be observed in Figure 1. This indicates a very strong negative linear relationship: as experience increases, the number of code smells decreases substantially. Compared to the other metrics, which had weaker and in some cases non-significant correlations, code smells demonstrated the clearest and most consistent alignment with the experience scores. Therefore, code smells can be considered the most accurate single metric for distinguishing between students with varying levels of programming experience in this dataset.

4.5 RQ3: Maintainability Over Time

Figure 6 shows the progression of maintainability scores over time, grouped by experience level. The chart tracks average scores from Weeks 4 to 7 for the high, mid, and low experience groups.

All groups demonstrate improvement between Weeks 4 and 6. The high-experience group starts highest and peaks in Week 6 before slightly declining. The mid-experience group shows a similar pattern with a flatter rise. The low-experience group starts lowest but improves steadily, especially between Weeks 4 and 6. By Week 7, the gap narrows across groups, though notable differences remain.

5 Responsible Research

This study involved the collection and analysis of data from student software development teams, including survey responses and version control records. Because these data sources may contain information that can be linked to individual participants, ethical considerations played a central role in the research design.

Anonymity and Privacy Protection

All participants were assigned pseudonyms at the beginning of the study to safeguard their identities. These pseudonyms were consistently used in both the survey data and Git-based code contributions, ensuring that no real names, student numbers, or other personally identifiable information were recorded or stored. This anonymization strategy was essential in minimizing the risk of exposing individual performance or behavior.

Informed Participation and Voluntariness

Participation in the study was entirely voluntary. Students were informed in advance about the purpose of the research, the types of data being collected, and how the results would be used. They were explicitly told that they could choose not to participate or stop participating without any consequences. Only the data of students who provided informed consent were included in the analysis, in accordance with ethical research practices.

Patterns and Evaluation

The study was designed to uncover general patterns in the relationship between programming experience and code maintainability, not to evaluate or grade individual students. The analysis focused on trends across the dataset rather than on individual outcomes, helping to ensure that the research remained exploratory rather than judgmental in nature.

Reproducibility and Transparency

To support reproducibility and transparency, all stages of the research process were thoroughly documented. This includes

the design of the experience survey, the weighting and normalization procedures used to compute experience scores, the tools and criteria applied in measuring code maintainability, and the statistical methods used in correlation analysis. All tools used in the study are publicly available and widely used in both academic and professional contexts. The analysis was carried out using simple and explained custom scripts, and open-source analysis software, so that the study can be independently repeated under similar conditions.

By following the principles of privacy, informed consent, methodological transparency, and non-evaluative analysis, this project demonstrates a strong commitment to responsible research, as these practices not only protect participants but also help make sure that the findings are credible, ethically sound, and valuable for future replication or further research in educational software engineering.

6 Discussion

6.1 Is there a correlation between experience and code maintainability?

The findings of this study demonstrate a clear and directionally consistent relationship between students' programming experience and the maintainability of their code. All five analyzed metrics exhibited correlations in the expected direction, indicating that students with higher experience scores tend to write code that is cleaner, more modular, and better documented. However, the strength and nature of these relationships varied, offering important nuance to the interpretation.

The strongest correlation was found between experience and the number of code smells per 1,000 lines of code, with a Pearson coefficient of $r = -0.84$. This strong negative association reinforces the idea that experienced students are more adept at avoiding structural anti-patterns such as long methods, large classes, or complex nesting. These types of flaws are commonly associated with poor maintainability and are unlikely to be avoided without an internalized understanding of code design principles. This observation aligns with previous research that identifies experience as a driver of architectural awareness and long-term quality thinking [16]. That this relationship persisted across teams using different programming languages (Rust, Svelte, Dart) further emphasizes its generalizability.

Moderate correlations were also observed for comment density ($r = 0.55$) and average lines of code per file ($r = -0.50$), both of which are commonly used indicators of maintainability. These suggest that more experienced students not only write cleaner code but also structure it in a way that is easier to read and navigate. Prior studies have highlighted that experienced developers are more likely to anticipate the needs of their peers by documenting functionality and maintaining focused, modular files [8,17]. Nonetheless, these metrics also reflect some variability, likely influenced by instructional guidelines, team practices, or the nature of individual tasks within the project.

The correlations for naming quality ($r = 0.27$) and code duplication ($r = -0.14$) were weaker, suggesting these aspects of maintainability may be less directly shaped by experience in the short term. The narrow range of naming scores, mostly clustered between 4 and 5, suggests a ceiling effect that limited the potential to detect differences. Furthermore, naming was manually scored using a rubric, introducing subjectivity and reducing variability. Code duplication, on the other hand, may be affected by non-individual factors such as framework-specific boilerplate, shared files, or inconsistent refactoring responsibilities across team members. These findings are consistent with the literature noting that not all maintainability outcomes are equally attributable to individual effort [11].

Overall, the consistency in correlation direction across all metrics supports the hypothesis that programming experience contributes positively to code maintainability. Yet, the differences in strength also highlight the complexity of both constructs. Maintainability is shaped not only by individual experience but also by collaborative norms, project structure, and the educational environment. Similarly, experience is a multidimensional construct encompassing time, exposure, and task diversity [5, 14]. These results suggest that while experience is a valuable predictor of code quality, it should be considered in the broader context of group dynamics and project-level scaffolding. As a practical implication, educators might consider pairing students of different experience levels to balance initial disparities and promote peer learning in maintainability practices.

6.2 Which maintainability metric best reflects experience?

Among the analyzed metrics, code smells most effectively captured the impact of programming experience. Its correlation with experience ($r = -0.84$) was not only the strongest statistically, but also the most visually consistent in the data, suggesting a reliable and robust relationship. This aligns with prior research, including Ardito et al.'s systematic review, which highlights code smells as a central indicator of maintainability across different tools and software domains [1]. The use of SonarQube to detect code smells in this study is further supported by its widespread application and validation in both educational and industrial settings [9].

Code smells represent recurring patterns in code that indicate poor design or implementation choices, such as long methods, high cyclomatic complexity, large classes, low cohesion, or lack of encapsulation. These structural flaws typically stem from deeper issues in program organization and logic, and they often accumulate when developers lack familiarity with design principles or refactoring techniques [7]. Because these smells are difficult to eliminate without fundamentally restructuring the code, they are less susceptible to superficial fixes or future improvements. As such, their presence, or absence, serves as a strong proxy for internalized programming practices and architectural awareness, the core aspects of what it means to be experienced.

In contrast, metrics like comment density and naming quality, while relevant, are more easily influenced by external factors. Students can add comments or rename variables late in the development cycle without necessarily improving the structural integrity of their code. These actions may reflect compliance with guidelines rather than actual design insight. Similarly, code duplication may result from team work-

flows, reused templates, or framework constraints, especially in frontend contexts, making it harder to isolate individual effects.

Comment density and average lines of code per file did show moderate correlations with experience, suggesting that more experienced students tend to write more readable and modular code. However, their interpretability is bounded by instructional norms or codebase conventions. Naming quality showed a positive but weaker relationship, and its limited variation across the dataset (with most students scoring between 4 and 5) likely reflects a ceiling effect in the rubric-based scoring. Code duplication had the weakest correlation, reinforcing the idea that it is not solely driven by experience.

In summary, code smells appear to be the most diagnostically valuable metric for assessing the influence of programming experience on code maintainability. Their association with deeper structural decisions, resistance to surface-level fixes, and relevance across languages and projects makes them a particularly meaningful indicator for educators and researchers aiming to evaluate or improve software quality in learning environments.

6.3 How does maintainability evolve throughout the project?

To explore the progression of maintainability over time, the composite maintainability scores for each experience group were tracked from Week 4 through Week 7. All groups demonstrated a general improvement, though the trajectory and rate of change differed substantially.

Students in the high-experience group began the project with the highest maintainability scores and continued to improve through Week 6, followed by a slight decrease in Week 7. This plateau and dip could reflect reduced emphasis on polishing near the deadline or increased workload across multiple responsibilities. The mid-experience group exhibited steady gains, with scores rising consistently before leveling off toward the end of the project. The low-experience group, starting with the lowest maintainability levels, showed the steepest upward trend, improving week after week and closing much of the initial gap by Week 7.

This convergence pattern aligns with previous studies indicating that structured, collaborative environments help less experienced students internalize good software practices over time [4, 13]. Iterative development, frequent code sharing, and exposure to teammates' contributions may improve gradual learning that reduces initial disparities. Moreover, these results underscore the developmental potential of team projects, not only as assessment tools but also as active mechanisms for improving student practices through sustained participation.

The differing trajectories also suggest that experience influences how students adapt during the course. More experienced students may reach a performance ceiling earlier or reallocate focus to other areas of the course. In contrast, lower-experience students may continue to benefit from each cycle of review and refinement. Further investigation is needed to examine how instructional scaffolding, peer feedback, or internal motivation contribute to these patterns. Nonetheless, the results support the value of longitudinal analysis when studying how code quality evolves within educational settings.

7 Conclusion, Limitations and Future Work

This study examined the relationship between students' prior programming experience and the maintainability of their code in a university-level group software project. The analysis was guided by three research questions: (1) whether programming experience correlates with maintainability outcomes, (2) which maintainability metric best reflects a student's experience level, and (3) how maintainability evolves over time during the course of a project.

Using a composite experience score derived from a weighted survey and five maintainability metrics obtained through a combination of static analysis and manual review, the study found that programming experience is meaningfully associated with code maintainability. The strongest correlation was observed for code smells ($r = -0.84$), followed by moderate associations for comment density and average file size. Naming quality and code duplication exhibited weaker but directionally consistent trends. These findings suggest that prior programming experience influences not only functional correctness, but also code structure, clarity, and maintainability.

The results contribute to growing evidence that experience is a multidimensional factor with observable impact on software quality. Code smells, in particular, emerged as a strong proxy for programming maturity and design awareness, making them a valuable metric for educators and researchers seeking to assess code quality in student settings.

Limitations

Several limitations should be acknowledged when interpreting the findings of this study:

First, the participant pool was limited to 20 students from a single university course. As such, the results may not generalize to other educational contexts, institutions, or populations with different levels of programming proficiency.

Second, the group-based nature of the project introduces potential confounding factors such as team dynamics, peer influence, and uneven task distribution. Although individual commit histories and team communication patterns were reviewed, differences in workload, ownership, or collaboration styles may still have impacted the maintainability metrics.

Third, the assessment of code maintainability relied on SonarQube and manual evaluation of naming quality. These methods inherently reflect specific assumptions about what constitutes high-quality code, which may not fully align with educational goals, client-specific coding standards, or alternative maintainability models.

Fourth, the analysis focused on bivariate correlations between experience and individual maintainability metrics. While this approach provides initial insight, it does not capture potential interactions between experience dimensions or account for additional variables such as project complexity or team behavior. A multivariate approach could offer a more comprehensive understanding.

Finally, the potential use of AI-assisted development tools during the course may have influenced code quality independently of programming experience. As such tools become increasingly integrated into the software development process, their impact represents an important area for future research.

Future Work

This research opens several avenues for further investigation:

First, longitudinal tracking across project weeks could explore the third research question, how maintainability evolves, in more detail. More granular analysis could uncover nuanced trends within and between experience groups.

Second, future studies could investigate whether group project settings inherently support learning-by-doing for less experienced students. For example, observational studies or interviews could reveal whether peer practices influence code maintainability habits.

Third, expanding the analysis to incorporate multivariate statistical models would provide a richer view of how multiple experience dimensions interact with maintainability outcomes. This could include factors such as cognitive style, confidence, or exposure to certain tools.

Fourth, the experience survey could be extended to include behavioral and psychological constructs, such as self-efficacy, problem-solving strategies, or reflective coding practices, to capture a more holistic picture of student background.

Finally, replicating the study in different educational contexts, programming languages, or institutional settings would help assess the generalizability of the results and determine how contextual factors mediate the experience–maintainability relationship.

Final Remarks

This study adds to the growing body of evidence that programming experience meaningfully shapes the quality of student-written code in collaborative settings. By combining quantitative metrics with a structured experience model, it highlights both the value and limitations of prior experience as a predictor of maintainability. Ultimately, the findings underscore the importance of not only technical skill, but also team context, peer learning, and instructional design in shaping software quality outcomes in computer science education.

Appendix

A.1 Survey Questions and Scales

The programming experience survey was adapted from Feigenspan et al. (2012) and included the following items:

Survey Question	Response Scale
How many years have you been programming?	Numeric (free entry)
What is the size of the largest project you have worked on?	Categorical: N/A, >900 LOC, 900–40,000 LOC, >40,000 LOC
How would you rate your own programming experience?	1 (very inexperienced) to 10 (very experienced)
How many programming languages do you know?	Numeric (free entry)
How experienced are you with object-oriented programming?	1 (no experience) to 5 (very experienced)
How experienced are you with logical programming?	1 (no experience) to 5 (very experienced)
How many programming-related courses have you completed?	Numeric (free entry)
How would you compare your programming experience to that of your classmates?	1 (much less) to 5 (much more)

Table 3: Programming experience survey questions and their response scales

A.2 Metric Extraction and Analysis Process

All scripts used for data preparation and analysis are publicly available at:

`https://github.com/EgemenTUD/maintainability`
(Last checked on June 21, 2025)

This repository includes:

- A commit extractor script that isolates a student's contributions from a shared repository by cherry-picking their commits across all branches.
- Scripts for conducting Shapiro-Wilk tests and computing Pearson correlations.
- Scripts for calculating the effect size of the correlations.
- A detailed explanation for how SonarQube was installed, set and used.

A.3 Use of Generative AI

During the preparation of this paper, generative AI tools were used in limited and responsible ways to support the writing and development process. Specifically:

- **Grammar and language refinement**: AI assistance was used to identify and correct grammatical issues, improve sentence clarity, and ensure fluency in academic writing.
- **Code and script support**: AI was consulted to assist in creating and debugging Python scripts related to data processing and statistical analysis.
- **LaTeX formatting**: AI provided guidance on formatting tables, figures, and section structure within the LaTeX environment.

References

[1] Luca Ardito, Riccardo Coppola, Luca Barbato, and Diego Verga. A tool-based perspective on software code maintainability metrics: A systematic literature review. *Scientific Programming*, 2020:1–26, august 2020.

[2] Sharon Avidan and Dror G Feitelson. Effects of variable names on comprehension: An empirical study. In *2017 IEEE 25th International Conference on Program Comprehension (ICPC)*, pages 55–65. IEEE, 2017.

[3] David Coppit and Jennifer M Haddox-Schatz. Large team projects in software engineering courses. *ACM SIGCSE Bulletin*, 37(1):137–141, 2005.

[4] Jialin Cui, Runqiu Zhang, Ruochi Li, Fangtong Zhou, Yang Song, and Edward Gehringer. How pre-class programming experience influences students' contribution to their team project: A statistical study. In *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1*, SIGCSE 2024, page 255–261, New York, NY, USA, 2024. Association for Computing Machinery.

[5] Janet Feigenspan, Christian Kästner, Jan Liebig, Sven Apel, and Stefan Hanenberg. Measuring programming experience. In *Proceedings of the 20th International Conference on Program Comprehension (ICPC)*, pages 73–82. IEEE, 2012.

[6] Andy Field. *Discovering Statistics Using IBM SPSS Statistics*. SAGE Publications Ltd, London, UK, 5th edition, 2017.

[7] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA, 1999.

[8] Jose Garcia et al. Improving students' programming quality with the continuous inspection process: a social coding perspective. *Journal of Systems and Software*, 156:1–15, 2019.

[9] Daniel Guaman, Pablo Alejandro Quezada Sarmiento, Luis Barba-Guamán, Paola Cabrera, and Liliana Enciso. Sonarqube as a tool to identify software metrics and technical debt in the source code through static analysis. In *Proceedings of the 7th International Workshop on Computer Science and Engineering (WCSE 2017)*, pages 171–175, Beijing, China, June 25–27 2017. WCSE.

[10] International Organization for Standardization. *Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models*, 2011. ISO/IEC 25010:2011.

[11] Hieke Keuning et al. Assessing understanding of maintainability using code review. In *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education*, pages 123–129. ACM, 2020.

[12] Jefferson Lopes, Johnatan Oliveira, and Eduardo Figueiredo. Evaluating the impact of developer experience on code quality: A systematic literature review. In *Anais do XXVII Congresso Ibero-Americano em Engenharia de Software (CIbSE 2024)*, pages 166–180, 05 2024.

[13] Emerson Murphy-Hill, Thomas Zimmermann, and Nachiappan Nagappan. Confessions of a used programming language: the effect of experience on program quality. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, pages 49–60. ACM, 2014.

[14] Janet Siegmund, Christian Kästner, Jens Liebig, Sven Apel, and Stefan Hanenberg. Measuring and modeling programming experience. *Empirical Software Engineering*, 19(5):1299–1334, 2014.

[15] Stijn van Waveren, Emily J. Carter, Oscar Ornberg, and Iolanda Leite. Exploring non-expert robot programming through crowdsourcing. *Frontiers in Robotics and AI*, 8:645599, 2021.

[16] Michael Wahler, Uwe Drofenik, and Will Snipes. Improving code maintainability: A case study on the impact of refactoring. In *Proceedings of the 32nd International Conference on Software Maintenance and Evolution (ICSME)*, pages 493–501, Budapest, Hungary, 2016. IEEE.

[17] Y. Zhou, E. Denney, and B. Fischer. Assessing the students' understanding and their mistakes in code review checklists. *arXiv preprint arXiv:2101.04837*, 2021.