2024 36th International Conference on Software Engineering Education and Training (CSEE&T) | 979-8-3503-7897-9/24/$31.00 ©2024 IEEE | DOI: 10.1109/CSEET62301.2024.10663028

Cultivating Software Quality Improvement in the Classroom: An Experience with ChatGPT

Eman Abdullah AlOmar*, Mohamed Wiem Mkaouer†
*Software Engineering Department, Stevens Institute of Technology, Hoboken, NJ, USA
†Computer Science Department, University of Michigan-Flint, Flint, MI, USA
ealomar@stevens.edu, mmkaouer@umich.edu

***Abstract*—Large Language Models (LLMs), like ChatGPT, have gained widespread popularity and usage in various software engineering tasks, including programming, testing, code review, and program comprehension. However, their effectiveness in improving software quality in the classroom remains uncertain. In this paper, our aim is to shed light on our experience in teaching the use of Programming Mistake Detector (PMD) to cultivate a bugfix culture and leverage LLMs to improve software quality in educational settings. This paper discusses the results of an experiment involving 102 submissions that carried out a code review activity of 1,230 rules. Our quantitative and qualitative analysis reveals that a set of PMD quality issues influences the acceptance or rejection of the issues, and design-related categories that take longer to resolve. Although students acknowledge the potential of using ChatGPT during code review, some skepticism persists. We envision our findings to enable educators to support students with code review strategies to raise students' awareness about LLMs and promote software quality in education.**

***Index Terms*—large language models, education, bugfix, code quality**

I. Introduction

Linting is one of the code inspection practices in which developers leverage static analysis to identify bad coding patterns, also known as issues. These issues have been known to hinder code quality, making it harder to understand and more prone to errors. Since their inception, linters have been introduced early to students, as part of conceptualizing the avoidance of bad programming practices [17]. Yet, linters output is in the form of warnings with no recommended fix. Due to the non-actionable nature of these warnings [25], and the lack of their comprehension [24], many developers end up considering them as false positives [12].

However, the rise of Large Language Models (LLMs), such as the Generative Pre-trained Transformer, which is the core model behind ChatGPT, has gained popularity due to its generative ability to create responses and design solutions for various input problems. Therefore, the use of ChatGPT in education has become an area of debate about its opportunities and threats to student learning [18]. Given the wide applications of LLMs in source code, including code quality [22], programming [4], bug management [11], and program comprehension [15], it presents an opportunity to bridge the gap between developers and the proper adoption of static analysis. In this context, little is known about ChatGPT's ability to help students effectively comprehend and address static analysis outcomes, as part of learning code inspection.

In this paper, we reflect on the experience of integrating ChatGPT in the code linting process, to support students with their task of debugging and improving the quality of existing systems. Specifically, students are instructed to leverage PMD[1], a state-of-the-art static analysis plugin, to inspect an open source system (code that is not theirs) and identify potential issues belonging to several categories ('Documentation', 'Design', 'Security', etc.). Then, for each identified issue, the student needs to discuss whether it is worth fixing, and so, a corrective action, in the form of a code change, is provided. The student is allowed to use ChatGPT as an assistant when reasoning over the decision of whether to fix an issue and how it should be addressed. In other terms, the language model explainable ability is solicited, along with code generation and debugging. These experiments aim to achieve pedagogical goals related to developing students' code debugging capabilities, which are generally known to be learned in industry [14]. It also familiarizes students with bad coding practices and how to refactor them. Furthermore, it trains students not only to use ChatGPT, but also to review its recommendations, and reason over their validity, with respect to analyzed code, before making a decision of whether and how it should be fixed. This last goal is of particular interest to us, as it gradually elevates students' inherent assumption of ChatGPT's ability to hold the ground truth. In fact, students experience how the language model's code suggestions may not address the issues reported by the PMD tool.

This paper contributes to promoting the wider acceptance of static analysis warnings and leveraging LLMs to improve software quality in educational settings by (i) designing a practical assignment for improving the quality of software systems, and (ii) reporting the experience of using the PMD tool and ChatGPT in software quality assurance course that has been taken by 102 undergraduate and graduate students. As part of the contributions of this paper, we provide the assignment description and the tool documentation for educators to adopt and extend[2].

The remainder of this paper is organized as follows. Section II outlines our experimental setup. Section III discusses our findings, while the reflection is discussed in Section IV. Section V reviews the existing studies. Section VI captures

[1]https://pmd.sourceforge.io
[2]https://refactorings.github.io/education/

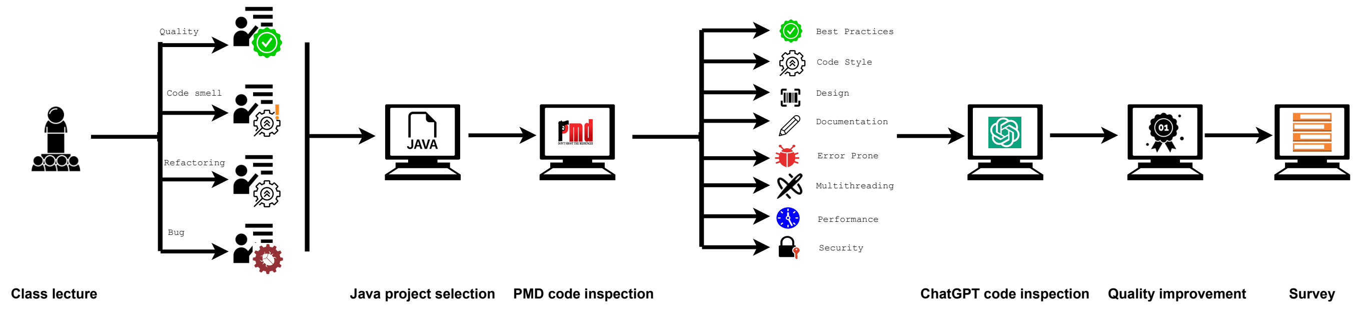


Figure 1: Key phases of our study.

any threats to the validity of our work, before concluding with Section VII.

II. Study Design

A. Goal & Research Questions

We formulate the main objective of our study based on the *Goal Question Metric* template [20], as follows:

> ***Analyze*** *the use of PMD* ***for the purpose of*** *cultivating the culture of bug fix and demonstrating a practical case study of leveraging Large Language Models (LLMs) with respect to support educating software quality* ***from the point view*** *of educators* ***in the context of*** *undergraduate and graduate students in SE/CS who analyze Java-based software projects.*

According to our goal, our aim is to answer the following research questions:

- **RQ$_1$.** ***What PMD-related problems are typically selected by students?***
  Motivation: This RQ explores what type of issues students are addressing using the help of the model.
  Measurement: We report the types of issues that are being addressed, clustered by PMD ruleset categories.
- **RQ$_2$.** ***What type of issues typically takes longer to be fixed?***
  Motivation: This RQ investigates which PMD ruleset takes a longer time to be fixed by students despite the use of ChatGPT. The finding raises educators' awareness of types of issues that are difficult for students to understand and address.
  Measurement: We examine the resolution time taken by students to fix each ruleset category.
- **RQ$_3$.** ***To what extent was ChatGPT successful in addressing the students' debugging needs?***
  Motivation: This RQ explores the ability of the language model to correctly explain the rationale of errors, and to provide potential code fixes when requested. The findings will inform educators about how ChatGPT can support students with code improvement.
  Measurement: We examine students' feedback to extract the necessary ratios of the language model's success in addressing their queries.

As part of this paper's contributions, we provide the assignment description, dataset, and tool documentation for educators to adopt and extend[3].

B. Course Overview

Software quality assurance is an undergraduate and graduate course consisting of two weekly lectures, one hour and 15 minutes each. The course explores the foundations of software quality and maintenance and introduces challenges linked to various aspects of software evolution, along with support tools to approach them. The course also covers various concepts related to software analysis and testing, along with practical tools widely used as industry standards. Students were also given several hands-on assignments on software quality metrics, code refactoring, bug reporting, unit and mutation testing, and technical debt management. The course deliverables consisted of five individual homework assignments, a research paper reading and presentation, and a long-term group project.

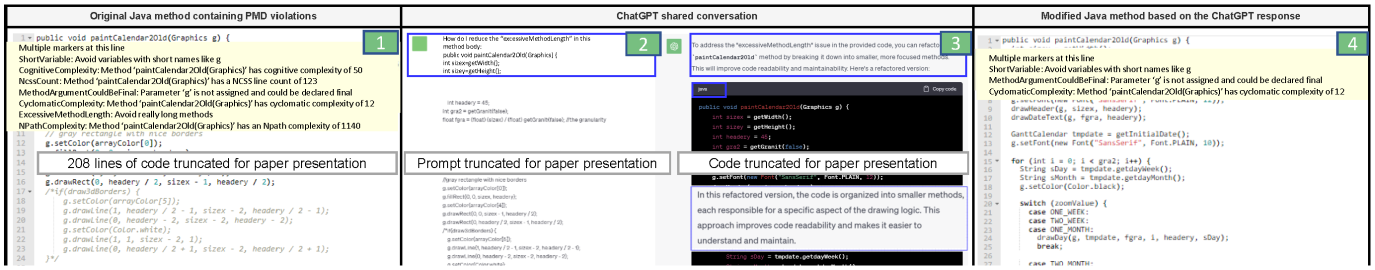
C. PMD

The Programming Mistake Detector (PMD) is an open-source static source code analyzer. It inputs source code from up to 16 languages and reports common programming issues, such as unsafe threading, god classes, and naming convention violations. These issues are clustered into 8 categories, namely: 'Best Practices', 'Code Style', 'Design', 'Documentation', 'Error Prone', 'Multithreading', 'Performance', and 'Security'. Each issue is identified using a detection rule. The tool has become popular, as it can be integrated with modern CI/CD servers.

D. Teaching Context and Participants

The study involves one assignment in the software quality assurance course. The course was taught at Stevens Institute of Technology and Rochester Institute of Technology. Before conducting the assignment, students have already learned about several code, and design quality concerns: (1) code quality (teaching quality concepts and how to measure software

[3]https://refactorings.github.io/education/



Original Java method containing PMD violations

1

```
public void paintCalendar2Old(Graphics g) {
```

Multiple markers at this line
ShortVariable: Avoid variables with short names like g
CognitiveComplexity: Method 'paintCalendar2Old(Graphics)' has cognitive complexity of 50
NcssCount: Method 'paintCalendar2Old(Graphics)' has a NCSS line count of 123
MethodArgumentCouldBeFinal: Parameter 'g' is not assigned and could be declared final
CyclomaticComplexity: Method 'paintCalendar2Old(Graphics)' has cyclomatic complexity of 12
ExcessiveMethodLength: Avoid really long methods
NPathComplexity: Method 'paintCalendar2Old(Graphics)' has an Npath complexity of 1140

```
    // gray rectangle with nice borders
    g.setColor(arrayColor[0]);
```

208 lines of code truncated for paper presentation

```
    g.drawRect(0, headery / 2, sizex - 1, headery / 2);
    /*if(draw3dBorders) {
        g.setColor(arrayColor[5]);
        g.drawLine(1, headery / 2 - 1, sizex - 2, headery / 2 - 1);
        g.drawLine(0, headery - 2, sizex - 2, headery - 2);
        g.setColor(Color.white);
        g.drawLine(1, 1, sizex - 2, 1);
        g.drawLine(0, headery / 2 + 1, sizex - 2, headery / 2 + 1);
    }*/
```

ChatGPT shared conversation

2

How do I reduce the "excessiveMethodLength" in this method body:
public void paintCalendar2Old(Graphics) {
int sizex=getWidth();
int sizey=getHeight();

```
  int headery = 45;
int gra2 = getGranit(false);
float fgra = (float) (sizex) / (float) getGranit(false); //the granularity
```

Prompt truncated for paper presentation

```
//gray rectangle with nice borders
g.setColor(arrayColor[0]);
g.fillRect(0, 0, sizex, headery);
g.setColor(arrayColor[4]);
g.drawRect(0, 0, sizex - 1, headery / 2);
g.drawRect(0, headery / 2, sizex - 1, headery / 2);
/*if(draw3dBorders) {
  g.setColor(arrayColor[5]);
  g.drawLine(1, headery / 2 - 1, sizex - 2, headery / 2 - 1);
  g.drawLine(0, headery - 2, sizex - 2, headery - 2);
```

3

To address the "excessiveMethodLength" issue in the provided code, you can refacto `paintCalendar2Old` method by breaking it down into smaller, more focused methods. This will improve code readability and maintainability. Here's a refactored version:

```java
public void paintCalendar2Old(Graphics g) {
    int sizex = getWidth();
    int sizey = getHeight();
    int headery = 45;
    int gra2 = getGranit(false);
```

Code truncated for paper presentation

```java
    g.setFont(new Font("SansSerif", Font.PLAIN, 12));
```

In this refactored version, the code is organized into smaller methods, each responsible for a specific aspect of the drawing logic. This approach improves code readability and makes it easier to understand and maintain.

```java
        String sDay = tmpdate.getdayWeek();
```

Modified Java method based on the ChatGPT response

4

```
public void paintCalendar2Old(Graphics g) {
```

Multiple markers at this line
ShortVariable: Avoid variables with short names like g
MethodArgumentCouldBeFinal: Parameter 'g' is not assigned and could be declared final
CyclomaticComplexity: Method 'paintCalendar2Old(Graphics)' has cyclomatic complexity of 12

```
  drawHeader(g, sizex, headery);
  drawDateText(g, fgra, headery);

  GanttCalendar tmpdate = getInitialDate();
  g.setFont(new Font("SansSerif", Font.PLAIN, 10));

  for (int i = 0; i < gra2; i++) {
    String sDay = tmpdate.getdayWeek();
    String sMonth = tmpdate.getdayMonth();
    g.setColor(Color.black);

    switch (zoomValue) {
      case ONE_WEEK:
      case TWO_WEEK:
      case ONE_MONTH:
        drawDay(g, tmpdate, fgra, i, headery, sDay);
        break;

      case TWO_MONTH:
```

Figure 2: ChatGPT in action showing the resolution of `ExcessiveMethodLength` PMD issue.

quality), (2) code smells (teaching bad programming practices that violate design principles), (3) code refactoring (teaching refactoring recipes that help improving software quality), and (4) bug management (teaching software bugs and how to locate and fix them). The assignment constituted 7.5% of the final grade. It was due 14 days after the four corresponding sessions.

In total, 102 students completed the assignment. The programming experience of the respondents ranged from 1 to more than 10 years, their Java experience ranged from 1 to 10 years, and their industrial/coop experience ranged from 0 to more than 10 years.

E. Assignment Content and Format

Assignment Design. Initially, students are asked to analyze one version of Java software approved by the instructor to ensure its eligibility based on popularity, in addition to ensuring that it compiles correctly since PMD requires it. The rationale behind giving students the choice of project is to let them choose one that they are comfortable with and that fits their interests. For students who do not want to search for a project, they have been given a list that the instructor has already curated (see Table I). We selected these projects because we already know that they contain a variety of software defects. The students are then asked to set up and run PMD to analyze the chosen project production code. Students are also given the choice of running either the stand-alone version of the tool or its plugin associated with a popular IDE (Eclipse), as we want the students to be familiar with the coding environment and reduce the setup overhead. Upon running PMD, students must choose a minimum of 10 warnings, and at least one from each category, if applicable. We enforce the diversification of warnings to ensure a wider exposure to potential issues, varying from design to multithreading, and documentation. It also increases students' learning curve as they cannot reuse their fix to address multiple instances of the same warning. Yet, we allow students to choose the instances they want to address. It implicitly makes students read many warnings, from all categories, which increases the likelihood of incidental learning. Furthermore, letting students choose the code fragments to review increases their confidence in their decision regarding the given warning. Finally, the students use ChatGPT to fix PMD issues supported by an online questionnaire. For our survey design, we followed the guidelines proposed by Kitchenham and Pfleeger [13]. The survey is divided into 2 parts. The first part of the survey includes demographic questions about participants. In the second part, we ask about their experience using ChatGPT to fix PMD issues and the perceived usefulness of ChatGPT. As suggested by Kitchenham and Pfleeger [13], we constructed the survey using 6 open-ended questions and 16 multiple-choice questions with an optional "Other" category, allowing respondents to share thoughts not mentioned in the list.

Pilot Study. We conducted a pilot study with two students to improve the instrumentation of the experiment and to ensure that the experiment's instructions were clear. Following the pilot study, we interactively refined the protocol and the assignment questions. The participants in the pilot study are undergraduate students in Software Engineering. Among others, we learned that prompt engineering is crucial when using ChatGPT, and we decided to refine the survey questions to better explore the topic. Therefore, we reformulated the assignment and excluded the data from the pilot study from our analysis.

In a nutshell, the students followed these steps:

1) Install the PMD.
2) Run PMD on a project of students' choice and select 10 issues of different types.
3) Use ChatGPT to analyze the issues and decide on whether to fix them and what is the appropriate code change action.
4) Report the findings for each issue: (1) the source code, (2) the type of issue, (3) how long it took to check it / fix it, and (4) the code snippet.
5) Add to the report a concise comment on the experience with ChatGPT (optional).

The evaluation of the artifacts of the submissions was based on 1) the assessment of the students' ability to understand the type of issues (concept understanding); and 2) assessment of whether students have provided acceptable fixes or proper justification in both cases or accepted or rejected PMD's recommendation (program analysis and evolution). Students' perception of the code was excluded from the evaluation process, as it can bias the experiment, as students would be filling out the survey arbitrarily, under the pressure of being graded. Additionally, feedback was anonymous and was not mandatory to increase the magnitude of PMD and ChatGPT usage experience. Although feedback was optional, all students completed it. Figure 1 shows an overview of the setup and

Table I: The list of open-source projects used in the assignment.

Project	# commits	# contributors	Domain
Ant	14,887	64	Java builder
GanttProject	4,361	38	Project management
Hutool	4,074	191	Code design
JCommander	1,009	64	Command line parsing
JFreeChart	4,218	24	Data visualization
JHotDraw	804	3	Data visualization
Log4J	12,211	137	Logging
Nutch	3,293	46	Web crawler
Rihno	4,119	80	Script builder
RxJava	6,004	289	Java VM
Xerces	6,463	5	XML parser

execution of our experiment.

F. ChatGPT Usage

Just like PMD, students were provided with the necessary background training to use ChatGPT. Since we are interested in students interaction with the language model, students were clearly instructed to fully disclose the usage of the tool through a given survey link. Figure 2 shows one of the PMD issues when analyzing the GanttProject[4]. This issue is called `ExcessiveMethodLength`, identified in the `paintCalendar201d(Graphics g)` method in the `GanttGraphicArea.java` file. PMD highlights this issue with 'Urgent' violation due to its `CognitiveComplexity` of 50, `CyclomaticComplexity` of 39, and `NPathComplexity` of 1140, all of which surpass the default acceptance threshold. ChatGPT indicates that refactoring is required to split the lengthy method into smaller and more manageable ones. It also provides a suggested code fix. The student has the option to implement the suggested fix if they agree with it. As shown in the figure, when the proposed fix was adopted, it removed `CognitiveComplexity` and `NPathComplexity` issues, and reduced `CyclomaticComplexity` to 12.

III. Results

A. What PMD-related problems are typically selected by students?

Upon analyzing students' assignment solutions, we cluster the issues according to the PMD ruleset categories listed in the PMD official documentation[5], namely, 'Best Practices', 'Code Style', 'Design', 'Documentation', 'Error Prone', 'Multithreading', 'Performance', and 'Security'. These categories were captured at different levels of granularity (*e.g.,* package, class, method, and attributes).

Looking at the PMD rules, Figure 3 depicts the percentages of PMD rules, clustered by category. As can be seen, the most common PMD ruleset category concerns 'Code Style', representing 32.3% of the issues. This observation is in line with the findings of previous studies describing that most code reviewers look for style conformance when evaluating

[4]https://github.com/bardsoftware/ganttproject
[5]https://pmd.sourceforge.io

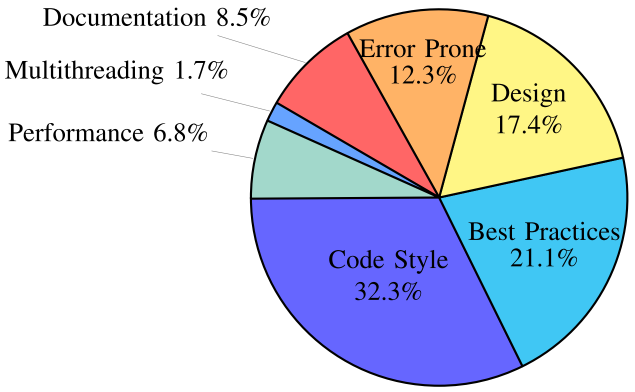


Figure 3: What PMD-related category have you chosen?

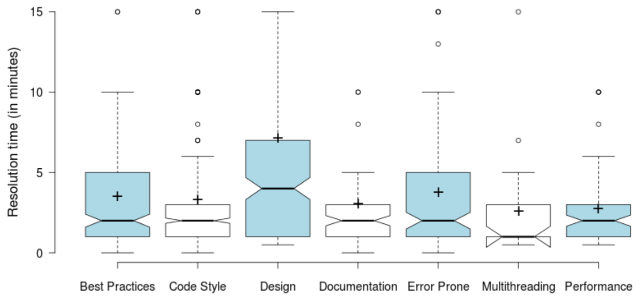


Figure 4: Boxplots of time taken to fix issues, clustered by PMD ruleset categories.

the quality of code [19]. The next most common categories are 'Best Practices', 'Design', and 'Error Prone', representing 21.1%, 17.4%, and 12.3% of the issues, respectively. This might indicate that students have different perspectives on whether developers follow the best practices, improve the architecture design of code, or make code less susceptible to errors. The categories 'Documentation', 'Performance', and 'Multithreading' had the least number of issues, with a ratio of 8.5%, 6.8%, and 1.7%, respectively.

***Summary for RQ$_1$:** Among the 1,230 analyzed issues, the most common PMD ruleset category concerns 'Code Style', representing 32.3% of the issues.*

B. What type of issues typically takes longer to be fixed?

Figure 4 shows that issues belonging to the 'Design', 'Error Prone', and 'Best Practices' categories (μ = 7.19, μ = 3.81, and μ = 3.51, respectively) tend to be more time-consuming for students to address. In particular, 'Design' issues have a significant spike compared to all other types. When taking a deeper look at these issues, we noticed this pattern: submissions where students could not receive support with addressing some type of 'Design' issues, namely `GodClass` issues. God Classes are known to be abnormally large, with a concentration of various responsibilities, by controlling and monitoring many other classes. Their existence hinders program maintenance and comprehension [3]. We speculate that design issues take significantly longer to be resolved due to substantial modifications that must be made for God classes. In fact, fixing a God class requires its decomposition into multiple classes, which requires separating multiple tangled concerns while maintaining their coupling to other classes. The needed fix translates into a series of refactorings that go beyond changing one or a few instructions, like when fixing other types of issues. When analyzing students' prompts for fixing for God classes, we noticed that ChatGPT was unable to propose a fix, mostly because the input windows of the model cannot receive the needed set of instructions, which represent the God class, and all its dependencies. providing a snapshot of the God class code would not be sufficient for the model to carry out a class extraction. The language model was only able to provide examples of extract class refactorings, for relatively smaller-scale classes. For instance, Figure 5 exhibits ChatGPT's response to a student query asking to address a `God Class` issue, for a given input class. In general, ChatGPT has input limitations when grasping the various contexts in large code segments. Due to this limitation, there have been instances where ChatGPT makes suggestions based on misunderstandings or false assumptions about the code. Through this experiment, students can experience how ChatGPT performance varies based on the length of the input. Increasing the input size can eventually lead to catastrophic forgetting [10].

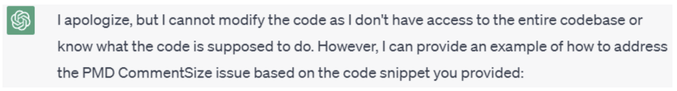

I apologize, but I cannot modify the code as I don't have access to the entire codebase or know what the code is supposed to do. However, I can provide an example of how to address the PMD CommentSize issue based on the code snippet you provided:

Figure 5: ChatGPT response to a student query to refactor a God Class.

On the other hand, ChatGPT provided more positive answers to student questions related to addressing issues when they are located in a continuous set of instructions (single location), such as `CyclomaticComplexity`. ChatGPT was able to propose the extract method refactoring to break down methods with a high cycomatic complexity score.

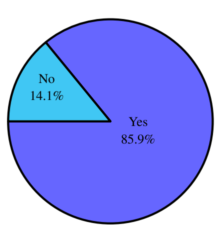
Summary for RQ$_2$: *'Design' PMD ruleset category takes longer to resolve. Unlike other PMD category, this requires going beyond one or few instructions, and ChatGPT has a limited understanding of the broader context and entirety of the codebase.*

C. To what extent was ChatGPT successful in addressing the students' debugging needs?

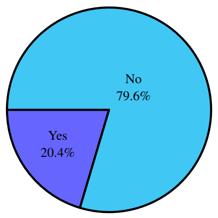
Figure 6 shows students' experience with utilizing ChatGPT in terms of four aspects: (1) the ChatGPT's capability to address issues, (2) the potential for ChatGPT to introduce regression in the code, (3) the frequency of interaction with ChatGPT for bug-fixing purposes, and (4) the possibility for ChatGPT's to introduce additional violations through PMD. When we asked students, "Was ChatGPT able to fix the buggy code?", 85.9% of the students indicated that ChatGPT can help with addressing the issues, while 14.1% of the students responded negatively. Overall, students are satisfied with ChatGPT's support. In terms of causing regression in the code, 79.6% of the students revealed that there is no regression, while 20.4% of students experienced it. Regarding the frequency of the usage and interaction with ChatGPT before finding (or not) a fix to the buggy code, 40% used it once, 21.3% used it twice. The remaining students (38.7%) used ChatGPT between three and more than five times. As for the potential to introduce additional violations through PMD, the majority of students (86.4%) mentioned that ChatGPT does not contain other errors detected by PMD, but some indicated otherwise.

The success of using any language model heavily depends on students' ability to properly prompt it. Therefore, one of these work's outcomes is sensitizing students to the power of prompt engineering in the context of debugging. As shown in Figure 7, when analyzing the students initial prompts, 27.7% of students revealed that they only copy-paste the buggy code fragments, thinking that ChatGPT can identify the intention behind prompting code. The majority of students (68.6%) communicated that they copied the buggy code fragment along with the error description to provide context for the pasted code. 20.6% of students mentioned that they copy-paste the buggy code fragment and added textual description of how to fix it. A few students reported that they show pairs of buggy and fixed code fragments and asked ChatGPT to do similar for the given buggy code. Besides these activities, 2.9% of the students mentioned in the "other" option: "*asking ChatGPT that the PMD violation still remained when it could not fix the code*", "*asking ChatGPT how to fix the error in general before copy pasting the offending code*", "*prompted ChatGPT with the actual line that was causing the error and asked it if there was something specific in the code segment that could be optimized*", "*asked ChatGPT to provide an example of how a similar situation could be fixed*", and "*ChatGPT says this is a false positive and is descriptive of what the variable is representing and I agree with it*". Figure 7 shows how the students' prompts are not uniform, where the majority of students argue over how to extract the necessary action from the models, while others overestimate the capabilities of the model, as outlined in previous studies that have proven that ChatGPT is susceptible to hallucination when it comes to coding semantic structures [15], [22]. So, experiencing the potentially inappropriate results of the model would raise the students' awareness of its limitations. In addition, it helps students refine their prompts, as shown in Figure 6 where the majority of students have used more than one prompt per issue (60%).

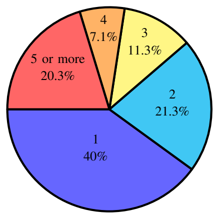
Furthermore, 68.6% of the students' prompts were zero-shot, *i.e.,* students rely on the generative ability of the model to either understand an issue, or to propose its corresponding code fix. Zero-shot learning challenges the model to make a



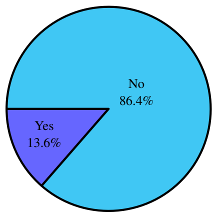
(a) Was ChatGPT generally helpful in addressing PMD issues?



(b) Did one of ChatGPT's proposed solutions cause regression in the code?



(c) How many times have you interacted with ChatGPT?



(d) Did the fix contain other issues detected by PMD?

Figure 6: Student's attitude toward utilizing ChatGPT for bug resolution.

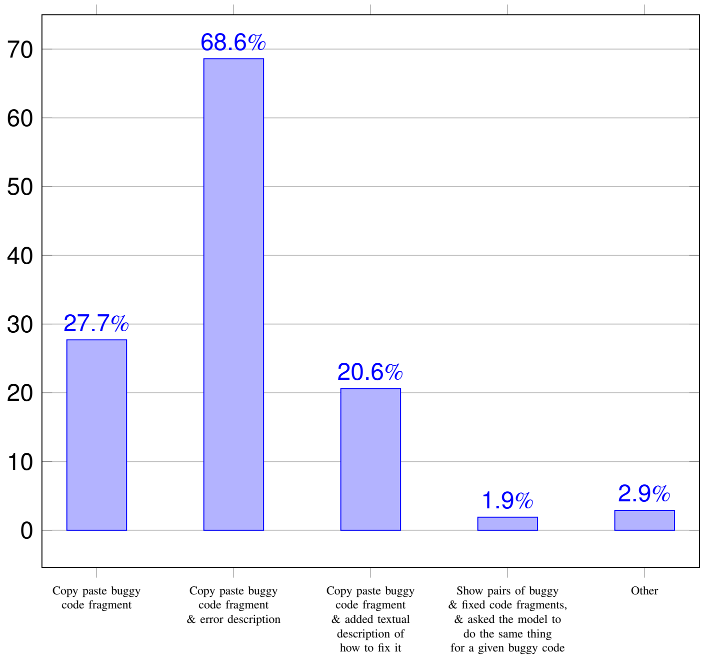


Figure 7: How did you use ChatGPT to fix buggy code?

decision over presumably unseen data [23]. The model relies on approximating the input with previously trained code. For instance, as shown in Figure 8, the student asks the model to reduce the cyclomatic complexity of an input source code. The same prompt can be augmented by adding a label to the unseen data, *i.e.,* one-shot [8]. For example, in Figure 8, mentioning how the reduction can be performed can guide the model towards the decision to take (method extraction). Other students opted for a few-shot learning by entering pairs of buggy and fixed code fragments and asked ChatGPT to propose similar fixes for an alternative buggy code. Few-shot learning is a response when dealing with complex tasks, to steer the model towards better decision making, by allowing in-context learning from provided examples [21]. In Figure 8, the input shows examples of code changes that address the complexity of a given method.

Given the variety of prompts used, our activity has triggered the students' reasoning about how to query the model to avoid its own inherent limitations. This also helps students be critical of ChatGPT responses, and not treat them as ground truth. For instance, 20.4% of students experienced regression in their code base after using the code generated by ChatGPT. Reported examples ranged from the introduction of compiler errors to failing unit tests. Such problems can be due to existing errors in the ChatGPT generated code or due to the wrong integration of the fix by students into the code base. It can also be eventually caused by an incorrect student query. Although it would be interesting to dive into this analysis more in-depth, our goal was to sensibilize students to how the process of querying language models can be error-prone, and thus it is critical to verify its outcomes before any adoption. We also aimed to sensitize students to the existence of multiple querying techniques, which are known in data science curricula, without providing any prompt engineering training.

In Table II, we report the main thoughts, comments, and suggestions about the overall impression of the usefulness, usability, functionality, and recommendation of the ChatGPT, in accordance with the conducted labeling. The table also presents samples of the students' comments to illustrate their impressions of each theme.

Usefulness. Generally, the respondents found the ChatGPT to be useful in regard to five main aspects: automation, quality, scope, awareness, and experience. A group of students commented on ChatGPT's ability to automatically fix bugs, its affordability, capacity for automating tedious debugging and bug fixing processes, and ability to deliver precise and thorough responses. Nearly 85.9% of the students commented that ChatGPT is useful as it gives explanations as to what the problem was and the suggestions to fix it. Further, 23.22% of students commented that it is a nice application and provides a good learning experience especially for those who are beginners in software quality and debugging. Some students (20.64%) revealed that ChatGPT was fast in terms of analyzing issues located in contiguous instructions, but not for identifying larger issues such as `GodClass` and `DataClass`. 4.5% of students communicated that detecting the issues assists in increasing its *readability*, which helps improve overall code quality.

Usability. Based on the feedback the students provided, the key usability areas were documentation, ease of use, explainability and user interface. 28.38% of the students pointed out

Zero-shot learning:

You are a Java expert. How do I fix this section of code that has a cyclomatic complexity of 21:
“‘ *Original code* ”’

One-shot learning:

You are a Java expert. Can you fix this section of code that has a cyclomatic complexity of 21 by breaking down the code into smaller methods:
“‘ *Original code* ”’

Few-shot learning:

You are a Java expert. I provide you with a pair of fixed and buggy code fragments, and I request that you apply similar fixes to a newly given buggy code.

An example of buggy code that has a cyclomatic complexity of 21:
“‘ *Original buggy code* ”’

An example of fixed:
“‘ *Original fixed code* ”’

Here is the newly buggy code, please propose similar fixes for this buggy code:
“‘ *Newly buggy code* ”’

Figure 8: Different types of learning settings used by students when prompting ChatGPT in the context of debugging.

that the tool is easy to use, has a user-friendly interface, and explains how it generated each bugfix and how to avoid that issue in the future. Other comments also stated that using these LLMs requires to carefully adjust the model’s output. It lacks originality and might give biased output.

Functionality. According to the students’ feedback about the tool’s functional features, 16.77% of the students’ comments appreciate ChatGPT’s correction, detection, and debugging functionality and are satisfied with various aspects of the tool’s operation, and how this feature helps to understand bad programming practices in real-world scenarios better. Additionally, the students commented on their ability to practice a variety of strategies to eliminate issues.

Recommendation. From the students’ feedback, we have also extracted suggestions to improve the LLMs. 3.8% of the students’ comments show a couple of suggested changes as a recommendation to be made to the tool’s operation. We extracted a set of recommendations related to context, promptness, accuracy, resolution, and verification. Students commented that it is undeniable that ChatGPT’s capabilities have the potential to transform the programming process for developers. With its remarkable ability to generate multiple solutions to a problem and adapt to a developer’s preferred approach. Others pointed out how ChatGPT can eventually hamper the creativity of developers. Students anticipate that ChatGPT will revolutionize the way developers approach coding in the future. Other students felt that ChatGPT could quickly fix language-specific coding errors, along with PMD default issue descriptions, when they are well explained. But when the errors are propagated or occur in multiple locations, the students had difficulty querying the model. These limitations can be mitigated through formal training in prompt engineering. The fact that ChatGPT’s replies are determined by the prompt it gets and the data it was trained on could be a drawback as the quality of its responses are highly dependent on the quality of the prompts. Its solutions may not always be precise or appropriate for the particular activity or topic. The students also mentioned that ChatGPT is helpful in most situations. However, it struggles to solve issues without context. For example, ChatGPT might find the code comprehensive and easy to read, but when students mention the problem with the code, it then recognizes the issue. The more detailed the description, the more specific and accurate the solution made by ChatGPT. However, some issues require investigating various coupled files that may or may not be present as part of the input. Therefore, even after inputting a proper description of the buggy code, there are no guarantees on the model’s ability to debug. Aside from the detailed description of the issue, providing the most relevant code fragments is key to avoid dragging ChatGPT attention to either unnecessary details or bug-free code fragments.

Summary for RQ$_3$: *Overall, this assignment helps cultivate analytical and critical thinking skills as students engage in the debugging process. In addition, it teaches students to be skeptical towards the use of ChatGPT by shedding light on the limitations of ChatGPT in identifying and solving problems. It highlights the importance of incorporating other traditional static analysis tools and techniques to improve the accuracy and efficiency of their predictions. Furthermore, the involvement of a human-in-the-loop, capable of comprehending code, can be highly valuable and desirable.*

IV. Reflection

This section provides the lessons learned from both the educator’s and the student’s perspectives.

Lesson #1: ***Develop complementary assignments.*** ChatGPT identified each of the PMD violations and attempted to fix them. In some cases, this would cause the problem to vanish completely from the PMD analyzer. In other cases,

Table II: Student’s insight about the usefulness, usability and functionality of ChatGPT.

Theme	Sub-theme	Example (Excerpts from a related student’s comment)
Usefulness	Automation	“*The advantages of ChatGPT include its affordability, **capacity for automating tedious processes**, and ability to deliver precise and thorough responses for customer service departments.*”
	Quality	“*PMD is a useful tool that can help identify potential issues in code and provide suggestions for improvement. It can also help enforce coding standards and best practices, **leading to cleaner, more maintainable code**. In my experience, both ChatGPT and PMD have been valuable tools. ChatGPT has been able to quickly provide helpful responses to various questions, while PMD has been able to catch potential issues that may have been missed during manual code reviews. However, it’s important to note that these tools should be used as aids and not as replacements for human analysis and judgment. Overall, I believe that the use of AI language models and code analysis tools like PMD can greatly **benefit developers and lead to better code quality and efficiency.***”
	Scope	“*ChatGPT was very informative about giving context and definitions of errors, and was even able to provide definitions and recommendations specific to the PMD plugin. Negatives included when code errors were too vague or covered code segments that were too large for ChatGPT to understand. I conclude that **ChatGPT is useful for refactoring small segments of code** like fixing errors in smaller methods, nested if statements, and generally cleaning up the code to run faster, **but not for larger issues like god classes**.*”
	Awareness	“*ChatGPT was extremely useful. **It had all of my answers I needed, and even was able to answer other questions for me**. When it fixed the code, it also gave explanations to each change it made to the code. Overall, ChatGPT was an extremely useful and easy to understand.*”
	Experience	“*The refactored code given by ChatGPT was bug free most of the time but at few times there were bugs in it . Overall the chatgpt is very **nice application specially for those who are begineers in software development and programming**.*”
Usability	Documentation	“*My experience with ChatGPT was largely positive. The tool was able to quickly identify a range of issues in the codebase, including code style issues, best practices, performance issues, and design problems. The **reports generated by ChatGPT were very informative and easy to understand, providing clear explanations of each issue and suggesting steps to fix them**.*”
	Ease of use	“*It was **quite easy to fix errors using ChatGPT**. The trick is to find the right prompt so that it understands what it needs to do quickly rather than going around explaining the issue without providing the fix. However, since ChatGPT only had limited code visibility (the code snippet provided by the user), it generated a few fixes which would break the code (ex change variable names, use packages without importing, etc.). We can’t say that the code fix provided by ChatGPT was entirely wrong though. Overall, it is **easy and quick to use ChatGPT to fix most PMD Errors, with a little bit of oversight**.*”
	Explanability	“*when you’re working on a small scale project it’s extremely useful. Can also **explain basic concepts you might have forgotten very well**.*”
	User interface	“***Chatgpt has a user-friendly interface**. It provides precise and thorough responses for customer service teams. However, there are also potential disadvantages to using these models, such as the need to carefully adjust the model’s output. It lacks originality and might give biased output.*”
Functionality	Correction	“*ChatGPT would **sometimes misunderstand what error I was trying to indicate**, or would be **confused by the fact that I was giving it a code excerpt rather than the full code**, which was too large to provide to it. Its corrections sometimes also misunderstood the purpose of an excerpt and would slightly alter the logic, but this was expected.*”
	Detection	“*Overall I had a positive experience with ChatGPT as it was able to **detect the errors that PMD reported**. However for the last 5 code segments I tested, it **started generating code that produced compile errors** and it didn’t always generate the code in java on the first few attempts.*”
	Debugging	“*ChatGPT is usually unable to identify the problems the pmd extension provides. However when it does it is able to go into great detail about the problems, and either fix the problems entirely or give guidance as to how to fix them. I **enjoy using chatgpt for debugging** and other things and I believe it is **a great academic learning tool**.*”
Recommendation	Context	“*ChatGPT was definitely helpful to fix errors. However, **many errors required a lot more context than what chatgpt accepts and that leads to incorrect fix** or sometimes makes the problem worse. There is also the problem with it dropping context time and again. I feel chatgpt is best considered as a supplement to the existing options like stackoverflow but cannot be blindly trusted. That being said it is still a great upgrade from stackoverflow and may be in future be more accurate. Overall a positive experience.*”
	Promptness	“*It was very useful, there was a bit of a learning curve as it took a few attempts to understand what exactly to prompt it to give you the information you want to know. I had used it before so thats why I understood how to problem solve in that way but chat GPT its self doesn’t really suggest other ways to prompt if it doesnt get enough to give an answer.*”
	Accuracy	“*GPT is not primarily designed for debugging but can assist in identifying syntax errors and offering debugging strategies. It can also suggest alternative solutions and offer insights into the program flow and potential issues. However, **its suggestions may not always be accurate**, and it **may not have a deep understanding of complex code structures, making it less effective for identifying all types of errors**.*”
	Resolution	“*While ChatGPT was able to identify issues in the code, **it did not always provide suggestions for how to fix them**. This meant that **developers still needed to have a good understanding of coding** best practices in order to address the issues identified by ChatGPT.*”
	Verification	“*ChatGPT is perfect as a **code-companion but not as a replacement**. The reason for this is because ChatGPT is unable to find context of the applications of code. This makes it more prone to missing logical bugs like the fall-through of a switch-case statement. **Having a human-in-the-loop** who is able to understand the applications and has a good idea of the overall structure of the project **is much more desirable**.*”

although a fix was applied, the PMD analyzer would still detect the problem as present. ChatGPT re-edits the code when prompted; sometimes, it will improve its response. However, when given the same prompt multiple times, it will generate the same response with alteration after many attempts. Therefore, students can use ChatGPT as a complementary tool alongside traditional static analysis tools to improve the efficiency of their software development process.

Lesson #2: ***Limited understanding of the broader context of the codebase.*** While ChatGPT was informative about giving context and definitions of errors and could provide specific recommendations for PMD violations, it has a limited understanding of the broader context and the entire codebase, which could lead to missing dependencies and codependencies. Due to this limitation, there have been instances where ChatGPT makes suggestions based on misunderstandings or false assumptions about the code. This assignment reveals this limitation in a practical manner to help students better understand the mechanics of the model, rather than treating it as a black box that autogenerates acceptable answers for any given query. In this experiment, not only students could not reach an acceptable answer for some queries, they also experienced how suggested code changes can be even problematic, as seen in Figure 6. Students experienced various negative side-effects of the autogenerated code. For example, in some cases, ChatGPT provided code has introduced compiler errors. Also, some students reported that the suggested refactoring did push some tests to fail, despite the fact that refactoring is supposed to preserve the system’s internal behavior. Furthermore, ChatGPT suggested a fix for the ‘Best Practices’ category; but its solution has increased the cyclomatic complexity of the method, probably due to the lack of context and explainability of the code snippets. This observation is consistent with a previous study [6] that found that bad warning messages and no suggested fixes are some of the pain points reported by industrial software developers when using program analyzers.

📌 **Lesson #3:** ***Verify ChatGPT's responses with human expertise.*** While ChatGPT can provide some help in debugging, it should not be relied on as the only debugging tool. ChatGPT is perfect as a code companion, but not a replacement. The reason for this is that ChatGPT cannot find the context of code applications. Having a human-in-the-loop who can understand the applications is much more desirable. Thus, experienced developers and specialized debugging tools remain essential for thorough and effective debugging.

📌 **Lesson #4:** ***Automate the debugging and bugfixing process.*** ChatGPT is an effective tool for programmers seeking assistance. Still, providing clear and detailed questions is essential to get the most accurate and helpful responses. It has helped students debug and clarify what each code segment did and explain how it performed each bugfix. Thus, the process is significantly shortened. Yet, in a few cases, it required some experimentation before coming up with a set of questions to accomplish the task. Additionally, it suggested changes that caused compiler errors, *e.g.,* renaming a variable in a field declaration or fixing a different issue in the code instead of the one given to analyze. Furthermore, ChatGPT tends to view PMD-related stylistic or best practice issues as not problematic and is too light handed in explaining the potential issues with stylistic/best practice problems in the source code. In the real world, this can lead to a student developing bad coding habits, ultimately detrimental to the productivity of the individual and the team they belong to. some of the flaws were observed with inexplicable fixes or suggestions.

📌 **Lesson #5:** ***Improve ChatGPT's accuracy and effectiveness.*** ChatGPT was useful in solving a few PMD rule violations and provided good recommendations for resolving them. However, it cannot be determined as a PMD rule violation in a few cases, even though the code snippet is provided with context. For a few violations, it answered as 'may', providing it satisfying given conditions since it only provided the code snippet and not the surrounding code. We believe that if ChatGPT is trained properly for the specific ruleset, it can be used to quickly eliminate PMD rule violations. Yet, one cannot highly rely only on ChatGPT; some level of understanding and knowledge is required by the developer to make the appropriate correction, since it sometimes generates code that produces compile errors.

📌 **Lesson #6:** ***Handle complex code and errors.*** The effectiveness of ChatGPT is closely related to the quality of its training data set. According to the students' comments, ChatGPT provides high-level and valuable suggestions about the bugs. For instance, it is useful in fixing non-pressing matters such as code style, adding comments, making the if statements more logical, etc. On the other hand, it would sometimes misunderstand the error the student was trying to indicate or would be confused by the fact that the input gave it a code excerpt rather than the full code, which was too large to provide. Its corrections sometimes also misunderstood the purpose of an excerpt and would slightly alter the logic. Therefore, the more complex the given code input, the more likely the code output will not work properly.

V. Related Work

Biswas [4] discussed an overview of ChatGPT as a language model developed by OpenAI that offers a wide range of computer programming capabilities. These capabilities include code completion, correction, prediction, error fixing, optimization, document generation, chatbot development, text-to-code generation, and technical query answering. The author highlighted the ability of ChatGPT to provide explanations and guidance to users and concluded that it is a powerful tool for the programming community. Haque and Li [11] explored the capabilities of ChatGPT as a debugging tool and the best practices for integrating it into the software development workflow. Their findings show that ChatGPT is a useful tool for debugging but should be used cautiously in software development. Ma *et al.* [15] performed a study evaluating ChatGPT capabilities and limitations in software engineering from three aspects: 1) syntax understanding, 2) static behavior understanding, and 3) dynamic behavior understanding. The authors concluded that ChatGPT possesses capabilities akin to an Abstract Syntax Tree (AST) parser, ChatGPT is susceptible to hallucination when interpreting code semantic structures and fabricating nonexistent facts, which underscore the need to explore methods for verifying the correctness of ChatGPT outputs to ensure its dependability in software engineering tasks.

VI. Threats to Validity

External Validity. Concerning the generalizability of our results, our study involves 102 submissions. Although we obtained valuable information and performed an accurate analysis, the results may not represent the larger student population that uses static analysis tools and ChatGPT to fix issues. Additionally, our analysis was performed on mature open-source Java projects that varied in size, contributors, and number of commits. However, we cannot claim the generality of our observations to projects written in other programming languages or belonging to different ecosystems. More research is needed on even more projects to mitigate this threat. Since ChatGPT training contains a large corpus of source code, from GitHub and StackOverflow, there is a significant chance of a data leakage problem, *i.e.,* the proposed fixes were previously seen in the training set, and so the fixes were previously memorized. However, the projects used actually contain some of these warnings in their current versions, which means that ChatGPT may have experienced these fixes in other projects, and its current decisions are based on inference. In addition, ChatGPT performance was not uniform across categories and underperformed for design-level issues. Finally, we recommend that students use the free version of ChatGPT (currently 3.5), but we did not control who uses the premium service, which features GPT-4.

Internal and Construct Validity. As for the completeness and correctness of our interpretation of open-ended comments about the tool, we did not extensively discuss all comments because some of them are open to various interpretations and we need further follow-up interviews to clarify them. Additionally, to avoid personal bias during manual analysis, each step of the manual analysis was conducted by two authors until a consensus was reached. The choice of PMD, as a static analysis tool, may introduce some bias to the way these issues are detected, especially since the detection of bad programming practices and code smells is known to be subjective [5], [16], [7], [2], [9], [1]. Also, students may have had a different experience if another tool was selected in this assignment. We chose PMD as it is one of the popular state-of-the-art tools, but in future work, we plan on trying other static analysis tools to see if they can also reach this level of satisfaction. In addition, any training for students may induce training bias. To mitigate it, both PMD and ChatGPT training were independent. During ChatGPT training, students were exposed to prompts related to software quality principles, that they can relate to, but none of the queries was targeting the type of issues raised by PMD.

VII. Conclusion and Future Work

Understanding the practice of reviewing code to improve the quality is of paramount importance for education. Although modern code review is widely adopted in open-source and industrial projects, the relationship between using LLMs such as ChatGPT and how students perceive it during code analysis remains unexplored. In this study, we conducted a quantitative and qualitative study to explore the effectiveness of PMD and ChatGPT in familiarizing students with improving source code, by i) detecting code issues and antipatterns and ii) implementing fixes for their correction. The paper develops a culture of reviewing and patching unknown codes.

Our results reveal several types of static analysis tools that students should pay more attention to during code review; reviewing design-related changes takes longer to complete compared to other changes, and students rated some aspects of ChatGPT positively while also providing valuable ideas for future model improvement. For future work, we plan on using other static analysis tools that will complement and validate our current study to provide the software engineering community with a more comprehensive view of the use of static analysis tools to engage students with software quality improvement from the educator and student perspectives. Moreover, we plan to investigate students' understanding of code review practice using various real-world applications in a semester-long course project.

References

[1] E. A. AlOmar, S. A. AlOmar, and M. W. Mkaouer. On the use of static analysis to engage students with software quality improvement: An experience with pmd. 2023.

[2] F. Arcelli Fontana, M. V. Mäntylä, M. Zanoni, and A. Marino. Comparing and experimenting machine learning techniques for code smell detection. *Empirical Software Engineering*, 21(3):1143–1191, 2016.

[3] G. Bavota, A. De Lucia, M. Di Penta, R. Oliveto, and F. Palomba. An experimental investigation on the innate relationship between quality and refactoring. *Journal of Systems and Software*, 107:1–14, 2015.

[4] S. Biswas. Role of chatgpt in computer programming.: Chatgpt in computer programming. *Mesopotamian Journal of Computer Science*, 2023:8–16, 2023.

[5] S. Bryton, F. B. e Abreu, and M. Monteiro. Reducing subjectivity in code smells detection: Experimenting with the long method. In *2010 Seventh International Conference on the Quality of Information and Communications Technology*, pages 337–342. IEEE, 2010.

[6] M. Christakis and C. Bird. What developers want and need from program analysis: an empirical study. In *Proceedings of the 31st IEEE/ACM international conference on automated software engineering*, pages 332–343, 2016.

[7] D. Di Nucci, F. Palomba, D. A. Tamburri, A. Serebrenik, and A. De Lucia. Detecting code smells using machine learning techniques: are we there yet? In *2018 ieee 25th international conference on software analysis, evolution and reengineering (saner)*, pages 612–621. IEEE, 2018.

[8] L. Fei-Fei, R. Fergus, and P. Perona. One-shot learning of object categories. *IEEE transactions on pattern analysis and machine intelligence*, 28(4):594–611, 2006.

[9] F. A. Fontana, P. Braione, and M. Zanoni. Automatic detection of bad smells in code: An experimental assessment. *J. Object Technol.*, 11(2):5–1, 2012.

[10] R. M. French. Catastrophic forgetting in connectionist networks. *Trends in cognitive sciences*, 3(4):128–135, 1999.

[11] M. A. Haque and S. Li. The potential use of chatgpt for debugging and bug fixing. *EAI Endorsed Transactions on AI and Robotics*, 2(1):e4–e4, 2023.

[12] N. Imtiaz, B. Murphy, and L. Williams. How do developers act on static analysis alerts? an empirical study of coverity usage. In *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*, pages 323–333. IEEE, 2019.

[13] B. A. Kitchenham and S. L. Pfleeger. Personal opinion surveys. In *Guide to advanced empirical software engineering*, pages 63–92. Springer, 2008.

[14] C. Li, E. Chan, P. Denny, A. Luxton-Reilly, and E. Tempero. Towards a framework for teaching debugging. In *Proceedings of the Twenty-First Australasian Computing Education Conference*, pages 79–86, 2019.

[15] W. Ma, S. Liu, W. Wang, Q. Hu, Y. Liu, C. Zhang, L. Nie, and Y. Liu. The scope of chatgpt in software engineering: A thorough investigation. *arXiv preprint arXiv:2305.12138*, 2023.

[16] M. V. Mäntylä and C. Lassenius. Subjective evaluation of software evolvability using code smells: An empirical study. *Empirical Software Engineering*, 11(3):395–431, 2006.

[17] S. A. Mengel and V. Yerramilli. A case study of the static analysis of the quality of novice student programs. In *The proceedings of the thirtieth SIGCSE technical symposium on Computer science education*, pages 78–82, 1999.

[18] M. M. Rahman and Y. Watanobe. Chatgpt for education and research: Opportunities, threats, and strategies. *Applied Sciences*, 13(9):5783, 2023.

[19] D. Singh, V. R. Sekar, K. T. Stolee, and B. Johnson. Evaluating how static analysis tools can reduce code review effort. In *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 101–105. IEEE, 2017.

[20] R. Van Solingen, V. Basili, G. Caldiera, and H. D. Rombach. Goal question metric (gqm) approach. *Encyclopedia of software engineering*, 2002.

[21] Y. Wang, Q. Yao, J. T. Kwok, and L. M. Ni. Generalizing from a few examples: A survey on few-shot learning. *ACM computing surveys (csur)*, 53(3):1–34, 2020.

[22] J. White, S. Hays, Q. Fu, J. Spencer-Smith, and D. C. Schmidt. Chatgpt prompt patterns for improving code quality, refactoring, requirements elicitation, and software design. *arXiv preprint arXiv:2303.07839*, 2023.

[23] Y. Xian, B. Schiele, and Z. Akata. Zero-shot learning-the good, the bad and the ugly. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4582–4591, 2017.

[24] X. Yang, Z. Yu, J. Wang, and T. Menzies. Understanding static code warnings: An incremental ai approach. *Expert Systems with Applications*, 167:114134, 2021.

[25] R. Yedida, H. J. Kang, H. Tu, X. Yang, D. Lo, and T. Menzies. How to find actionable static analysis warnings: A case study with findbugs. *IEEE Transactions on Software Engineering*, 2023.