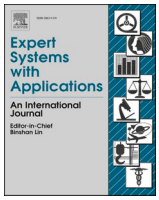
Expert Systems With Applications 249 (2024) 123602

Contents lists available at ScienceDirect

Expert Systems With Applications

journal homepage: www.elsevier.com/locate/eswa



Exploring ChatGPT's code refactoring capabilities: An empirical study

Kayla DePalma, Izabel Miminoshvili, Chiara Henselder, Kate Moss, Eman Abdullah AlOmar *

Stevens Institute of Technology, Hoboken, NJ, USA

A B S T R A C T

ChatGPT has shown great potential in the field of software engineering with its ability to generate code. Yet, ChatGPT's ability to interpret code has been deemed unreliable and faulty, which causes concern for the platform's ability to properly refactor code. To confront this concern, we carried out a study to assess ChatGPT's abilities and limitations in refactoring code. We divided the study into three parts: if ChatGPT can refactor the code, if the refactored code preserves the behavior of the original code segments, and if ChatGPT is capable of providing documentation for the refactored code to provide insights into intent, instructions, and impact. We focused our research specifically on eight quality attributes to use when prompting ChatGPT to refactor our dataset of 40 Java code segments. After collecting the refactored code segments from ChatGPT, as well as data on whether the behavior was preserved, we ran the refactored code through PMD, a source code analyzer, to find programming flaws. We also tested ChatGPT's accuracy in generating documentation for the refactored code and analyzed the difference between the results of each quality attribute. We conclude that ChatGPT can provide many useful refactoring changes that can improve the code quality which is crucial. ChatGPT offered improved versions of the provided code segments 39 out of 40 times even if it is as simple as suggesting clearer names for variables or better formatting. ChatGPT was able to recommend numerous options ranging from minor changes such as renaming methods and variables to major changes such as modifying the data structure. ChatGPT's strengths and accuracy were in suggesting minor changes because it had difficulty addressing and understanding complex errors and operations. Although most of the changes were minor, they made significant improvements because converting loops, simplifying calculations, and removing redundant statements have a crucial effect on runtime, memory, and readability. However, our results also indicate how ChatGPT can be unpredictable in its responses which threatens the reliability of ChatGPT. Asking ChatGPT the same prompt often yields different results, so some outputs were more accurate than others. This makes it difficult to fully access ChatGPT's capabilities due to its variation and inconsistency. Due to ChatGPT's limitations of its reliance on its data set, it lacks understanding of the broader context so it may occasionally make errors and suggest alternations that are neither applicable nor necessary. Overall, ChatGPT has proved to be a beneficial tool for programming as it is capable of providing advantageous suggestions, even if it is on a small scale. However, human programmers are still needed to oversee these changes and determine their significance. ChatGPT should be used as an aid to programmers since we cannot completely depend on it yet.

1. Introduction

The effectiveness of a software system hinges on its ability to uphold high-quality design amidst continuous evolution. However, managing software growth while continually enhancing its functionality presents a formidable challenge, often constituting up to 75 % of total development efforts. Refactoring emerges as a pivotal practice in addressing this challenge. Refactoring, defined as the art of restructuring software design without altering its functionality (Fowler, 2018; Al Dallal & Abdin, 2017), gained prominence through (Fowler, 2018), who identified 72 refactoring types and furnished examples of their application in their catalog. This essential software maintenance activity, performed by developers for a variety of reasons (Silva et al., 2016; Kim et al., 2012), offers insights into how developers manage code across different development phases and over extended periods.

Refactoring code is a vital part of software design as it is aimed to improve the overall structure and cleanliness of the software, making the code more efficient (AlOmar et al., 2021; Fowler, 2018). With the invention of ChatGPT, the possibilities in software engineering, specifically with refactoring code, are much greater considering the efficiency and wide scope of abilities that ChatGPT has. ChatGPT is a generative AI that allows users to enter a prompt and receive human-like text, images, videos, etc., responses. It works using the generative pre-trained transformer to use algorithms that identify patterns with data sequences. Because refactoring is such an essential part of software engineering (AlOmar et al., 2021; Kim & Ernst, 2007; Mens & Tourwé, 2004; Murphy-Hill et al., 2011; Silva et al., 2016), the abilities of ChatGPT are monumental in terms of software engineering and refactoring code.

Therefore, studies have been conducted on ChatGPT's abilities regarding software engineering as a whole. Ma et al. (Ma et al., 2023) specifically investigate its ability to comprehend code syntax and semantic structures because despite ChatGPT being widely used in

* Corresponding author.
E-mail addresses: kdepalma@stevens.edu, imiminos@stevens.edu, chenseld@stevens.edu (C. Henselder), kmoss@stevens.edu (K. Moss), ealomar@stevens.edu (E.A. AlOmar).

https://doi.org/10.1016/j.eswa.2024.123602
Received 7 August 2023; Received in revised form 29 February 2024; Accepted 1 March 2024
Available online 6 March 2024

software engineering tasks, the capabilities of the platform and the depth of its understanding regarding both syntax and semantics are unknown. The survey concluded that ChatGPT does excel at understanding code syntax, but overall has a difficult time interpreting code semantic structures. ChatGPT often fabricates non-existent facts, which leads to doubts regarding its dependability in the field of software engineering. Sun et al. (Sun et al., 2023) investigate ChatGPT's ability in automatic code summarization. By measuring the quality of the comments created using three popular metrics, the study concludes that ChatGPT's performance in summarizing code is significantly worse than the three established state-of-the-art code summarization models, despite the platform showing some advantages for software development. While the studies explore both the limitations and capabilities of ChatGPT from a software engineering perspective, they do not explore its abilities in refactoring code. By researching ChatGPT's abilities to refactor code, there is the opportunity for great potential in optimizing both the quality of code, as well as productivity concerning the time and cost it takes to conduct the refactoring. Furthermore, ChatGPT has the potential to revolutionize software engineering as it has capabilities that other AIs lack. ChatGPT's ability to remember multi-message chats and have human-like conversations is extraordinary. Therefore, studying ChatGPT's capabilities is crucial because it is a new tool that can lead to additional significant advancements in software engineering. ChatGPT is helping technology evolve and it is important to analyze its effect. By determining both the abilities and limitations of using ChatGPT to refactor code, the software engineering community can enhance their under- standing and utilization of the platform by gaining valuable insights from our research.

To address the lack of research regarding ChatGPT's abilities to refactor code, this paper aims to test the platform's ability to refactor code and measure its accuracy and consistency when doing so. In particular, our objective is to use prompt engineering to create formulas for prompts to both refactor the code and then test whether the behavior of the original code is preserved after ChatGPT's refactoring. Prompts are to be judged on the correctness, accuracy, and consistency of their responses. We drive our study using the following research questions:

- RQ1: Does ChatGPT demonstrate effectiveness in performing code refactoring?
- RQ2: Does ChatGPT maintain the functionality of the refactored code?
- RQ3: Is ChatGPT capable of generating comments or documentation for the refactored code segments, providing insights into intent, instructions, and impact?

To answer our research questions, we test ChatGPT's ability to generate documentation for the refactored code segments and perform a qualitative analysis of how the messages differ between the quality attributes. Focusing on eight different quality attributes, we test how including each one at a time in the prompt will change how ChatGPT will refactor the code, and both the disadvantages and advantages of each quality attribute. With our findings, we are able to develop an understanding of ChatGPT's ability in refactoring code and its applications in software engineering.

Through our data analysis, we discovered that ChatGPT is widely capable of refactoring code segments when prompted to do so, as it failed to refactor an inputted code segment only once out of 320 times. It can provide both large and small-scale improvements to a prompted code segment. ChatGPT is also largely successful in preserving the behavior of the original inputted code segment through the refactored code in its response. In the 320 trials, the behavior was preserved 311 times. Furthermore, we found that ChatGPT can accurately generate documentation for refactored code, regarding goals, refactored changes, and impacts/quality attributes of the refactored code. In the 320 trials, only 10 comments were inaccurate. Thus, ChatGPT proves to be a valuable asset in code refactoring, as it consistently offers precise and rapid solutions when presented with a code segment. It is important to note that while we do not advocate relying solely on ChatGPT for code refactoring, it can serve as a helpful aide for implementing simple alterations and initiating the first steps of the refactoring process, thereby conserving developers' time and energy to focus on more complex modifications.

To summarize, this paper makes the following key contributions:

- Novel Refactoring Insights: Our research provides unique insights into how developers interact with ChatGPT for code refactoring. By analyzing a dataset of ChatGPT prompts and responses, we uncover patterns and trends that shed light on the effectiveness and challenges of the tool.
- Ethical Implications: While assessing ChatGPT's performance, we investigate the ethical considerations associated with AI-driven code refactoring. This includes thoroughly examining biases, privacy concerns, and potential risks developers should know when incorporating such tools into their workflows.
- Practical Recommendations: The paper not only evaluates ChatGPT but also provides practical recommendations for developers, addressing ways to enhance collaboration with AI models in the context of code refactoring. These recommendations are drawn from our analysis of developer-ChatGPT interactions.

These contributions collectively advance the understanding of AI assistance in code refactoring, providing valuable insights for researchers and practitioners. To further replicate and extend our study, we provide our comprehensive experiments package[1], which contains the code segments, analyzed data, and survey questions.

The remainder of this paper is organized as follows. In Section 2, we give the design of our empirical study, mainly with regard to the utilization of the dataset and quantitative and qualitative analysis. Section 3 presents the study results while further discussing our findings and takeaways in Section 4. Section 5 reports limitations of our experiments followed by the ethical implication of using ChatGPT in code refactoring. In Section 7, we report future work of our study. Section 8 enumerates the previous related studies, before concluding the paper in Section 9.

2. Methodology

For this research, we began with a dataset containing 40 Java files. All files contained only one class and were between 20 and 50 lines long. This diverse dataset was taken from an existing study regarding the scope of ChatGPT in software engineering by (Ma et al., 2023). The 40 Java files served as the foundation of our research for all three research questions. For every Java code segment within the dataset, we accompanied it with a well-formulated prompt to ask ChatGPT to refactor it. To measure the effectiveness of ChatGPT's refactoring techniques, we conducted an analysis of the refactored code segments using PMD, aiming to identify any coding violations and errors.

2.1. Prompt engineering

As demonstrated in Fig. 2.1, we used prompt engineering to create prompts for our research questions. Research Question 1 consisted of providing ChatGPT with a code segment and asking it to refactor the code to improve one of the eight quality attributes shown in Fig. 2.1. Research Question 2 used the refactored code from RQ1 and determined whether the behavior was preserved between the original and refactored code segments. To further analyze ChatGPT's refactored code segments we used the PMD tool to check for any code violations present within the code segments (Kim & Ernst, 2007; Liu et al., 2018; Plosch et al., 2008;

[1] https://sites.google.com/stevens.edu/chatgptdataanalysis/home.

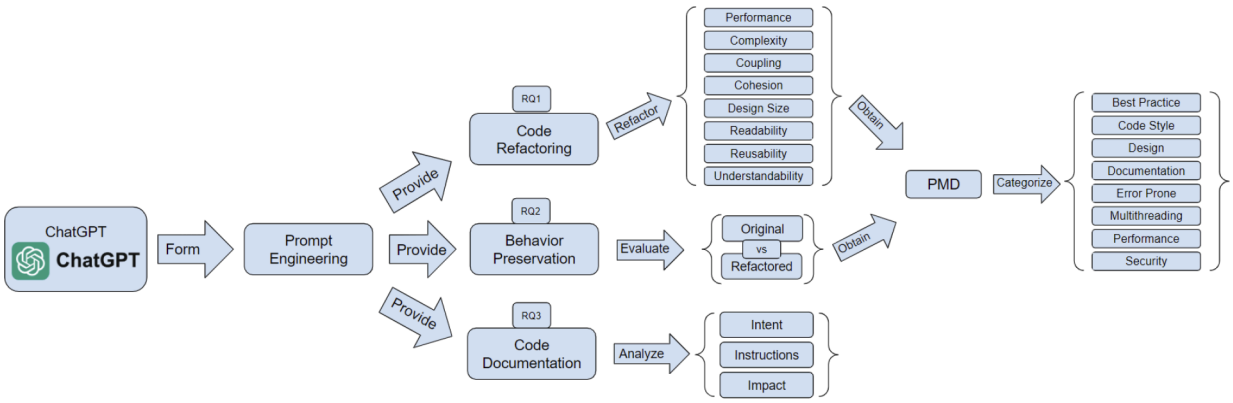


Fig. 2.1. Approach overview.

Romano et al., 2022). The PMD violations were categorized into the categories indicated in Fig. 2.1. Research Question 3 focused on analyzing ChatGPT's capabilities in delivering documentation for the refactored code segment produced in RQ1. This documentation included a detailed description of its intent, instructions, and impact.

After prompt engineering for all three research questions, we decided that the most consistently efficient way was trial and error. We began with basic prompts that portrayed a good foundation to achieve our goals. Using the initial prompt, we inputted it into ChatGPT 3.5 along with one of the code segments and studied the response to determine its strengths and weaknesses. We then questioned what we could add or take out of the prompt to get a clearer and more useful answer, and we created a new version of the past question based on our findings from the last response. With every new prompt we created, we submitted it to ChatGPT with approximately 5 different code segments to also determine the consistency of the answers that prompt would generate. With the analysis of every prompt we had considered, we came together as a group to discuss which one generated the most consistent and accurate answers that were fitting to the goal of our research.

Picking appropriate prompts to fit each research question was very time-consuming because it was a sensitive process. Small changes in a prompt, such as using acronyms instead of the full name or the addition of a colon, created major differences in the responses received from ChatGPT. Realizing that the smallest details of the prompt were still extremely important helped us tremendously in choosing which prompt was the best. This allowed us to recognize minor differences in the responses that were extremely important as well, which specifically helped with the second research question because we noticed the possibility of introducing bias. Overall, observing and manipulating the minuscule changes during the prompt engineering process led to the biggest differences in ChatGPT's responses and allowed our group to find the most efficient and accurate prompts to use for our research. The preceding explanation offers a general overview of our prompt development approach. However, in the following sections, we provide a thorough description of our prompt construction process for each research question. Our first research question investigates the proficiency of ChatGPT in code refactoring: "Can ChatGPT effectively refactor code?" Meanwhile, our second research question explores ChatGPT's capability to maintain the behavior of refactored code: "Can ChatGPT preserve the behavior of the refactored code?" Lastly, our third research question analyzes ChatGPT's efficiency in providing code documentation: "Can ChatGPT provide comments/documentation for the refactored code segments to describe intent, instructions, and impact?".

2.1.1. RQ1: Does ChatGPT demonstrate effectiveness in performing code refactoring?

For this research question, our objective was to utilize ChatGPT to refactor Java code based on a given prompt. The null hypothesis is defined by H0: Refactoring suggestions provided by ChatGPT do not lead to improvements in code quality. Thus, the alternative hypothesis indicates that H1: Refactoring suggestions provided by ChatGPT result in higher code quality compared to manual refactoring efforts. We created one prompt to use for eight different quality attributes. The quality attributes include performance, complexity, coupling, cohesion, design size, readability, reusability, and understandability (Al Dallal & Abdin, 2017; AlOmar et al., 2019; Bavota et al., 2015; Chaparro et al., 2014; Chávez et al., 2017; Fernandes et al., 2020; Kaur & Singh; 2017; Moser et al., 2006; Pantiuchina et al., 2020; Stroggylos & Spinellis, 2007). To ensure clarity and simplicity, we collectively agreed to begin the prompt with the phrase "With no explanation," indicating that we desired ChatGPT's output to solely consist of the refactored code. Additionally, we specifically mention that the provided code is written in the Java programming language to prevent potential misidentification by ChatGPT, which could lead to incorrect syntax modifications during refactoring. To indicate the start of the code segment, we found it beneficial to conclude the prompt with a colon. Taking these considerations into account, we formulated four prompt variations and evaluated their performance with numerous Java files. The prompts were as follows:

-With no explanation refactor the Java code:

-With no explanation refactor the Java code to improve its quality:

-With no explanation refactor the Java code to improve its [quality attribute]:

-With no explanation refactor the Java code to improve its quality and [quality attribute]:

The first two prompts yielded similar outputs, as ChatGPT predominately made the same modifications. These changes primarily encompassed basic edits focused on enhancing method and variable names, improving readability, and adhering to Java conventions. Although these modifications were generally minor, some appeared more crucial than others. The similarity between ChatGPT's interpretation when prompted to refactor the code or refactor to improve its quality may stem from ChatGPT's perception of the broad contextual nature of the term "quality."

The third prompt resulted in more extensive modifications, emphasizing alterations to data structures and optimization of method implementations and loop structures. These changes were not observed in the previous prompts. However, this prompt lacked improvements targeting code readability and Java conventions, which were present in the first two prompts. Notably, when prompted to refactor the code to improve performance, ChatGPT attempted to modify elements of the code that would improve its runtime and memory utilization, rather than solely addressing readability concerns.

Lastly, the fourth prompt resulted in a combination of both minor and substantial edits. While it contained fewer significant modifications compared to the third prompt, it still introduced noteworthy changes

absent in the first two prompts. In addition to the major modifications, there were significant minor-scale edits, which were absent in the third prompt. Thus, by instructing ChatGPT to refactor the code to enhance both quality and another quality attribute, we observed a broader range of changes. Considering the comprehensive coverage of details, encompassing both minor and major aspects, we concluded that the fourth prompt was the most suitable choice to proceed with our research.

However, before finalizing our prompt selection, we deliberated on whether bias would be introduced by including the phrase "to improve [quality attribute]" in comparison to only saying "to improve quality." To address this concern, we conducted extensive research comparing the second and fourth prompts, specifically examining the impact of solely using the term "quality" versus incorporating both "quality and [quality attribute]." We aimed to determine if ChatGPT's behavior would be influenced by the inclusion of the quality attribute.

We discovered that when the prompt only included the term "quality," ChatGPT consistently generated the following default output statement: "Overall, these changes aim to improve readability, follow coding conventions, and use more descriptive names." These changes were generally minor, and it could be expected that the code's behavior will always remain consistent since no substantial modifications were made.

On the other hand, when the prompt incorporated both "quality and [quality attribute]," ChatGPT did not always provide a default output statement. However, it was observed that ChatGPT often generated a variation of the following statement: "Overall, these changes aim to improve the readability, adhere to Java naming conventions, and simplify the logic while maintaining the functionality of the original code." This suggests that by explicitly including quality-related keywords in the prompt, ChatGPT was inclined towards performing more significant modifications. Based on our analysis, our main conclusion was that ChatGPT will do major refactoring if we enforce some quality attribute keywords.

Therefore, we selected the following prompt for our research: "With no explanation refactor the Java code to improve its quality and [quality attribute]: " [include original code here].

In order to evaluate the effectiveness of ChatGPT's refactoring techniques, we utilized Programming Mistake Detector (PMD) to detect multiple violations across various categories in each refactored code segment (Kim & Ernst, 2007; Liu et al., 2018; Plosch et al., 2008; Romano et al., 2022). These categories included Best Practices, Code Style, Design, Documentation, Error Prone, Performance, Multithreading, and Security. The violations identified within these categories have varied levels of severity, spanning from important, urgent, and critical to blocker. In the RQ1 results section, we thoroughly analyze the data obtained from PMD and derive meaningful conclusions.

2.1.2. RQ2: Does ChatGPT maintain the functionality of the refactored code?

For this research question, our primary goal was to evaluate whether the refactored code generated by ChatGPT for RQ1 retained the original behavior of the Java code. The null hypothesis is defined by H0: ChatGPT does not preserve the behavior of the refactored code. Thus, the alternative hypothesis indicates that H1: ChatGPT preserves the behavior of the refactored code. To do this, we would utilize ChatGPT to determine if the original Java code and ChatGPT's refactored code preserved the code's original behavior.

During the experimentation process, we encountered a challenge when attempting to include both the original Java code and the refactored code in a single prompt to ChatGPT, as it often exceeded the prompt's capacity. To address this issue, we devised a two-step approach. First, we instructed ChatGPT to remember the original code segment by providing it in the first message. ChatGPT would then acknowledge that it has stored the code and repeat it back to us. Subsequently, in a separate message, we would write the prompt we formulated and include the refactored code along with it.

In this example, we start the process by instructing ChatGPT to retain our message using the original code, as shown in Fig. 2.2. In Fig. 2.3, ChatGPT acknowledges its memory of the code segment and proceeds to provide a concise summary of the code's components and objectives. Afterward, in Fig. 2.4, we present ChatGPT with our prompt incorporating the refactored code. In response, evident in Fig. 2.5, ChatGPT either confirms or denies the preservation of the code's behavior, along with a thorough explanation. ChatGPT's explanation makes specific references to elements present in both code segments, demonstrating its accurate recollection of the original code. When the message provided is excessively long, a statement is generated to indicate an error as shown in Fig. 2.6.

To ensure an unbiased assessment, we decided to avoid labeling the code segments as original and refactored, which eliminated any implicit relationship between them. Instead, we presented the two code segments independently. A few examples of the prompts we created during our experimentation are listed below.

-Was the behavior of this Java code preserved?

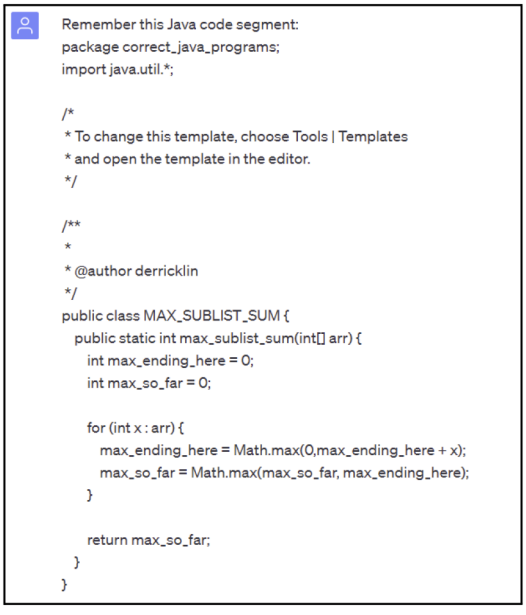
-Does this Java code preserve the behavior of the above code segment why or why not?

-Does this Java code preserve the behavior of the above code segment? Explain in detail why or why not:

The first prompt yielded sufficient results but lacked a detailed explanation to justify ChatGPT's decision on whether the behavior was preserved. We did not want ChatGPT to output solely a yes or no, we wanted to understand its justification. In order to prompt ChatGPT to reveal more information about its decision we decided to ask it why or why not at the end of the prompt. This helped lengthen ChatGPT's output and made it more concise and clear, but some specific and detailed reasoning was still absent.

Example: LCS_LENGTH.java and its refactored counterpart based on refactoring for performance

In this example, by comparing prompt one in Fig. 2.7 and prompt two in Fig. 2.8, it becomes evident that the list of numbers lengthens by two, and each element in the list becomes more detailed. prompt two delves into each element of the list, offering concise descriptions of the modifications between the original and refactored code, typically spanning one to two full sentences. As a result, prompt two provides a more comprehensive response, evaluating multiple aspects of each code



Remember this Java code segment:

```
package correct_java_programs;
import java.util.*;

/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

/**
 *
 * @author derricklin
 */
public class MAX_SUBLIST_SUM {
  public static int max_sublist_sum(int[] arr) {
    int max_ending_here = 0;
    int max_so_far = 0;

    for (int x : arr) {
      max_ending_here = Math.max(0,max_ending_here + x);
      max_so_far = Math.max(max_so_far, max_ending_here);
    }

    return max_so_far;
  }
}
```

Fig. 2.2. First prompt to ChatGPT asking it to remember the original code segment because providing both code segments exceeded the character limit.

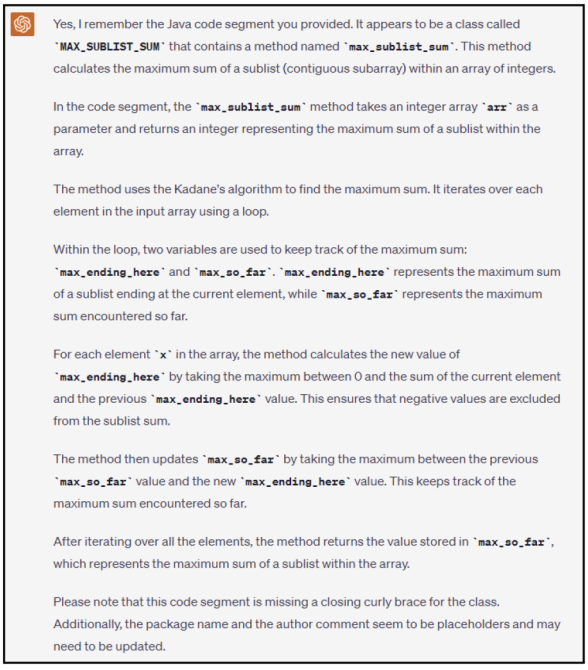


Fig. 2.3. ChatGPT's response to our first prompt summarizing the original code segment we provided to show that it understands it.

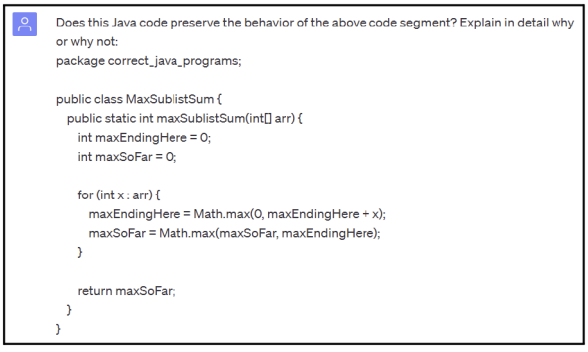


Fig. 2.4. Second prompt to ChatGPT asking if the refactored code segment preserves the behavior of the original code segment shown in Fig. 2.2 and providing the refactored code segment.

segment and effectively explaining the similarities between them. Despite the potential variation in outcomes when asking ChatGPT the same prompt, the comparison between both prompts remains consistent.

Also, in the prompt for research question two, we decided that we should specify that the code segments were written in Java programming language so ChatGPT would be able to correctly identify the syntax and reduce the chance of interpreting it as a different language. In our third prompt, we decided to add the phrase, "Explain in detail," to distinctly instruct ChatGPT to provide a thorough explanation with its output. Out of all three prompts, the third one was the most successful because it resulted in ChatGPT's yes or no answer and a comprehensive list of reasons to justify its claim. For this reason, we decided to continue our research with the third prompt. Since including the two code segments in one message sometimes exceeded the message's capacity, we structured our prompt as follows:

Message 1: Remember this Java code segment: [include original code here].

Message 2: Does this Java code preserve the behavior of the above code segment? Explain in detail why or why not: [include refactored code here].

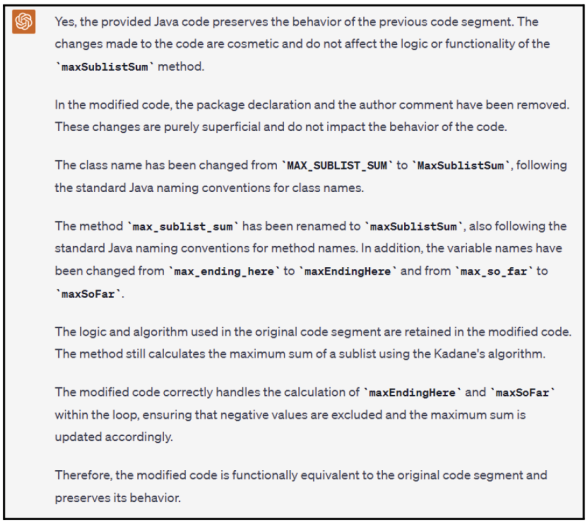


Fig. 2.5. ChatGPT's response to the prompt shown in Fig. 2.4 with a confirmation or denial of the preservation of the code with a detailed explanation of why.

2.1.3. RQ3: Is ChatGPT capable of generating comments or documentation for the refactored code segments, providing insights into intent, instructions, and impact?

In this research question, our main objective was to explore whether ChatGPT could effectively provide documentation for the refactored code segments, describing their intent, instructions, and impact. The null hypothesis is defined by H0: ChatGPT could not effectively provide documentation for the refactored code segments, describing their intent, instructions, and impact. Thus, the alternative hypothesis indicates that H1: ChatGPT could effectively provide documentation for the refactored code segments, describing their intent, instructions, and impact. We defined "intent" as the motivation and goal behind the code changes, "instructions" as the specific refactoring modifications, and "impact" as the effect on code quality.

Initially, we attempted this by only presenting ChatGPT with the refactored code segment from RQ1 in the prompt. However, we quickly noticed that ChatGPT produced more insightful comments when given both the original and refactored code segments. As depicted in Fig. 2.9 ChatGPT's output when only providing the refactored code segment is short and vague. ChatGPT provides three one-sentence descriptions for each of the goals, refactoring changes, and impact, but lacks an in-depth description that explains how it came to that conclusion. This may be because ChatGPT does not have another code segment to compare the refactored code to, so there are not many conclusions it can derive. However, as demonstrated in Fig. 2.10, ChatGPT's output when providing both the original and refactored code is more extensive and descriptive. In addition to the three one-sentence summaries as before, it includes an additional bullet point with a paragraph of detailed explanations for each. This enhancement allows us to gain a deeper understanding of ChatGPT's analysis process and establishes greater trust in its reliability. Therefore, to elicit the most valuable responses, we adjusted our prompt to include both the original code segment and the refactored version.

Similarly to RQ2, we decided to use the strategy of breaking the interaction into two messages. We first asked ChatGPT to remember the original code segment in one message, and in a subsequent message, we presented our prompt along with the refactored code. This strategy helped us overcome the character limit issue and allowed ChatGPT to process the information more effectively.

A few examples of prompts we created during the process are listed below:

-Provide comments in this provided Java code to describe the intent,

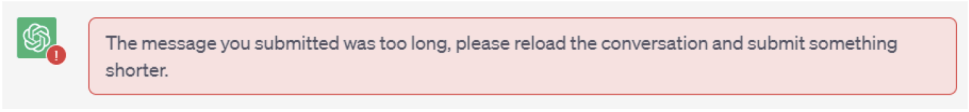


Fig. 2.6. When providing too long of a message, this is the statement outputted to signal an error.

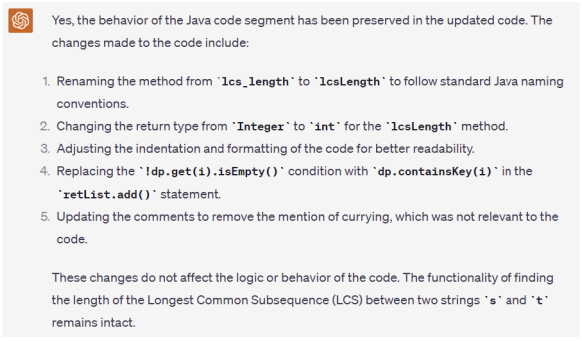


Fig. 2.7. Using prompt one: ChatGPT's output when not asking why or why not.

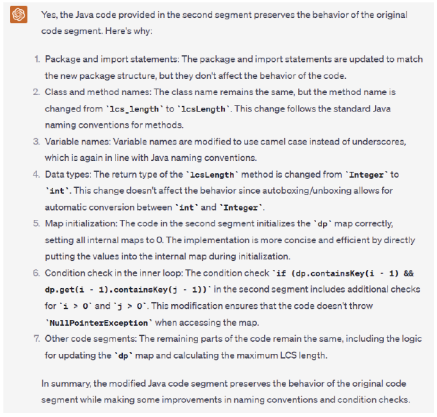


Fig. 2.8. Using prompt two: ChatGPT's output when asking why or why not.

instructions, and impact of quality of the code in relation to the previous code segment:

-Provide docstrings and comments in this provided Java code to describe the intent, refactoring changes, and impact of quality of the code in relation to the previous code segment:

-Provide concise commit messages that describe the overall goal, refactoring changes, and impact of quality of the following code:

In our exploration of the three different prompts, we encountered varying levels of consistency in ChatGPT's responses. Among these prompts, the first one produced the most inconsistent results. This inconsistency arose because ChatGPT interpreted the terms "intent" and "instructions" differently each time, lacking a clear and consistent understanding of these terms. Furthermore, we encountered variations in the type of comments provided by ChatGPT in response to the first prompt. At times, it generated single in-line comments within the code, while other times it offered docstrings for every method or bullet-pointed comments outside the code. Fig. 2.11 displays ChatGPT's response to the first prompt listed above when using the words "intent" and "instructions." Fig. 2.12 also depicts ChatGPT's response to the same prompt used for Fig. 2.11, however, the outputs differ significantly. The output in Fig. 2.11 interpreted describing "intent" to mean adding docstrings and single in-line comments to explain the purpose of the method and if statements within the method. The second output in Fig. 2.12 interpreted "intent" by only adding single in-line comments

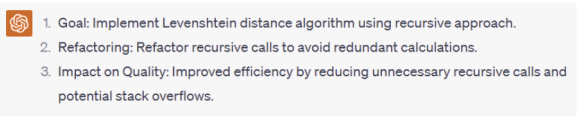


Fig. 2.9. ChatGPT's output when only providing the refactored code in the prompt.

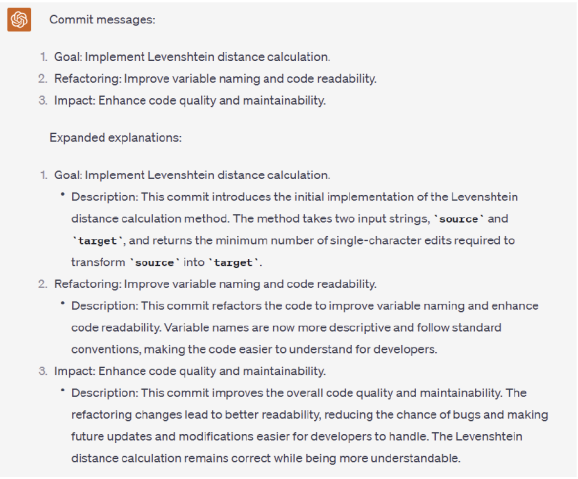


Fig. 2.10. ChatGPT's output when providing both the original and refactored code in the prompt.

that explain the overall goal of the code and the purpose of each line. The response shown in Fig. 2.11 interpreted "instructions" as listing the refactoring changes outside the code segment and only providing a short paragraph for the impact. In Fig. 2.12 the response fails to provide any refactoring changes but instead provides a lengthy list of bullet points about the impact on quality. Therefore, the inconsistency of responses would have a negative impact on our data analysis because they would be difficult to compare.

To address these issues, we proceeded with the second prompt, using the term "refactoring changes" instead of "instructions." This modification led to more consistent responses as ChatGPT understood that it needed to identify the specific refactoring techniques applied to the original code segment to produce the refactored version. Additionally, we attempted to clarify the type of comments by using "docstring and comments" in the second prompt. While this approach did yield docstrings for all methods and one-line comments, we realized this might not be the most suitable format for the desired documentation.

As a result, for the third prompt, we asked ChatGPT to provide "concise commit messages," commonly used by programmers to explain changes made to a project. We also replaced "intent" with "goal," as ChatGPT seemed to interpret it more accurately. Fig. 2.13 depicts ChatGPT's output when using the third prompt but with the word "intent," and Fig. 2.14 illustrates ChatGPT's output when using the third prompt with the word "goal." Upon examining both outputs, it becomes evident that the term "intent" occasionally results in ChatGPT's response not precisely indicating the identified intent of refactoring. This can create confusion for us in determining the intended goal and harder for us to evaluate ChatGPT's ability to document it. Conversely, "goal"

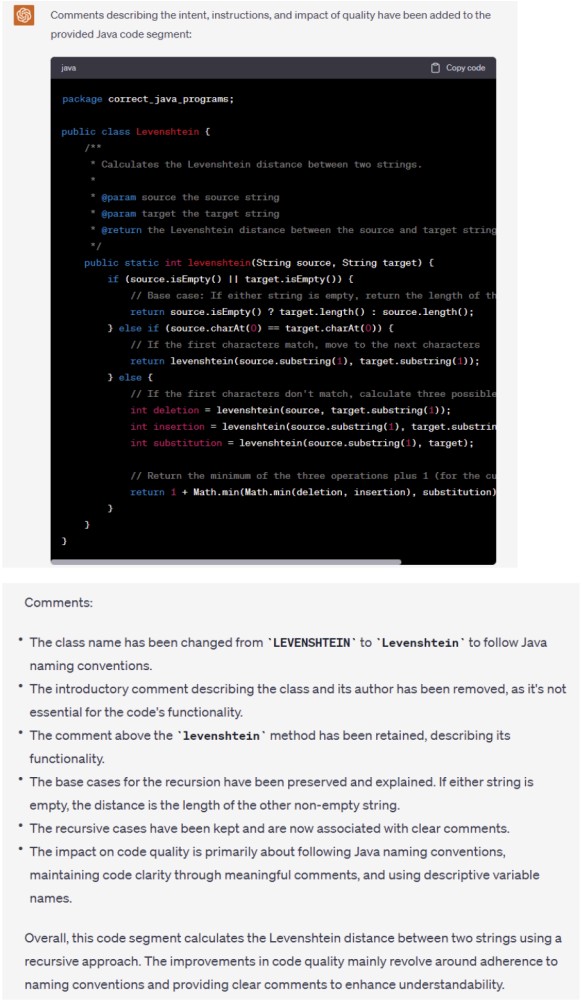


Fig. 2.11. First example of ChatGPT's output when using the words "intent" and "instructions" in the prompt.

Fig. 2.12. Second example of ChatGPT's output when using the words "intent" and "instructions" in the prompt.

prompts ChatGPT to distinctly outline the goal, refactoring change, and the consequent impact. Additionally, the inclusion of "goal" in the prompt induces ChatGPT to provide a longer list, confirming ChatGPT's adequate understanding of the refactoring process. Thus, we chose to utilize the word "goal" because ChatGPT responded with more succinct and organized answers, allowing us to better evaluate its understanding of documentation.

By adding "overall" in the prompt, ChatGPT was prompted to focus on the code segment as a whole and its broader context. This final prompt produced the best results, with ChatGPT consistently interpreting commit messages and providing the documentation explanations we were seeking. Therefore, the prompt we chose to continue our research for RQ3 with was as follows:

Message 1: Remember this Java code segment: [include original code here].

Message 2: Provide concise commit messages that describe the overall goal, refactoring changes, and impact of quality of the following code: [include refactored code here].

Example: With Levenshtein.java and its refactored counterpart based on refactoring for performance.

2.1.4. Overview of methodology

To summarize the steps of our study, for research question 1 we used to prompt "With no explanation refactor the Java code to improve its quality and [quality attribute]: [include original code here]." For each of the eight quality attributes of performance, complexity, coupling,

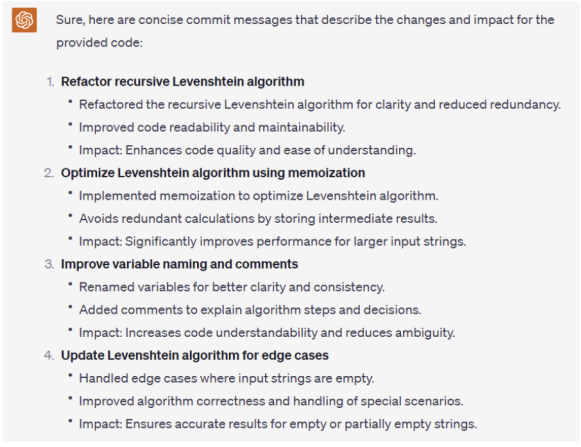


Fig. 2.13. ChatGPT's response when using the word "intent" in the prompt instead of "goal".

cohesion, design size, readability, reusability, and understandability we accompanied this prompt with the original code of each of the 40 Java code segments and inputted it into ChatGPT. Therefore, the results consisted of 320 refactored Java code segments. For each of the

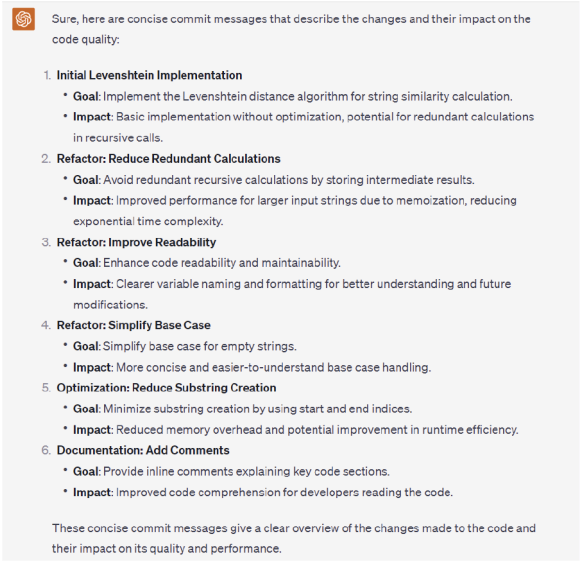


Fig. 2.14. ChatGPT's response when using the word "goal" in the prompt instead of "intent".

refactored code segments, we documented whether ChatGPT was able to successfully refactor them or not. We considered any change to be successful refactoring and if ChatGPT stated that it could not implement any modifications to the code then it was unsuccessful refactoring. We saved each of the refactored code segments outputted by ChatGPT and used PMD to identify code violations and errors in each of the refactored code segments. We used the IDE IntelliJ IDEA and downloaded the PMD software to do this. After running PMD, we recorded all the violations on a spreadsheet so we could analyze the outcomes. We also used the refactored code to answer our second research question where we first used the prompt "Remember this Java code segment: [include original code here]" and followed up with the prompt "Does this Java code preserve the behavior of the above code segment? Explain in detail why or why not: [include refactored code here]." We provided ChatGPT with this prompt 320 times, one for each of the 320 refactored Java files. For each of the 320 code segments, we recorded if ChatGPT identified if the original code segment's behavior was preserved in the refactored code. Then, for our third research question, we asked ChatGPT to "Remember this Java code segment: [include original code here]" and followed it with the message, "Provide concise commit messages that describe the overall goal, refactoring changes, and impact of quality of the following code: [include refactored code here]." We used this prompt 320 times, one for each of the 320 refactored code segments so ChatGPT provided 320 code segments with comments and documentation. For each of the 320 documented code segments, we reported if ChatGPT's output identified accurate goals and motivations behind the refactoring changes, provided accurate refactoring changes and instructions that were implemented to improve the original code to the refactored code, and provided accurate impacts of the refactoring had on code quality. We logged if ChatGPT's response had any of these three criteria missing and determined if ChatGPT's answers were accurate based on our knowledge of the original and refactored code segments and changes. We also kept track of the quality attributes ChatGPT distinguished as improved when providing documentation about impact based on the original and refactored code segments. After collecting the data for all three research questions, we created tables and pie charts to derive conclusions and assess our research questions to ultimately determine ChatGPT's refactoring capabilities.

3. Results and discussion

3.1. RQ1: Does ChatGPT demonstrate effectiveness in performing code refactoring?

Overall, ChatGPT does have the ability to refactor code well. There were some instances where the refactored code generated by ChatGPT had bugs, however, it was widely successful in its ability to refactor the Java code segments. Below, the results regarding the refactored code are discussed and analyzed with respect to each quality attribute, separately.

3.1.1. Performance

After ChatGPT refactored the 40 files of code segments for quality and performance improvement, we noticed little to no change between the refactored and the original code. The little changes we noticed were modifications such as changing an ArrayList declaration to a List declaration, changing an if-statement to a while loop, if it was already a while loop, then it would add a case that was not accounted for, and it would change a for each loop into a for loop. Lastly, if addLast() or addFirst() were used, ChatGPT would just use add() instead. Changing an ArrayList to a List does help with performance since going through a List is faster. Additionally, changing an if-statement to a while loop does not affect performance unless the code is being run a lot of times, then a while loop is better. However, the other changes have no effect on performance.

Once the refactoring of the 40 files was finished, we put the refactored files through the PMD tool. Through this tool, we identified a total of 505 violations which consisted of 6 blocker violations, 3 critical violations, 6 important violations, and 490 urgent violations. The severity of these violations begins with blocker violations being the most severe to errors being the least. However, the results of these 40 files, the lowest severity violation was important, and the highest was blocker. Fig. 3.1 depicts the most common categories that each violation fell under. The categories are Best Practices, Code Style, Design, Documentation, Error Prone, Performance, Multithreading, and Security (Table 1.1, Table 2.1). Code Style was the most common category at 58.6 % of total violations. Within the Code Style category, "Local Variable Could Be Final" and "Method Argument Could Be Final" were the most prominent violations at 127 and 83 instances respectively. The rest of the categories ranked from highest to lowest were Documentation at 18.6 %, Best Practices at 8.7 %, Design at 8.5 %, Performance at 2.8 %, Multithreading at 1.8 %, and Error Prone at 1.0 %.

Overall, ChatGPT's refactoring of code with quality and performance in mind, it is noteworthy that for the PMD categories, Performance did not rank high even though ChatGPT was supposed to refactor the code for improved performance. For example, the most common performance violation was "Avoid Instantiating Objects In Loops," which was mentioned a total of 10 times.

3.1.2. Complexity

After ChatGPT refactored the files for quality and complexity

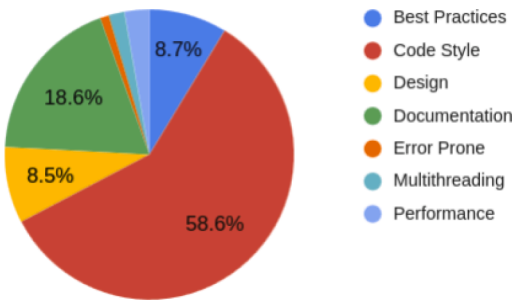


Fig. 3.1. PMD Violations for Performance.

Table 1.1
List of categorized PMD violations for performance. The number in parenthesis is the number of occurrences of that violation.

Performance	
Best Practices	-AvoidReassigningParameters (5)-LooseCoupling (31)-ReplaceVectorWithList (2)-SystemPrintIn (1)-UnusedLocalVariable (1)-UseVarargs (4)
Code Style	-AtLeastOneConstructor (1)-CommentDefaultAccessModifier (3)-ConfusingTernary (3)-FormalParameterNamingConventions (2)-LocalVariableCouldBeFinal (1 27)-LocalVariableNamingConventions (3)-LongVariable (1)-MethodArgumentCouldBeFinal (83)-OnlyOneReturn (33)-ShortClassName (3)-ShortVariable (34)-UseUnderscoreInNumericLiterals (2)-UseDiamondOperator (1)
Design	-CognitiveComplexity (1)-CyclomaticComplexity (1)-DataClass (1)-UseUtilityClass (40)
Documentation	-CommentRequired (94)
Error Prone	-AvoidLiteralsInIfCondition (4)-CompareObjectsWithEquals (1)
Multithreading	-UseConcurrentHashMap (9)
Performance	-AvoidinstantiatingObjectsInLoops (10)-UseIndexOfChar (2)-UselessStringValueOf (2)

Table 2.1
List of categorized PMD violations by severity for performance. The number in parenthesis is the number of occurrences of that violation.

Performance	
Blocker	-CyclomaticComplexity (1)-FormalParameterNamingConventions (2)-LocalVariableNamingConventions (3)
Critical	-AvoidReassigningParameters (3)
Urgent	-AtLeastOneConstructor (1)-AvoidLiteralsInIfCondition (4)-AvoidReassigningParameters (2)-AvoidinstantiatingObjectsInLoops (10)-CognitiveComplexity (1)-CommentDefaultAccessModifier (3)-CommentRequired (94)-CompareObjectsWithEquals (1)-ConfusingTernary (3)-DataClass (1)-LocalVariableCouldBeFinal (1 27)-LongVariable (1)-LooseCoupling (31)-MethodArgumentCouldBeFinal (83)-OnlyOneReturn (33)-ReplaceVectorWithList (2)-ShortClassName (1)-ShortVariable (34)-SystemPrintIn (1)-UnusedLocalVariable (1)-UseConcurrentHashMap (9)-UseDiamondOperator (1)-UseIndexOfChar (2)-UseUnderscoreInNumericLiterals (2)-UseUtilityClass (40)-UselessStringValueOf (2)
Important	-ShortClassName (2)-UseVarargs (4)

improvement, there was little to no change between the refactored and the original code. The minor changes we noticed were adjustments such as changing an ArrayList declaration to a List declaration and changing an Object type to List < Object >. Changing an ArrayList to a List does help with complexity since going through a List is faster. However, the other change has no effect on complexity.

Once the refactoring was finished, we identified a total of 508 violations which consisted of 5 blocker violations, 4 critical violations, 10 important violations, and 489 urgent violations. However, the results of these files, the lowest severity violation was important, and the highest was blocker. Fig. 3.2, Table 1.2, and Table 2.2 depict the most common categories that each violation fell under. Code Style was the most common category at 58.9 % of total violations. Within the Code Style category, "Local Variable Could Be Final" and "Method Argument Could Be Final" were the most prominent violations at 129 and 81 instances respectively. The rest of the categories ranked from highest to lowest were Documentation at 18.9 %, Design at 8.7 %, Documentation at 8.1 %, Performance at 2.8 %, Multithreading at 1.8 %, and Error Prone at 1.0 %.

Overall, ChatGPT's refactoring of code with quality and complexity in mind was also successful; however, compared to the other quality attributes, there was no significant difference to be made.

3.1.3. Coupling

When asking ChatGPT to refactor the 40 files to improve quality and coupling there were very few unique changes made to the code segments, especially when compared to the changes ChatGPT would make for the other quality attributes. In terms of coupling-specific refactoring changes, the only consistent change ChatGPT would make was removing unnecessary statements. Removing unnecessary statements not only helps to improve code readability but also helps to reduce the amount of coupling between different parts of the code. Some of the other modifications that coupling only shared with one or two other attributes are changing enhanced for loops to regular for loops and splitting calculations across multiple variables rather than doing it all in one.

When evaluating the 40 code segments refactored for coupling with PMD, it was revealed that there were a total of 534 violations and 15 compiler errors. Breaking these violations down from most severe to least there were 15 blocker errors, 5 critical, 502 urgent errors, and 12 important. Fig. 3.3, Table 1.3, and Table 2.3 show the violations broken down by categories. The most prevalent category was Code Style with 61 % of the violations followed by Documentation at 17.6 %, Best Practices and Design tied at 8.2 %, then Performance at 2.2 %, Multithreading at 1.7 %, and Error Prone at 0.9 %. The two most common violations throughout all 40 files were "Local Variable Could Be Final" with 131 appearances and "Method Argument Could Be Final" with 88 appearances, and both violations fell under the Code Style category.

When comparing these results against the other quality attributes it can be gathered that coupling files had the highest amount of compiler errors and the third highest amount of total violations.

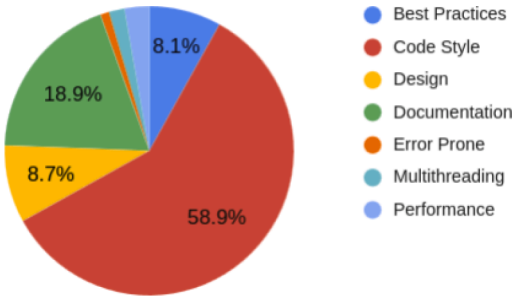


Fig. 3.2. PMD Violations for Complexity.

Table 1.2
List of categorized PMD violations for complexity. The number in parenthesis is the number of occurrences of that violation.

Complexity	
Best Practices	-AvoidReassigningParameters (3)-LooseCoupling (30)-ReplaceVectorWithList (2)-SystemPrintIn (1)-UnusedLocalVariable (1)-UseVarargs (4)
Code Style	-AtLeastOneConstructor (1)-CommentDefaultAccessModifier (1)-ConfusingTernary (4)-FormalParameterNamingConventions (2)-LocalVariableCouldBeFinal (1 2 9)-LocalVariableNamingConventions (3)-LongVariable (1)-MethodArgumentCouldBeFinal (81)-OneDeclarationPerLine (1)-OnlyOneReturn (33)-ShortClassName (6)-ShortVariable (33)-UseUnderscoreInNumericLiterals (2)-UnnecessarySemicolon (1)-UseDiamondOperator (1)
Design	-CognitiveComplexity (2)-CyclomaticComplexity (1)-DataClass (1)-UseUtilityClass (40)
Documentation	-CommentRequired (95)-CommentSize (1)
Error Prone	-AvoidLiteralsInIfCondition (4)-CompareObjectsWithEquals (1)
Multithreading	-UseConcurrentHashMap (9)
Performance	-AvoidinstantiatingObjectsInLoops (10)-UseIndexOfChar (2)-UselessStringValueOf (2)

Table 2.2
List of categorized PMD violations by severity for complexity. The number in parenthesis is the number of occurrences of that violation.

Complexity	
Blocker	-FormalParameterNamingConventions (2)-LocalVariableNamingConventions (3)
Critical	-AvoidReassigningParameters (3)-SystemPrintIn (1)
Urgent	-AtLeastOneConstructor (1)-AvoidLiteralsInIfCondition (4)-AvoidinstantiatingObjectsInLoops (10)-CognitiveComplexity (2)-CommentDefaultAccessModifier (1)-CommentRequired (95)-CommentSize (1)-CompareObjectsWithEquals (1)-ConfusingTernary (4)-CyclomaticComplexity (1)-DataClass (1)-LocalVariableCouldBeFinal (1 2 9)-LongVariable (1)-LooseCoupling (30)-MethodArgumentCouldBeFinal (81)-OnlyOneReturn (33)-ReplaceVectorWithList (2)-ShortClassName (1)-ShortVariable (33)-UnnecessarySemicolon (1)-UnusedLocalVariable (1)-UseConcurrentHashMap (9)-UseDiamondOperator (1)-UseIndexOfChar (2)-UseUnderscoreInNumericLiterals (2)-UseUtilityClass (40)-UselessStringValueOf (2)
Important	-OneDeclarationPerLine (1)-ShortClassName (5)-UseVarargs (4)

3.1.4. Cohesion

Asking ChatGPT to refactor the 40 files to improve quality and cohesion revealed only one cohesion-specific modification: adding in a ternary operator. Adding in a ternary operator makes it so conditional statements can be written in a more readable and concise way, thereby increasing cohesion. Additionally, cohesion shared similar modifications with both design size and complexity. ChatGPT's refactoring for cohesion and design size both shared the trait of making variables and methods private and/or public. Refactoring to improve cohesion and complexity both involved the modification of replacing the equals() function with two equals signs (==).

After running the 40 code segments refactored to improve cohesion through PMD we found a total of 518 violations and 0 compiler errors. Breaking down the 518 violations by severity from highest to lowest there were 7 critical violations, 496 urgent, and 15 important violations. These violations fit into 7 major categories with 57.7 % of the violations falling under the Code Style category, 18.3 % under Documentation, 10.6 % under the Best Practices category, 8.3 % under Design, 2.3 % fell under Performance, 1.7 % under Multithreading, and 1 % under the Error Prone category as shown in Fig. 3.4. From the category with the most violations, Code Style, the two most common violations were "Local Variable Could Be Final" and "Method Argument Could Be Final" at 125 and 85 violations respectively (Table 1.4, Table 2.4).

Important factors to consider when comparing these PMD results against the other 7 quality attributes are that the cohesion files had the least amount of compiler errors and were middle-of-the-road in terms of total violations.

3.1.5. Design size

When requesting ChatGPT to refactor the code segments for quality and design size improvement, we observed only one unique change: utilizing the length() method on an array instead of using a separate

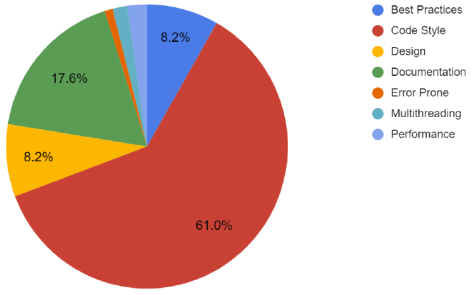


Fig. 3.3. PMD Violations for Coupling.

counter variable. This modification eliminated the need for an additional variable that would consume unnecessary memory, as the length () method achieved the same outcome. Furthermore, both design size and coupling shared the change of removing unused methods, while design size and cohesion shared the change of adding access modifiers to variables and methods.

After evaluating ChatGPT's 40 refactored code segments with the PMD tool, we identified a total of 535 violations, including 11 compiler errors. These violations varied in severity, ranging from important, urgent, critical, to blocker. Specifically, we discovered 504 urgent violations, 6 critical violations, 14 important violations, and 11 blocker violations (Table 1.5, Table 2.5). Fig. 3.5 illustrates that the most prevalent violation category, comprising 58.9 % of the total violations, was Code Style. Within the code style category, two of the most common violations were 127 instances of "Local Variable Could Be Final" and 89 instances of "Method Argument Could Be Final." The remaining PMD violation categories, ranked from greatest to least, were Documentation

Table 1.3
List of categorized PMD violations for coupling. The number in parenthesis is the number of occurrences of that violation.

Coupling	
Best Practices	-AvoidReassigningParameters (4)-LooseCoupling (31)-ReplaceVectorWithList (2)-SystemPrintIn (1)-UnusedLocalVariable (1)-UseVarargs (5)
Code Style	-AtLeastOneConstructor (1)-ClassNamingConventions (5)-CommentDefaultAccessModifier (2)-ConfusingTernary (4)-FormalParameterNamingConventions (3)-LocalVariableCouldBeFinal (1 3 1)-LocalVariableNamingConventions (5)-LongVariable (1)-MethodArgumentCouldBeFinal (88)-MethodNamingConventions (2)-OnlyOneReturn (32)-ShortClassName (5)-ShortVariable (42)-UnnecessaryImport (2)-UseUnderscoreInNumericLiterals (2)-UseDiamondOperator (1)
Design	-CognitiveComplexity (1)-CyclomaticComplexity (1)-ImmutableField (2)-UseUtilityClass (40)
Documentation	-CommentRequired (94)
Error Prone	-AvoidLiteralsInIfCondition (4)-CompareObjectsWithEquals (1)
Multithreading	-UseConcurrentHashMap (9)
Performance	-AvoidinstantiatingObjectsInLoops (8)-UseIndexOfChar (2)-UselessStringValueOf (2)

Table 2.3
List of categorized PMD violations by severity for coupling. The number in parenthesis is the number of occurrences of that violation.

Coupling	
Blocker	-ClassNamingConventions (5)-FormalParameterNamingConventions (3)-LocalVariableNamingConventions (5)-MethodNamingConventions (2)
Critical	-AvoidReassigningParameters (4)-SystemPrintIn (1)
Urgent	-AtLeastOneConstructor (1)-AvoidLiteralsInIfCondition (4)-AvoidinstantiatingObjectsInLoops (8)-CognitiveComplexity (1)-CommentDefaultAccessModifier (2)-CommentRequired (94)-CompareObjectsWithEquals (1)-ConfusingTernary (4)-CyclomaticComplexity (1)-ImmutableField (2)-LocalVariableCouldBeFinal (1 3 1)-LongVariable (1)-LooseCoupling (31)-MethodArgumentCouldBeFinal (88)-OnlyOneReturn (32)-ReplaceVectorWithList (2)-ShortVariable (42)-UnusedLocalVariable (1)-UseConcurrentHashMap (9)-UseDiamondOperator (1)-UseIndexOfChar (2)-UseUnderscoreInNumericLiterals (2)-UseUtilityClass (40)-UselessStringValueOf (2)
Important	-ShortClassName (5)-UnnecessaryImport (2)-UseVarargs (5)

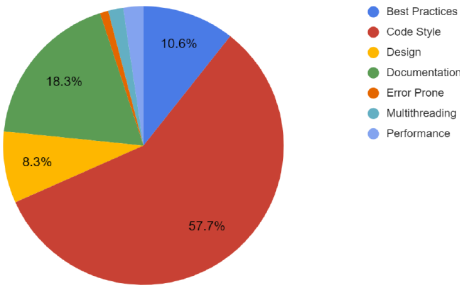


Fig. 3.4. PMD Violations for Cohesion.

Table 1.4
List of categorized PMD violations for cohesion. The number in parenthesis is the number of occurrences of that violation.

Cohesion	
Best Practices	-AvoidReassigningParameters (6)-LooseCoupling (42)-ReplaceVectorWithList (2)-SystemPrintIn (1)-UseVarargs (4)
Code Style	-AtLeastOneConstructor (1)-CommentDefaultAccessModifier (1)-ConfusingTernary (4)-LocalVariableCouldBeFinal (1 2 5)-LongVariable (1)-MethodArgumentCouldBeFinal (85)-NoPackage (1)-OneDeclarationPerLine (1)-OnlyOneReturn (32)-ShortClassName (6)-ShortVariable (36)-UnnecessaryImport (4)-UseUnderscoreInNumericLiterals (2)
Design	-CognitiveComplexity (1)-CyclomaticComplexity (1)-DataClass (1)-UseUtilityClass (40)
Documentation	-CommentRequired (95)
Error Prone	-AvoidLiteralsInIfCondition (4)-CompareObjectsWithEquals (1)
Multithreading	-UseConcurrentHashMap (9)
Performance	-AvoidinstantiatingObjectsInLoops (8)-UseIndexOfChar (2)-UselessStringValueOf (2)

with 17.8 %, Best Practices with 10.5 %, Design with 8.0 %, Performance with 2.2 %, Multithreading with 1.7 %, and Error-Prone with 0.9 %.

When compared to the PMD violation results for the other quality attributes, it is noteworthy that design size exhibited the second-highest number of violations and the third-highest number of compiler errors.

3.1.6. Readability

When prompting ChatGPT to refactor the code segment for improved quality and readability, we discovered a single change that stood out. This change involved importing the collections class to simplify the code segments by allowing repeated calls to Collections.max(). By importing the collections class, the code became more concise and easier to understand. Additionally, readability was one of three quality attributes that added comments to the code segments.

After evaluating ChatGPT's 40 refactored code segments using the PMD tool, we identified a total of 552 violations, including 12 compiler errors. These violations encompassed various levels of severity, ranging from important, urgent, critical, to blocker. Specifically, there were 522 urgent violations, 5 critical violations, 13 important violations, and 12 blocker violations (Table 1.6, Table 2.6). Fig. 3.6 illustrates that the most prevalent violation category, accounting for 55.1 % of the total

Table 2.4
List of categorized PMD violations by severity for cohesion. The number in parenthesis is the number of occurrences of that violation.

Cohesion	
Blocker	
Critical	-AvoidReassigningParameters (6)-SystemPrintIn (1)
Urgent	-AtLeastOneConstructor (1)-AvoidLiteralsInIfCondition (4)-AvoidinstantiatingObjectsInLoops (8)-CognitiveComplexity (1)-CommentDefaultAccessModifier (1)-CommentRequired (95)-CompareObjectsWithEquals (1)-ConfusingTernary (4)-CyclomaticComplexity (1)-DataClass (1)-LocalVariableCouldBeFinal (1 2 5)-LongVariable (1)-LooseCoupling (42)-MethodArgumentCouldBeFinal (85)-NoPackage (1)-OnlyOneReturn (32)-ReplaceVectorWithList (2)-ShortVariable (36)-UseConcurrentHashMap (9)-UseIndexOfChar (2)-UseUnderscoreInNumericLiterals (2)-UseUtilityClass (40)-UselessStringValueOf (2)
Important	-OneDeclarationPerLine (1)-ShortClassName (6)-UnnecessaryImport (4)-UseVarargs (4)

Table 1.5
List of categorized PMD violations for design size. The number in parenthesis is the number of occurrences of that violation.

Design Size	
Best Practices	-AvoidReassigningParameters (5)-LooseCoupling (44)-ReplaceVectorWithList (2)-SystemPrintIn (1)-UseVarargs (4)
Code Style	-AtLeastOneConstructor (1)-ClassNamingConventions (6)-CommentDefaultAccessModifier (3)-ConfusingTernary (3)-FormalParameterNamingConventions (2)-LocalVariableCouldBeFinal (1 2 7)-LocalVariableNamingConventions (3)-LongVariable (3)-MethodArgumentCouldBeFinal (89)-OneDeclarationPerLine (1)-OnlyOneReturn (32)-ShortClassName (6)-ShortVariable (34)-UnnecessaryImport (3)-UseUnderscoreInNumericLiterals (2)
Design	-CognitiveComplexity (1)-CyclomaticComplexity (1)-DataClass (1)-UseUtilityClass (40)
Documentation	-CommentRequired (95)
Error Prone	-AvoidLiteralsInIfCondition (4)-CompareObjectsWithEquals (1)
Multithreading	-UseConcurrentHashMap (9)
Performance	-AvoidinstantiatingObjectsInLoops (8)-UseIndexOfChar (2)-UselessStringValueOf (2)

violations, was Code Style. Within the Code Style category, there were notable occurrences of violations such as 124 instances of "Local Variable Could Be Final" and 34 instances of "Short Variable." The

Table 2.5
List of categorized PMD violations by severity for design size. The number in parenthesis is the number of occurrences of that violation.

Design Size	
Blocker	-ClassNamingConventions (6)-FormalParameterNamingConventions (2)-LocalVariableNamingConventions (3)
Critical	-AvoidReassigningParameters (5)-SystemPrintIn (1)
Urgent	-AtLeastOneConstructor (1)-AvoidLiteralsInIfCondition (4)-AvoidinstantiatingObjectsInLoops (8)-CognitiveComplexity (1)-CommentDefaultAccessModifier (3)-CommentRequired (95)-CompareObjectsWithEquals (1)-ConfusingTernary (3)-CyclomaticComplexity (1)-DataClass (1)-LocalVariableCouldBeFinal (1 2 7)-LongVariable (3)-LooseCoupling (44)-MethodArgumentCouldBeFinal (89)-OnlyOneReturn (32)-ReplaceVectorWithList (2)-ShortVariable (34)-UseConcurrentHashMap (9)-UseIndexOfChar (2)-UseUnderscoreInNumericLiterals (2)-UseUtilityClass (40)-UselessStringValueOf (2)
Important	-OneDeclarationPerLine (1)-ShortClassName (6)-UnnecessaryImport (3)-UseVarargs (4)

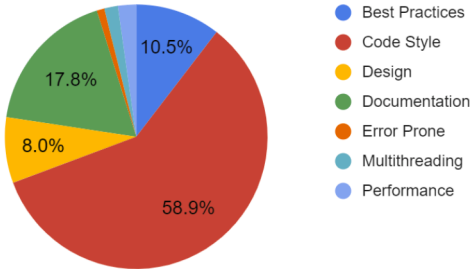


Fig. 3.5. PMD Violations for Design Size.

remaining PMD violation categories, ranked from highest to lowest, were Documentation with 17.2 %, Best Practices with 14.5 %, Design with 8.0 %, Performance with 2.7 %, Multithreading with 1.6 %, and Error-Prone with 0.9 %.

Compared to our PMD violation results for the other quality attributes, readability had the greatest number of violations and the second-greatest number of compiler errors. Also, readability had the lowest percentage of Code Style violations.

3.1.7. Reusability

After cueing ChatGPT to refactor the code segments to specifically improve quality and reusability, we recognized four changes distinctly for the reusability quality attribute. These changes consisted of the addition of helper functions to prevent the code from having a singular crowded method, as well as a change to make variables final. There were also changes focused on the loops, specifically a change from a for loop to an enhanced for loop which makes the code more readable and reduces the chance of bugs, as well as a change in the loop conditions which made the code more concise. Reusability was also one of two quality attributes that moved variables inside the for loop and changed the index of loops.

Table 1.6
List of categorized PMD violations for readability. The number in parenthesis is the number of occurrences of that violation.

Readability	
Best Practices	-AvoidReassigningParameters (4)-LooseCoupling (69)-ReplaceVectorWithList (2)-SystemPrintIn (1)-UseVarargs (4)
Code Style	-AtLeastOneConstructor (1)-ClassNamingConventions (4)-CommentDefaultAccessModifier (2)-ConfusingTernary (4)-FormalParameterNamingConventions (2)-LocalVariableCouldBeFinal (1 2 4)-LocalVariableNamingConventions (3)-LongVariable (1)-MethodArgumentCouldBeFinal (82)-MethodNamingConventions (3)-OnlyOneReturn (33)-ShortClassName (6)-ShortVariable (34)-UnnecessaryImport (3)-UseUnderscoreInNumericLiterals (2)
Design	-CognitiveComplexity (1)-CyclomaticComplexity (1)-DataClass (1)-MutableStaticState (1)-UseUtilityClass (40)
Documentation	-CommentRequired (95)
Error Prone	-AvoidLiteralsInIfCondition (4)-CompareObjectsWithEquals (1)
Multithreading	-UseConcurrentHashMap (9)
Performance	-AvoidinstantiatingObjectsInLoops (11)-UseIndexOfChar (2)-UselessStringValueOf (2)

Table 2.6
List of categorized PMD violations by severity for readability. The number in parenthesis is the number of occurrences of that violation.

Readability	
Blocker	-ClassNamingConventions (4)-FormalParameterNamingConventions (2)-LocalVariableNamingConventions (3)-MethodNamingConventions (3)
Critical	-AvoidReassigningParameters (4)-SystemPrintIn (1)
Urgent	-AtLeastOneConstructor (1)-AvoidLiteralsInIfCondition (4)-AvoidinstantiatingObjectsInLoops (11)-CognitiveComplexity (1)-CommentDefaultAccessModifier (2)-CommentRequired (95)-CompareObjectsWithEquals (1)-ConfusingTernary (4)-CyclomaticComplexity (1)-DataClass (1)-LocalVariableCouldBeFinal (1 2 4)-LongVariable (1)-LooseCoupling (69)-MethodArgumentCouldBeFinal (82)-MutableStaticState (1)-OnlyOneReturn (33)-ReplaceVectorWithList (2)-ShortVariable (34)-UseConcurrentHashMap (9)-UseIndexOfChar (2)-UseUnderscoreInNumericLiterals (2)-UseUtilityClass (40)-UselessStringValueOf (2)
Important	-ShortClassName (6)-UnnecessaryImport (3)-UseVarargs (4)

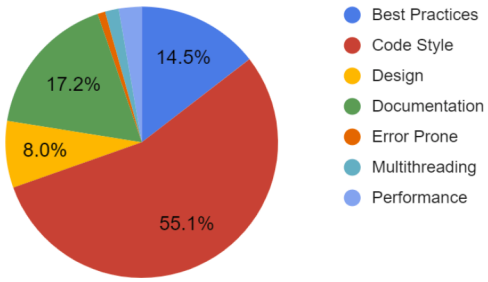


Fig. 3.6. PMD Violations for Readability.

After evaluating ChatGPT's 40 refactored code segments using the PMD tool, we recorded 500 total violations, ranging from important, urgent, critical, to blocker in terms of increasing severity. There were 17 important violations, 470 urgent violations, 5 critical violations, and 8 blocker violations (Table 1.7, Table 2.7). Fig. 3.7 illustrates that Code Style was the most prevalent violation category, as it accounted for 64.6 % of the violations. Within the Code Style category, there were 138 instances of "Local Variable Could Be Final" violations and 84 instances of "Method Argument Could Be Final" violations. The remaining PMD violation categories, in descending order, were Documentation with 17.4 %, Design with 8.4 %, Best Practices with 4.4 %, Performance with 2.4 %, Multithreading with 1.8 %, and Error Prone with 1.0 %.

In comparison to the PMD violation results for the remaining quality attributes, reusability had the least amount of violations and had the greatest percentage of violations for Code Style.

3.1.8. Understandability

When prompting ChatGPT to refactor the code to specifically improve quality and understandability, we identified one specific change that was unique to the refactored code with respect to

Table 1.7
List of categorized PMD violations for reusability. The number in parenthesis is the number of occurrences of that violation.

Reusability	
Best Practices	-AvoidReassigningParameters (4)-ForLoopCanBeForeach (2)-LooseCoupling (8)-ReplaceVectorWithList (2)-SystemPrintIn (1)-UseVarargs (5)
Code Style	-AtLeastOneConstructor (1)-ClassNamingConventions (2)-CommentDefaultAccessModifier (2)-ConfusingTernary (4)-ControlStatementBraces (2)-FormalParameterNamingConventions (2)-LocalVariableCouldBeFinal (1 3 9)-LocalVariableNamingConventions (3)-LongVariable (2)-MethodArgumentCouldBeFinal (84)-MethodNamingConventions (1)-OnlyOneReturn (32)-ShortClassName (5)-ShortVariable (35)-UnnecessaryImport (6)-UseUnderscoreInNumericLiterals (2)-UnnecessarySemicolon (1)
Design	-CognitiveComplexity (1)-CyclomaticComplexity (1)-UseUtilityClass (40)
Documentation	-CommentRequired (87)
Error Prone	-AvoidLiteralsInIfCondition (4)-CompareObjectsWithEquals (1)
Multithreading	-UseConcurrentHashMap (9)
Performance	-AvoidinstantiatingObjectsInLoops (10)-UseIndexOfChar (2)

Table 2.7
List of categorized PMD violations by severity for reusability. The number in parenthesis is the number of occurrences of that violation.

Reusability	
Blocker	-ClassNamingConventions (2)-FormalParameterNamingConventions (2)-LocalVariableNamingConventions (3)-MethodNamingConventions (1)
Critical	-AvoidReassigningParameters (4)-SystemPrintIn (1)
Urgent	-AtLeastOneConstructor (1)-AvoidLiteralsInIfCondition (4)-AvoidinstantiatingObjectsInLoops (10)-CognitiveComplexity (1)-CommentDefaultAccessModifier (2)-CommentRequired (87)-CompareObjectsWithEquals (1)-ConfusingTernary (4)-ControlStatementBraces (2)-CyclomaticComplexity (1)-ForLoopCanBeForeach (2)-LocalVariableCouldBeFinal (138)-LongVariable (2)-LooseCoupling (8)-MethodArgumentCouldBeFinal (84)-OnlyOneReturn (32)-ReplaceVectorWithList (2)-ShortVariable (35)-UnnecessarySemicolon (1)-UseConcurrentHashMap (9)-UseIndexOfChar (2)-UseUnderscoreInNumericLiterals (2)-UseUtilityClass (40)
Important	-ShortClassName (6)-UnnecessaryImport (6)-UseVarargs (5)

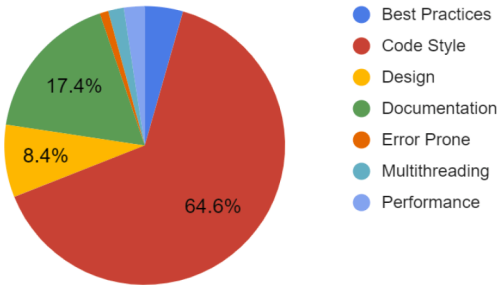


Fig. 3.7. PMD Violations for Reusability.

understandability. This change aligned variable declarations and assignments which increased consistency. Additionally, understandability was one of three quality attributes that consistently added comments and split calculations into multiple variables, rather than doing the entire calculation in one.

After evaluating ChatGPT's 40 refactored code segments using the PMD tool, we identified a total of 532 violations, ranging from important, urgent, critical, to blocker in terms of increasing severity. There were 10 urgent violations, 516 important violations, 5 critical violations, and 1 blocker violation (Table 1.8, Table 2.8). Fig. 3.8 shows that the most common violation category was Code Style, which accounted for 57.1 % of violations. Within the Code Style category, there were 136 instances of "Local Variable Could Be Final" violations and 88 instances of "Method Argument Could Be Final" violations. The remaining PMD violation categories, ranked from highest to lowest, were Documentation with 17.9 %, Best Practices with 11.5 %, Design with 8.3 %, Performance with 2.6 %, Multithreading with 1.7 %, and Error Prone with 0.9 %.

In relation to the other attributes, understandability had the third-

Table 1.8
List of categorized PMD violations for understandability. The number in parenthesis is the number of occurrences of that violation.

Understandability	
Best Practices	-AvoidReassigningParameters (4)-LooseCoupling (50)-ReplaceVectorWithList (2)-SystemPrintIn (1)-UseVarargs (4)
Code Style	-AtLeastOneConstructor (1)-ClassNamingConventions (1)-CommentDefaultAccessModifier (1)-ConfusingTernary (3)-LocalVariableCouldBeFinal (136)-LongVariable (2)-MethodArgumentCouldBeFinal (88)-OnlyOneReturn (32)-ShortClassName (6)-ShortVariable (32)-UseUnderscoreInNumericLiterals (2)
Design	-CognitiveComplexity (2)-CyclomaticComplexity (1)-DataClass (1)-UseUtilityClass (40)
Documentation	-CommentRequired (95)
Error Prone	-AvoidLiteralsInIfCondition (4)-CompareObjectsWithEquals (1)
Multithreading	-UseConcurrentHashMap (9)
Performance	-AvoidinstantiatingObjectsInLoops (10)-UseIndexOfChar (2)-UselessStringValueOf (2)

Table 2.8
List of categorized PMD violations by severity for understandability. The number in parenthesis is the number of occurrences of that violation.

Understandability	
Blocker	-ClassNamingConventions (1)
Critical	-AvoidReassigningParameters (4)-SystemPrintIn (1)
Urgent	-AtLeastOneConstructor (1)-AvoidLiteralsInIfCondition (4)-AvoidinstantiatingObjectsInLoops (10)-CognitiveComplexity (2)-CommentDefaultAccessModifier (1)-CommentRequired (95)-CompareObjectsWithEquals (1)-ConfusingTernary (3)-CyclomaticComplexity (1)-DataClass (1)-LocalVariableCouldBeFinal (136)-LongVariable (2)-LooseCoupling (50)-MethodArgumentCouldBeFinal (88)-OnlyOneReturn (32)-ReplaceVectorWithList (2)-ShortVariable (32)-UseConcurrentHashMap (9)-UseIndexOfChar (2)-UseUnderscoreInNumericLiterals (2)-UseUtilityClass (40)-UselessStringValueOf (2)
Important	-ShortClassName (6)-UseVarargs (4)

greatest number of violations and second-lowest number of errors. The PMD results for understandability were overall average.

3.1.9. *Venn Diagram comparison*

As shown in Fig. 3.9, there were a total of 5 different violations, all of which were classified as blockers. The violation pertaining to "Cyclomatic Complexity" was exclusive to the performance attribute. The coupling, readability, and reusability attributes shared the violations of

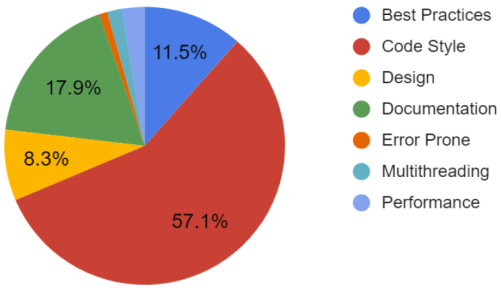


Fig. 3.8. PMD Violations for Understandability.

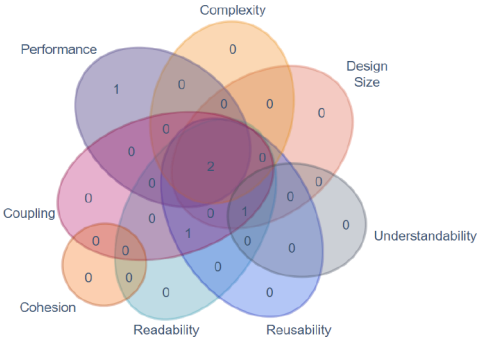


Fig. 3.9. Venn Diagram illustrating the overlapping occurrences of violations within the Blocker category across the 8 quality attributes.

"Method Naming Conventions." Additionally, the coupling, design understandability, readability, reusability, and size attributes all had violations of "Class Naming Conventions." Lastly, the violations of "Formal Parameter Naming Conventions" and "Local Variable Naming Conventions" were common among the performance, complexity, coupling, design size, readability, and reusability attributes.

Fig. 3.10 depicts how out of the total violations, only 2 were classified as critical violations. All the quality attributes, with the exception of performance, exhibited the violation known as "System Print ln." Additionally, all 8 quality attributes were found to have a violation related to "Avoid Reassigning Parameters."

Fig. 3.11 portrays a total of 34 violations categorized as urgent. Among the 8 quality attributes, 6 had urgent violations that were specific to them. The performance attribute stood out with violations exclusively related to "Avoid Reassigning Parameters" and "System Print ln." The complexity attribute was unique in having the "Comment Size" violation, while the coupling attribute had the exclusive violation of "Immutable Field." The cohesion attribute was found to have the exclusive violation "No Package," and the readability attribute had only the violation "Mutable Static State." The reusability attribute, on the other hand, had exclusive violations for "For Loop Can Be For Each" and

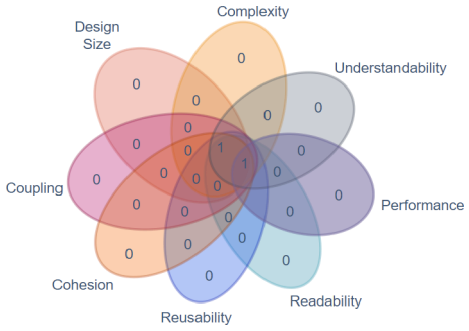


Fig. 3.10. Venn Diagram illustrating the overlapping occurrences of violations within the Critical category across the 8 quality attributes.

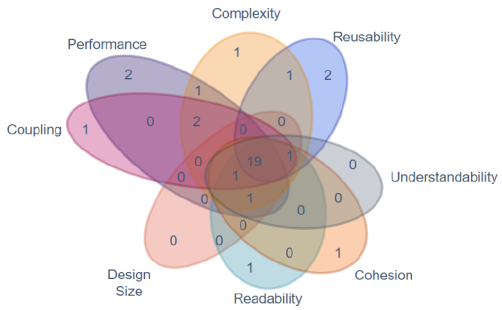


Fig. 3.11. Venn Diagram illustrating the overlapping occurrences of violations within the Urgent category across the 8 quality attributes.

"Control Statement Braces." The design size and understandability attributes did not exhibit any distinct critical violations compared to the others.

The performance and complexity attributes shared the violation of "Short Class Name." The complexity and reusability attributes had the violation "Unnecessary Semicolon" in common. Additionally, the performance, complexity, and coupling attributes shared the violations "Unused Local Variable" and "Use Diamond Operator." Furthermore, the performance, design size, cohesion, complexity, readability, and understandability attributes were all affected by the "Data Class" violation.

With the exception of reusability, all quality attributes shared the violation "Useless String Of." Similarly, all quality attributes, excluding performance, shared the "Cyclomatic Complexity" violation. Interestingly, all 8 quality attributes had a total of 19 violations in common, including "Loose Coupling," "Replace Vector With A List," "At Least One Constructor," "Comment Default Access Modifier," "Confusing Ternary," "Local Variable Could Be Final," "Long Variable," "Method Argument Could Be Final," "Only One Return," "Short Variable," "Use Underscore In Numeric Literals," "Cognitive Complexity," "Use Utility Class," "Comment Required," "Avoid Literals In If Condition," "Compare Objects With Equals," "Use Concurrent Hash Map," "Avoid Instantiating Objects In Loops," and "Use Index Of Char.".

Fig. 3.12 illustrates how there were a total of 4 important violations. The complexity, cohesion, and design size attributes had the "One Declaration Per Line" violation in common. In addition, the coupling, cohesion, design size, readability, and reusability attributes were all found to have the "Unnecessary Import" violation. Lastly, all 8 quality attributes shared the violations of "Short Class Name" and "Use Varargs.".

3.1.10. PMD violation table

3.1.10.1. Refactoring examples for each quality attribute. Performance:

After requesting ChatGPT to enhance both code quality and performance in this code segment, LIS.java, several noteworthy changes were

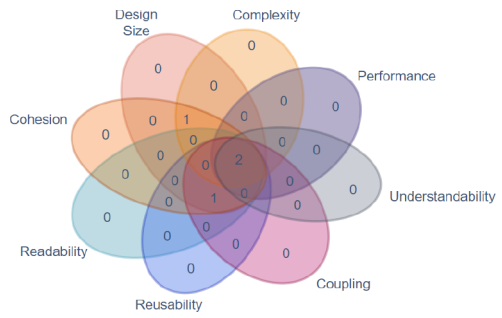


Fig. 3.12. Venn Diagram illustrating the overlapping occurrences of violations within the Important category across the 8 quality attributes.

```java
package correct_java_programs;
import java.util.*;
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

/**
 *
 * @author derricklin
 */
public class LIS {
    public static int lis(int[] arr) {
        Map<Integer,Integer> ends = new HashMap<Integer, Integer>(100);
        int longest = 0;

        int i = 0;
        for (int val : arr) {

            ArrayList<Integer> prefix_lengths = new ArrayList<Integer>(100);
            for (int j=1; j < longest+1; j++) {
                if (arr[ends.get(j)] < val) {
                    prefix_lengths.add(j);
                }
            }

            int length = !prefix_lengths.isEmpty() ? Collections.max(prefix_lengths) : 0;

            if (length == longest || val < arr[ends.get(length+1)]) {
                ends.put(length+1, i);
                longest = Math.max(longest,length + 1);
            }

            i++;
        }
        return longest;
    }
}
```

Fig. 3.13. Original code for LIS.java.

With no explanation refactor the Java code to improve its quality and performance:

```java
package correct_java_programs;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.Collections;

public class LIS {
    public static int lis(int[] arr) {
        Map<Integer, Integer> ends = new HashMap<>();
        int longest = 0;

        for (int i = 0; i < arr.length; i++) {
            List<Integer> prefixLengths = new ArrayList<>();

            for (int j = 1; j <= longest; j++) {
                if (arr[ends.get(j)] < arr[i]) {
                    prefixLengths.add(j);
                }
            }

            int length = !prefixLengths.isEmpty() ? Collections.max(prefixLengths) : 0;

            if (length == longest || arr[i] < arr[ends.get(length + 1)]) {
                ends.put(length + 1, i);
                longest = Math.max(longest, length + 1);
            }
        }

        return longest;
    }
}
```

Fig. 3.14. LIS refactored for performance.

implemented. Fig. 3.13 shows the original version of LIS.java and Fig. 3.14 displays the refactored version of LIS.java to improve performance. One particular improvement between the original and refactored code for LIS.java that stands out is how ChatGPT converted the for-each loop into a traditional for loop. This modification effectively eliminated the need to track the index of "arr" outside the loop header, resulting in a more efficient process. Moreover, in contrast to the original code where a HashMap "ends" was created with a predetermined size of 100, ChatGPT decided not to set a size limit, allowing for a more dynamic approach. Because of this, within the converted for loop, ChatGPT employed the condition "i < arr.length" to iterate through all elements in the HashMap effectively. Additionally, in the second loop, instead of the original condition "j < longest + 1," ChatGPT opted for "j <= longest." This modification enhances the iteration process by making it more comprehensive and precise.

Complexity:

When we asked ChatGPT to refactor the LIS.java code segment with a focus on improving quality and complexity, it surprisingly did not introduce any unique changes specific to this attribute, unlike some of the other attributes. The refactored code of LIS.java for complexity is indicated in Fig. 3.15. Although the refactored code did alter the second loop condition to "j <= longest," similar to what was done for performance, it still retained the for-each loop. However, the primary changes made in this code segment involved adhering to Java conventions for renaming variables, importing specific classes instead of using java.util,

With no explanation refactor the Java code to improve its quality and complexity:

```java
package correct_java_programs;

import java.util.ArrayList;
import java.util.Collections;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

public class LIS {
    public static int lis(int[] arr) {
        Map<Integer, Integer> ends = new HashMap<>(100);
        int longest = 0;
        int i = 0;

        for (int val : arr) {
            List<Integer> prefixLengths = new ArrayList<>(100);
            for (int j = 1; j <= longest; j++) {
                if (arr[ends.get(j)] < val) {
                    prefixLengths.add(j);
                }
            }

            int length = !prefixLengths.isEmpty() ? Collections.max(prefixLengths) : 0;

            if (length == longest || val < arr[ends.get(length + 1)]) {
                ends.put(length + 1, i);
                longest = Math.max(longest, length + 1);
            }

            i++;
        }

        return longest;
    }
}
```

Fig. 3.15. LIS refactored for complexity.

utilizing the diamond operator, and converting the data type of "prefixLength" from ArrayList to List. It is important to note that these changes were made across multiple attributes and not exclusively for this one.

Coupling:

When we prompted ChatGPT to refactor the LIS.java code segment with a focus on enhancing quality and coupling, the changes it made were remarkably similar to the ones made for complexity. Fig. 3.16 shows the code segment after being refactored by ChatGPT to improve coupling. Just like in the complexity improvements, the for loop condition was altered to "j <= longest," while the for-each loop structure remained unchanged. Other modifications included renaming variables, reformatting the code, utilizing the diamond operator, and switching the data type of "prefixLength" from ArrayList to List. However, these modifications were consistent across the attributes, with no unique alterations specifically for quality and coupling.

Cohesion:

Interestingly, the refactored LIS code aimed at improving quality and cohesion, illustrated in Fig. 3.17, displayed only minimal changes compared to the original code. The alterations consisted of renaming variables, reformatting the code, and adopting the diamond operator, which are not unique changes to the cohesion attribute.

Design Size:

When we asked ChatGPT to refactor the code to improve quality and design size, it led to several unique modifications that were not

With no explanation refactor the Java code to improve its quality and coupling:

```java
package correct_java_programs;

import java.util.*;

public class LIS {
    public static int lis(int[] arr) {
        Map<Integer, Integer> ends = new HashMap<>(100);
        int longest = 0;
        int i = 0;

        for (int val : arr) {
            List<Integer> prefixLengths = new ArrayList<>(100);

            for (int j = 1; j <= longest; j++) {
                if (arr[ends.get(j)] < val) {
                    prefixLengths.add(j);
                }
            }

            int length = !prefixLengths.isEmpty() ? Collections.max(prefixLengths) : 0;

            if (length == longest || val < arr[ends.get(length + 1)]) {
                ends.put(length + 1, i);
                longest = Math.max(longest, length + 1);
            }

            i++;
        }

        return longest;
    }
}
```

Fig. 3.16. LIS refactored for coupling.

With no explanation refactor the Java code to improve its quality and cohesion:

```java
package correct_java_programs;

import java.util.*;

public class LIS {
    public static int lis(int[] arr) {
        Map<Integer, Integer> ends = new HashMap<>(100);
        int longest = 0;

        int i = 0;
        for (int val : arr) {
            ArrayList<Integer> prefixLengths = new ArrayList<>(100);
            for (int j = 1; j < longest + 1; j++) {
                if (arr[ends.get(j)] < val) {
                    prefixLengths.add(j);
                }
            }

            int length = !prefixLengths.isEmpty() ? Collections.max(prefixLengths) : 0;

            if (length == longest || val < arr[ends.get(length + 1)]) {
                ends.put(length + 1, i);
                longest = Math.max(longest, length + 1);
            }

            i++;
        }
        return longest;
    }
}
```

Fig. 3.17. LIS refactored for cohesion.

With no explanation refactor the Java code to improve its quality and design size:

```java
package correct_java_programs;

import java.util.ArrayList;
import java.util.Collections;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

public class LIS {
    public static int lis(int[] arr) {
        Map<Integer, Integer> ends = new HashMap<>(100);
        int longest = 0;

        for (int i = 0; i < arr.length; i++) {
            List<Integer> prefixLengths = new ArrayList<>(100);
            for (int j = 1; j <= longest; j++) {
                if (arr[ends.get(j)] < arr[i]) {
                    prefixLengths.add(j);
                }
            }

            int length = prefixLengths.isEmpty() ? 0 : Collections.max(prefixLengths);

            if (length == longest || arr[i] < arr[ends.get(length + 1)]) {
                ends.put(length + 1, i);
                longest = Math.max(longest, length + 1);
            }
        }
        return longest;
    }
}
```

Fig. 3.18. LIS refactored for design size.

prompted by all other attributes. The code after ChatGPT refactored LIS.java to improve design size is displayed in Fig. 3.18. Similar to the performance improvements, the refactored code for design size transformed the for-each loop into a for loop. Additionally, in the original LIS code, the loop variables "i" and "j" were declared outside the loops and initialized separately. However, in the refactored code, these loop variables were declared within the loops' headers. This change appropriately scoped the loop variables, reducing their visibility and preventing potential naming conflicts. Furthermore, the original code used the conditional expression "!prefix_lengths.isEmpty()?Collections.max(prefix_lengths):0" to calculate the length, meanwhile, the refactored code modified this conditional expression to "prefixLengths.isEmpty()?0: Collections.max(pr-efixLengths)." Despite providing the same functionality, this change adheres to a more conventional conditional expression format.

Readability:

Almost identically to cohesion, there were minimal modifications between the original LIS.java code and the refactored LIS.java code shown in Fig. 3.19 to improve quality and readability. The changes included renaming, reformatting, using the diamond operator, and importing specific classes instead of java.util. Therefore, there were no distinctive changes implemented in relation to the readability attribute for the LIS.java code segment.

Reusability:

Upon requesting ChatGPT to refactor the LIS code with a focus on

With no explanation refactor the Java code to improve its quality and readability:

```java
package correct_java_programs;

import java.util.ArrayList;
import java.util.Collections;
import java.util.HashMap;
import java.util.Map;

public class LIS {
    public static int lis(int[] arr) {
        Map<Integer, Integer> ends = new HashMap<>(100);
        int longest = 0;

        int i = 0;
        for (int val : arr) {
            ArrayList<Integer> prefixLengths = new ArrayList<>(100);
            for (int j = 1; j < longest + 1; j++) {
                if (arr[ends.get(j)] < val) {
                    prefixLengths.add(j);
                }
            }

            int length = !prefixLengths.isEmpty() ? Collections.max(prefixLengths) : 0;

            if (length == longest || val < arr[ends.get(length + 1)]) {
                ends.put(length + 1, i);
                longest = Math.max(longest, length + 1);
            }

            i++;
        }
        return longest;
    }
}
```

Fig. 3.19. LIS refactored for readability.

With no explanation refactor the Java code to improve its quality and reusability:

```java
package correct_java_programs;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

public class LIS {
    public static int lis(int[] arr) {
        Map<Integer, Integer> ends = new HashMap<>(100);
        int longest = 0;

        for (int val : arr) {
            List<Integer> prefixLengths = new ArrayList<>(100);
            for (int j = 1; j <= longest; j++) {
                if (arr[ends.get(j)] < val) {
                    prefixLengths.add(j);
                }
            }

            int length = !prefixLengths.isEmpty() ? prefixLengths.get(prefixLengths.size() - 1) : 0;

            if (length == longest || val < arr[ends.get(length + 1)]) {
                ends.put(length + 1, val);
                longest = Math.max(longest, length + 1);
            }
        }

        return longest;
    }
}
```

Fig. 3.20. LIS refactored for reusability.

improving quality and reusability, there were a few notable changes specific to the reusability attribute. Fig. 3.20 indicates the code of LIS.java after refactoring it for reusability. In the original LIS.java code, the expression "Collections.max(prefix_lengths)" is utilized to find the maximum value in the "prefix_lengths" list. However, in the refactored code, the expression "prefixLengths.get(prefixLengths.size()-1)" is used instead to retrieve the last element in the "prefixLengths" list, assuming the list is sorted in ascending order. This modification achieves the same outcome without relying on the Collections class. Additionally, in the original LIS.java code, the statement "ends.put(length + 1, i)" is used to insert values into the "ends" map, with "i" representing the index in the array. In contrast, the refactored code employs "ends.put(length + 1, val)", where "val" represents the actual value from the array. While the original code stores indices in the "ends" map, the refactored code stores the corresponding values at those indices. The choice between using indices or values as map values depend on the specific requirements and logic of the implemented LIS algorithm, and neither approach is inherently better than the other.

Understandability:

Similar to the design size improvements, when we asked ChatGPT to refactor the LIS.java code to enhance quality and understandability conveyed in Fig. 3.21, the conditional expression in the original code "!prefix_lengths.isEmpty()?Collections.max(prefix_lengths):0" was altered to "prefixLengths.isEmpty()?0:Collections.max(prefixLengths)" in the refactored code. This modification preserves the same functionality of the code but adheres to a more conventional format for conditional

With no explanation refactor the Java code to improve its quality and understandability:

```java
package correct_java_programs;

import java.util.ArrayList;
import java.util.Collections;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

public class LIS {
    public static int lis(int[] arr) {
        Map<Integer, Integer> ends = new HashMap<>(100);
        int longest = 0;
        int i = 0;

        for (int val : arr) {
            List<Integer> prefixLengths = new ArrayList<>(100);

            for (int j = 1; j < longest + 1; j++) {
                if (arr[ends.get(j)] < val) {
                    prefixLengths.add(j);
                }
            }

            int length = prefixLengths.isEmpty() ? 0 : Collections.max(prefixLengths);

            if (length == longest || val < arr[ends.get(length + 1)]) {
                ends.put(length + 1, i);
                longest = Math.max(longest, length + 1);
            }

            i++;
        }

        return longest;
    }
}
```

Fig. 3.21. LIS refactored for understandability.

expressions. Apart from this modification, no other unique changes were made. The remaining alterations included renaming variables, reformatting the code, changing import statements, using the diamond operator, and switching the data type of "prefixLengths" to List, but these alterations are common amongst the other attributes.

3.1.11. *Survey to assess ChatGPT's refactoring capabilities*

To achieve a deeper understanding of ChatGPT's refactoring capabilities in relation to human programmer abilities, we conducted a user study with 15 students studying computer science and related fields. In this survey, we provided five randomly selected code segments and included the code before and after ChatGPT's refactorization. We asked the survey participants to carefully inspect both code segments and to determine whether ChatGPT's refactoring techniques were beneficial. Participants then had to choose between eight multiple-choice options indicating their level of agreement with the refactored code generated by ChatGPT. The eight choices ranged from 0 indicating complete disagreement to 7 indicating complete agreement.

Based on all five code segments, 42.9 % of participants evaluated ChatGPT's refactoring capabilities at a 6 expressing that they mostly agree with ChatGPT's changes. Another 42.9 % of participants chose 5 meaning that they neutrally agree with ChatGPT's refactor techniques, 7.1 % chose 4 for slightly agreeing, and 7.1 % chose 7 for strongly agreeing. These results demonstrate that most participants believe that ChatGPT did an adequate job of refactoring, and no one believed that ChatGPT had a negative impact on the code segments. The lowest number of agreements was 4 which means that all participants believed that at least half of the refactoring techniques that ChatGPT used were beneficial. However, 92.9 % decided that there still remains a need for improvement and additional refactoring changes should be made to the code segments even after ChatGPT refactored them. Only 7.1 % of participants believed that there were no other changes that needed to be made after ChatGPT's refactorization.

In addition to asking participants to select a number to represent their level of agreement, we requested them to share any suggestions on how ChatGPT can be improved for code refactoring. Most suggested that ChatGPT should be better equipped to address complex code segments and it should be trained with a more extensive range of data to improve its accuracy. Many also recognized that ChatGPT's edits sometimes had a negative impact on readability and understandability because its refactoring techniques increased the complexity of the code. Overall, participants agreed with ChatGPT's refactoring modifications and recognized its potential as a refactoring tool. However, most participants believe that ChatGPT is not perfect and improvements can be instituted for ChatGPT's refactoring performance to be better optimized and accurate.

3.2. *RQ2: Does ChatGPT maintain the functionality of the refactored code?*

There were many reasons why the refactored code did preserve the behavior of the original code, but these 6 reasons were seen repeatedly in all 312 out of 321 refactored files.

The first reason was the declaration and importation of necessary packages and classes. In Fig. 3.24, it can be seen that ChatGPT imported all of the necessary classes from the "java.util.*" package used in the original code segment shown in Fig. 3.23, which includes the Hashset, Queue, and Set. This behavior does not change the functionality of the code. ChatGPT used the necessary packages for the code rather than utilizing the whole package when it was not necessary.

The second reason is the declaration of classes to create objects. Fig. 3.24 depicts how 2 public classes were declared compared to the 3 public classes in Fig. 3.23.

The third reason is that ChatGPT created variables using Java naming conventions. For example, in Fig. 3.23, the public class BREADTH_FIRST_SEARCH does not follow Java naming conventions, which is why ChatGPT renamed it to BreadthFirstSearch. Even though this change does not alter the functionality of the code, it makes it easier to read.

The fourth reason is the creation or use of methods to make it easier to perform smaller functions. If a method was created, then ChatGPT followed the Java naming conventions. As depicted in Fig. 3.24, the node "node" used the poll() method instead of the removeFirst() method in Fig. 3.23. The reason this was done is because poll() is a method of Queue while removeFirst() is a method used for linked lists. It is better to use poll() since the code works with Queues.

The fifth reason is that ChatGPT does not declare and initialize variables at the same time. It will declare the variable first, and then in a new line initialize it. For instance in Fig. 3.24, ChatGPT declared the Queue node "queue" as an ArrayDeque(), and in the next line initialized it using the add() method. This allows for the object to be initialized later in the code if it is not known what value this object will hold. The last reason is iterative calculations. ChatGPT uses while loops and for loops to perform recursive operations. In the case of the code used in Figs. 3.23 and 3.24, the while loop has remained the same which portrays that the behavior of the while loop as preserved. However, there were some cases

```java
package correct_java_programs;
import java.util.*;
import java.util.ArrayDeque;

import java_programs.Node;

/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

/**
 *
 * @author derricklin
 */
public class BREADTH_FIRST_SEARCH {

    public static Set<Node> nodesvisited = new HashSet<>();

    public static boolean breadth_first_search(Node startnode, Node goalnode) {
        Deque<Node> queue = new ArrayDeque<>();
        queue.addLast(startnode);

        nodesvisited.add(startnode);

        while (!queue.isEmpty()) {
            Node node = queue.removeFirst();

            if (node == goalnode) {
                return true;
            } else {
                for (Node successor_node : node.getSuccessors()) {
                    if (!nodesvisited.contains(successor_node)) {
                        queue.addFirst(successor_node);
                        nodesvisited.add(successor_node);
                    }
                }
            }
        }
        /**
         * The buggy program always drops into while(true) loop and will not return false
         * Removed below line to fix compilation error
         */
        return false;
    }

}
```

Fig. 3.23. Original code segment BREADTH_FIRST_SEARCH.java.

```java
package correct_java_programs;

import java.util.ArrayDeque;
import java.util.HashSet;
import java.util.Queue;
import java.util.Set;

import java_programs.Node;

public class BreadthFirstSearch {
    public static boolean breadthFirstSearch(Node startNode, Node goalNode) {
        Set<Node> nodesVisited = new HashSet<>();
        Queue<Node> queue = new ArrayDeque<>();
        queue.add(startNode);
        nodesVisited.add(startNode);

        while (!queue.isEmpty()) {
            Node node = queue.poll();

            if (node == goalNode) {
                return true;
            } else {
                for (Node successorNode : node.getSuccessors()) {
                    if (!nodesVisited.contains(successorNode)) {
                        queue.add(successorNode);
                        nodesVisited.add(successorNode);
                    }
                }
            }
        }

        return false;
    }
}
```

Fig. 3.24. Refactored code segment BREADTH_FIRST_SEARCH for performance.

where if there was an if-statement used to perform recursive operations, ChatGPT would use a while or for loop to perform the same operation.

Overall, most of the changes made by ChatGPT did not change the functionality of the code, which is why ChatGPT was able to preserve the behavior of 311 segments (97.2 %) out of the 320 refactored code segments.

3.2.1. Reasons that the code behavior was not preserved in refactored segments

There were 3 main reasons as to why the refactored code segments did not preserve the behavior and logic of the original files.

The first reason is that the refactored code makes a call to the wrong variable, whether that be from a name change or just a mistake made by the AI. This can be seen in Figs. 3.25 and 3.26 where recursive calls were passed through the wrong variable creating an infinite loop causing the code to break from the behavior of the original segment. The first reason for making a call to the wrong variable was why 6 of the 9 code segments did not preserve the behavior of the original with all 6 segments being refactored versions of PossibleChange.java for different quality attributes.

The next reason that behavior was not preserved was that in refactored code segments the parameters for for loops were “less than or equal” whereas, in the original code, the parameters were just “less than.” This modification changed the number of times a for loop was

```java
package correct_java_programs;
import java.util.*;
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

/**
 *
 * @author derricklin
 */
public class POSSIBLE_CHANGE {
    public static int possible_change(int[] coins, int total) {
        if (total == 0) {
            return 1;
        }
        if (total < 0 ||coins.length==0) {
            return 0;
        }

        int first = coins[0];
        int[] rest = Arrays.copyOfRange(coins, 1, coins.length);
        return possible_change(coins, total-first) + possible_change(rest, total);
    }
}
```

Fig. 3.25. Original PossibleChange.java code segment.

```java
package correct_java_programs;

import java.util.Arrays;

public class PossibleChange {
    public static int possibleChange(int[] coins, int total) {
        if (total == 0) {
            return 1;
        }
        if (total < 0 || coins.length == 0) {
            return 0;
        }

        int first = coins[0];
        int[] rest = Arrays.copyOfRange(coins, 1, coins.length);
        return possibleChange(coins, total - first) + possibleChange(rest, total);
    }
}
```

Fig. 3.26. PossibleChange.Java segment that was refactored for performance.

iterated through thus altering the behavior of the code. The changing of loop parameters was the reason 2 out of the 9 segments did not preserve the behavior.

The last reason that behavior was not preserved from the original file to the refactored version is the fact that in the refactored file one of the variables was changed from an ArrayList to a List and in another section of the file there was a call to the wrong variable when updating the List.

3.3. RQ3: Is ChatGPT capable of generating comments or documentation for the refactored code segments, providing insights into intent, instructions, and impact?

For the majority of the 320 refactored code segments ChatGPT was able to describe the intent, instructions, and impact. For example, when ChatGPT was asked to provide comments for the code shown in Fig. 3.28 based on the original code in Fig. 3.27 it was able to provide accurate comments for each of the three things we asked. The documentation that ChatGPT provided for Fig. 3.28 was fairly basic. It said the intent was to refactor the file, that there were improved naming conventions and formatting for instructions, and that the impact was to increase the code readability and maintainability. This was a very common occurrence for the comments made by ChatGPT for the refactored files. So while the majority of the files did have comments (and only 1 file that didn’t have comments for more than one of the 3 parts), the comments were pretty simple.

4. Goals/Motivations

Of the 320 files that ChatGPT was asked to provide a comment to accurately describe the goals, there were only 4 files (1.2 %) that it did not. For 3 of the 4 files ChatGPT only provided a comment that described the overall purpose of the function rather than the goal/motivation of refactoring. For the other file the AI just simply did not provide a comment to describe the goals of refactoring. Additionally, the 4 files that did not have an accurate goal statement had accurate statements for

```java
package correct_java_programs;
import java.util.*;

import java_programs.Node;
import java_programs.WeightedEdge;

import java.lang.Math.*;

/**
 *
 * @author Angela Chen
 */
public class SHORTEST_PATHS {

    // Define Infinite as a large enough value. This value will be used
    // for vertices not connected to each other
    final static int INF = 99999;


    public static Map<String, Integer> shortest_paths(String source, Map<List<String>,Integer> weight_by_edge) {
        Map<String,Integer> weight_by_node = new HashMap<String,Integer>();
        for (List<String> edge : weight_by_edge.keySet()) {
                weight_by_node.put(edge.get(1), INF);
                weight_by_node.put(edge.get(0), INF);
        }

        weight_by_node.put(source, 0);
        for (int i = 0; i < weight_by_node.size(); i++) {
            for (List<String> edge : weight_by_edge.keySet()) {
                int update_weight = Math.min(
                            weight_by_node.get(edge.get(0))
                                    + weight_by_edge.get(edge),
                            weight_by_node.get(edge.get(1)));

                weight_by_node.put(edge.get(1), update_weight);
            }
        }
        return weight_by_node;
    }
}
```

Fig. 3.27. Original SHORTEST_PATHS.java code.

```
package correct_java_programs;

import java.util.HashMap;
import java.util.List;
import java.util.Map;

public class ShortestPaths {
    final static int INF = 99999;

    public static Map<String, Integer> shortestPaths(String source, Map<List<String>, Integer> weightByEdge) {
        Map<String, Integer> weightByNode = new HashMap<>();

        for (List<String> edge : weightByEdge.keySet()) {
            weightByNode.put(edge.get(0), INF);
            weightByNode.put(edge.get(1), INF);
        }

        weightByNode.put(source, 0);

        for (int i = 0; i < weightByNode.size(); i++) {
            for (List<String> edge : weightByEdge.keySet()) {
                int updateWeight = Math.min(
                        weightByNode.get(edge.get(0)) + weightByEdge.get(edge),
                        weightByNode.get(edge.get(1)));

                weightByNode.put(edge.get(1), updateWeight);
            }
        }

        return weightByNode;
    }
}
```

Fig. 3.28. SHORTEST_PATHS.java refactored for performance.

both of the other parts and all of the files were refactored to improve cohesion.

The goals ChatGPT was able to accurately provide for the 316 files were generally similar. ChatGPT would state that it made refactoring changes to improve the code's readability, maintainability, type handling, clarity, simplicity, and/or consistency. Other goals included adhering to Java conventions, creating helper methods to make calculations easier to read and reuse, and removing unnecessary type casting and import statements. However, ultimately, the most common goal was to optimize class, method, and variable names to enhance readability. The goals that ChatGPT named often correlated with the impact ChatGPT reported the refactoring changes had as well, meaning ChatGPT was able to successfully modify the code segments for its intentions.

4.1. Refactoring Changes/Instructions

ChatGPT was unable to provide accurate refactoring changes/instructions for 5 (1.6 %) out of the 320 files. Of the 5 files only 1 of them was also missing an accurate comment describing the impact, the other 4 files were only missing accurate refactoring changes comments. For the files missing accurate instruction comments, 1 of the files was given a comment that contained refactoring changes that ChatGPT fabricated, and the other 4 were not given comments at all. In terms of quality attributes that the files were refactored for 1 of the files was refactored to improve complexity, 1 for cohesion, and 3 for reusability.

The refactoring changes that ChatGPT accurately identified for the 315 files were extremely similar to those reported in RQ1. For example, the most frequent refactoring change that was reported for almost all of the files was renaming classes, methods, and variables to follow standard Java naming conventions, which was also the most popular change that we observed in RQ1. In RQ1 we realized that all 8 quality attributes used renaming as a tool of refactoring. Other modifications ChatGPT identified included adding indentation and reformatting the code, removing unused imports, simplifying calculations with helper methods and additional variables, adjusting loop conditions, and switching variable types. ChatGPT also described adding comments, simplifying loop implementations, replacing the wrapper class with primitive data types, using the diamond operator, moving variable declarations inside loops, combining variable declarations and instantiations, replacing.equals() method with two equals signs (==), and converting enhanced for loops to a standard for loops or vice versa. All of these refactoring changes have been observed in RQ1, meaning that ChatGPT was able to accurately determine what modifications have been made between the original and refactored code segments. In summary, our observation indicates that ChatGPT offers generic refactoring operations but does not provide specific refactoring operations as defined by Fowler (Chaparro et al., 2014).

4.2. Impacts

There was only 1 file (0.3 %) that the AI did not provide a comment that accurately described the impact of the refactoring. For this file, ChatGPT just did not provide a comment mentioning the impact at all. Additionally, the single file that did not have a comment describing the impact was a file that was refactored to improve cohesion. Thus making cohesion the quality attribute with the most missing comments at 5 files followed closely by reusability at 3 and then complexity at 1 file leading to a total of 9 files without at least one of the comments asked for.

4.2.1. Performance

Using the 40 files that ChatGPT refactored to improve quality and performance, as illustrated in Table 3.33, the most prominently improved quality attribute that ChatGPT determined after refactoring was readability. Among the 40 files analyzed, ChatGPT identified readability improvements in 39 instances. The next most notable improvement was in maintainability, detected in 37 files of all improved quality attributes identified by ChatGPT. The remaining quality attributes, ranked in descending order, were understandability, observed in 18 files, performance, identified in 10 files, flexibility, present in 2 files, and reusability, noticeable in 1 file. Despite refactoring all 40 files to improve performance, ChatGPT recognized the impact of performance in only 10 files.

4.2.2. Complexity

For the 40 files that ChatGPT refactored to improve quality and complexity, similar to the performance attribute, the two predominant quality attributes identified as improved for most files were readability and maintainability, evidently in Table 3.33. Readability was recognized in 39 of the files and maintainability was identified in 36 files. The remaining quality attributes, ranked from greatest to least, were performance with 10 files, code consistency with 8 files, flexibility with 3 files, understandability and reusability each with 2 files, and complexity with 1 file. However, even though the 40 files were refactored to improve complexity, ChatGPT only determined that 1 file had demonstrated improved complexity after the refactoring process.

4.2.3. Coupling

After analyzing the files that ChatGPT refactored to enhance quality and coupling, similar to the preceding attributes, Table 3.33 shows that the two primary quality attributes that ChatGPT saw improvement in for most files were readability with 34 files and maintainability with 33 files. Following these attributes, understandability ranked next with 21 files. Subsequently, reusability showed enhancement in 12 files, flexibility in 4 files, and performance in 3 files. Unexpectedly, ChatGPT did not identify any improvements for coupling, despite the code segments being refactored with a focus on improving coupling.

4.2.4. Cohesion

Upon instructing ChatGPT to assess the impact on quality for the files refactored to improve quality and cohesion, the two most prevalent attributes evident in Table 3.33, accounting for approximately 75 % of the results, were readability with 38 files and maintainability with 32 files. The remaining quality attributes, listed in descending order, were understandability with 11 files, flexibility with 5 files, performance with 4 files, and reusability with 2 files. Although the refactoring efforts of these files were aimed at cohesion improvement, similar to the results for coupling, ChatGPT did not identify any files that demonstrated enhanced cohesion after refactoring.

4.2.5. Readability

After asking ChatGPT to determine the improved quality attributes for the files refactored to enhance quality and readability, the most surprising finding was that maintainability appeared in the highest number of files shown in Table 3.33, with 39 instances of all the quality

Table 3.33
This table indicates the number of files each quality attribute was mentioned for the 40 files refactored for all quality attributes.

	Readability	Maintainability	Performance	Understandability	Flexibility	Reusability	Complexity	Code consistency
Performance	39	37	10	18	2	1	0	0
Cohesion	38	32	4	11	5	2	0	0
Complexity	39	36	10	2	3	2	1	8
Coupling	34	33	3	21	4	12	0	0
Readability	36	39	3	18	1	3	0	0
Design Size	38	33	2	22	3	2	0	0
Reusability	33	37	4	19	7	2	0	0
Understandability	39	30	2	19	5	2	0	0

attributes identified by ChatGPT for the 40 files. Following maintainability, the next most prominent quality attribute was readability, identified in 36 files. Subsequently, understandability was recognized in 18 files, while performance and reusability each appeared in 3 files, and flexibility in 1 file. ChatGPT was able to determine that 36 of the 40 files that were refactored to improve readability had indeed achieved the desired improvement.

4.2.6. Reusability

For the 40 files that ChatGPT refactored to improve quality and reusability, as illustrated in Table 3.33, the two primary quality attributes ChatGPT determined were improved following refactoring was readability with 33 files and maintainability with 37 files. The next most notable improvement was in understandability, detected in 19 files of all improved quality attributes identified by ChatGPT. The remaining quality attributes, ranked from greatest to least, were flexibility with 7 files, performance with 4 files, and reusability with 2 files. Despite refactoring all 40 files to improve reusability, ChatGPT recognized the impact of reusability in only 2 of the 40 files.

4.2.7. Design size

For the files ChatGPT refactored to improve quality and design size, the two dominant quality attributes identified as improved for most files were readability and maintainability, as demonstrated in Table 3.33. Readability was recognized in 38 of the files and maintainability was identified in 33 files. The remaining quality attributes, ranked in descending order, were understandability with 22 files, flexibility with 3 files, reusability with 2 files, and performance with 2 files. However, even though the 40 files were refactored to improve design size, ChatGPT determined that none of the files demonstrated improvements in design size after the refactoring process.

4.2.8. Understandability

After instructing ChatGPT to assess the impact on quality for the files refactored to improve quality and understandability, the two most prevalent attributes as shown in Table 3.33, were readability with 39 files and maintainability with 30 files. The remaining quality attributes, listed from greatest to least, were understandability with 19 files, flexibility with 5 files, performance with 2 files, and reusability with 2 files. Although the refactoring efforts of the 40 files were aimed to improve understandability, ChatGPT identified 19 of the files as having demonstrated enhanced understandability after refactoring.

After testing for all 8 quality attributes, we analyzed the documentation for the impacts ChatGPT had on the refactored code. We noticed that the most common quality attributes impacted were readability and maintainability. While readability was among the 8 quality attributes we tested, maintainability was not. The main reason why this occurred is because of the fact that the code segments tested were refactored to improve quality and [quality attribute]. The fact that the code was improved is the reason why "maintainability" came up as one of the top impacted attributes. The same can be said for readability. However, in the case of readability, it was to improve the understanding of the code. Table 3.33 depicts readability, maintainability, and understandability as the top 3 impacted attributes for performance. This same pattern can be seen for the other quality attributes with a few changes. For example, in the case of complexity, performance outranked understandability. Additionally, in the cases of readability and reusability, maintainability outranked readability. Overall, ChatGPT is able to accurately identify the goal(s)/motivation(s), refactored change(s)/instruction(s), and impact(s)/quality attribute(s) of the refactored code.

5. Takeaways

5.1. ChatGPT's keyword sensitivity

We observed the impact of solely using the term "quality" versus incorporating both "quality and [quality attribute]." We discovered that when the prompt only included the term "quality," ChatGPT consistently generated the following default output statement: "Overall, these changes aim to improve readability, follow coding conventions, and use more descriptive names." These modifications were generally minor, and it could be expected that the code's behavior would always remain consistent since no substantial modifications were made. On the other hand, when the prompt used both keywords "quality" and "[quality attribute]," ChatGPT did not always provide a default output statement. However, we noticed that ChatGPT often generated a variation of the following statement: "Overall, these changes aim to improve the readability, adhere to Java naming conventions, and simplify the logic while maintaining the functionality of the original code." This suggests that by explicitly including quality-related keywords in the prompt, ChatGPT was inclined towards performing more significant changes. Therefore based on our analysis, our main conclusion was that ChatGPT will do major refactoring if we enforce some quality attribute keywords.

5.2. ChatGPT's ability to suggest helpful refactoring changes

Most of the time, ChatGPT will offer improved versions of the provided code, even if it is as simple as suggesting enhancements to variable names for better clarity. Therefore, ChatGPT can be a valuable tool for assisting programmers by proposing various modifications, ranging from minor to significant, to enhance their code. As a programmer, it is pivotal to consider an external perspective when reviewing your code since others will often examine the code you have written, and it is essential that they can easily comprehend it. While not all suggestions from ChatGPT may be crucial or applicable, our research has shown that certain changes such as loop conversions, additions of helper functions to simplify calculations, and removal of unused and redundant statements can indeed be valuable. Changes like these can significantly improve runtime, memory storage, and readability which are crucial for high-quality code. ChatGPT is proficient at providing quick and helpful tips for debugging and improving one's code, particularly regarding syntax, naming, and adherence to Java conventions.

5.3. ChatGPT's changes based on the quality attribute used in the prompt

Our findings revealed that there were generally no significant variations in the modifications made by ChatGPT for each attribute in the prompt. However, we did observe some discernible differences.

Although the changes we observed did not apply uniformly to every code segment that was refactored to improve a specific quality attribute, they occurred at least once. Interestingly, the terms performance and reusability prompted more specific changes from ChatGPT, while the others appeared to be more interchangeable.

Although we identified only 2 distinct changes exclusively associated with the performance attribute, it shared modifications with another attribute twice and with 2 other attributes 4 times. For example, replacing an enhanced for loop with a regular for loop was a change observed in the performance, coupling, and design size attributes. The performance attribute shared 1 modification with reusability and complexity attributes separately. The performance attribute also shared a modification with the pairs: complexity and cohesion, coupling and design size, coupling and reusability, and complexity and reusability.

Reusability exhibited a few changes that were unique to it, with a total of 2 modifications. In contrast, excluding the performance attribute, the other 6 attributes had only one distinct change when compared to the others. The reusability attribute shared modifications with another attribute once and with 2 other attributes 2 times.

It's also worth noting that all eight quality attributes shared eight common modifications. These changes included actions such as renaming classes, methods, and variables, formatting the code, adjusting import statements, and adhering to Java conventions. One surprising discovery was that all attributes involved changing data types from wrapper classes to primitive types. These eight changes that all attributes shared consisted of most of the changes ChatGPT made to each code segment. Therefore, when asking ChatGPT a prompt, the choice of words generally had little impact on the changes it made. However, there were instances where ChatGPT seemed to give more weight to certain words than others. For instance, when asking ChatGPT to refactor the code to improve performance versus understandability, the changes made are notably distinct. Refactoring for performance involves converting calculations to be more concise, modifying loop indexes, and removing redundant statements. These changes aim to enhance the code's efficiency and overall performance. Conversely, when requesting refactoring for improved understandability, the focus is on aligning variable declarations and assignments for better consistency and adding comments. As a result, there are fewer changes made in comparison to performance-oriented refactoring, and they primarily revolve around reformatting and readability rather than substantial code alterations. It is worth noting that while all these changes offer benefits, those associated with performance had a greater impact.

5.4. ChatGPT's effectiveness for refactoring each attribute according to PMD results

After reviewing the pie charts illustrating the PMD violations for each of the eight quality attributes, we derived several conclusions. Notably, the category with the highest percentage of violations across all attributes was Code Style, accounting for over half of all the pie charts. Within the Code Style violations, the quality attribute with the greatest percentage was reusability at 64.6 %, while readability had the lowest percentage at 55.1 %, resulting in a variation of approximately 9 % for Code Style violations. Another observation we made pertained to readability's percentage of Best Practices violations, which exceeded the others by at least 3 %. This means that readability exhibited a higher number of violations under the Best Practice category compared to the other attributes. Specifically, readability had 14.5 % of Best Practices violations, while the next highest was understandability at 11.5 %, which demonstrates the minimum difference of 3 %. Across all attributes, we consistently observed 5 error-prone violations and 9 multithreading violations, comprising slightly less than 3 % of each pie chart. Furthermore, when comparing the percentages of violations in the Design, Documentation, Performance, Error-prone, and Multithreading categories across all attributes, the differences were within a 2 % range, indicating a relatively similar distribution. For instance, the highest percentage of Documentation violations occurred in the complexity attribute at 18.9 %, while the lowest was in readability at 17.2 %, representing a 1.7 % range between all Documentation percentages. In summary, the results of the PMD violations displayed a general similarity across attributes but with slight variations and differences.

5.5. ChatGPT's limited understanding of the broader context

While ChatGPT demonstrates its capability to propose beneficial refactoring techniques by analyzing and comprehending provided code segments, it has limitations in grasping the larger context in which these segments are utilized. Due to this limitation, there have been instances where ChatGPT makes suggestions based on misunderstandings or false assumptions about the code. In contrast, human programmers possess the knowledge and understanding necessary to consider such broader contextual information. Therefore, ChatGPT may occasionally make errors and suggest fixes that are neither applicable nor necessary.

5.6. How our findings relate to other research findings

There have been numerous published studies focusing on researching ChatGPT, as described in detail in the Related Works section of our paper. These studies reveal both similarities and differences between our findings and theirs. For example, (Haque & Li, 2023) also evaluated ChatGPT's ability to enhance code segments. The researchers discovered that while ChatGPT can assist in the debugging process, it has certain limitations that prevent full reliance on its capabilities. These limitations include a limited understanding of context, dependence on training data, and a restricted knowledge base, as well as an inability to address complex errors. As outlined in the takeaways and limitations sections of our paper, we have encountered similar limitations in our own research regarding ChatGPT's limited understanding of the broader context, reliance on its training data, insufficiency to fix complex errors, and unpredictability of its output. Additionally, this paper highlights the advantages of employing ChatGPT and emphasizes its ability to suggest beneficial refactoring changes. Our research demonstrates how ChatGPT can provide and execute refactoring changes to enhance code quality. Although this particular paper does not offer an in-depth analysis of ChatGPT's effectiveness in improving code quality, we concur that ChatGPT does provide some benefits. Furthermore, (Tian et al., 2023) identified ChatGPT's ability to handle typical programming challenges. However, it points out that ChatGPT has a limited attention span and struggles to accurately solve problems when presented with long descriptions. This aligns with our own findings, as we observed that ChatGPT performs well in identifying and modifying basic bugs and code that does not adhere to Java conventions. However, its performance deteriorates when overloaded with lengthy code segments and poorly crafted prompts. While our findings align with those of other studies, what sets ours apart is our discovery of ChatGPT's sensitivity to the wording of prompts. We also realized that the most effective attributes to improve and request ChatGPT to refactor the code sufficiently are performance and reusability.

6. Limitations

6.1. Complexity and size of the dataset

The main source behind many limitations in our research is the complexity and size of the dataset we used. Our dataset consisted of 40 files, each of which contained only one class and for the most part, contained one or two methods. The methods themselves were not overly complex; in fact, the majority of the files contained between 15 and 35 lines of code. The simplicity of the files presented quite a few problems with the main one being ChatGPT's ability to refactor the code. The simplicity of the code provided to ChatGPT made it so there wasn't much that the AI could do regarding refactoring. For at least 5 out of the

40 code segments, all ChatGPT did to improve the code was rename a handful of variables and methods. To try and combat this limitation we asked ChatGPT to refactor the code segments on multiple different quality attributes in hopes of drawing out different refactoring techniques.

Another way that the lack of file complexity hindered our research was in our data analysis. We originally had a research question where we asked ChatGPT to evaluate the original and refactored code on different software metrics. However, the simplicity of the code and the way it resulted in ChatGPT only making minuscule changes to the segments meant that there were very few differences between the metric values for the original and refactored code. There was only one metric where you could definitely conclude that the refactoring done by ChatGPT improved the quality of the code. For all of the other metrics there either was not enough data to tell (some of the metrics only had values of 0 s or 1 s because the files only contained 1 class) or there was no significant difference in the values with both the refactored and original code having similar means and ranges. We worked around this limitation by completely scraping that research question and developing a new third question to perform our data analysis on. The new third question we came up with portrays ChatGPT's ability to provide comments and documentation for the refactored code segments. In this research question, we provided ChatGPT with the original and the refactored code segments and asked ChatGPT to provide comments and documentation of the code to describe intent, instructions, and impacts. While ChatGPT was able to provide comments for these characteristics, there was a limit to how complex ChatGPT's responses were. For example, for the intent of the refactored code, ChatGPT would respond with the goal of the function, which is what we were hoping for, but for impact, we were seeking the quality attribute the code was improved for. For instance, when we gave ChatGPT the refactored code for performance, it returned the impact as performance for some of the files, but not all of them. Rather it returned maintainability and readability as the most common quality attributes. There could be many reasons for this kind of response, but it goes to show that ChatGPT has limits.

6.2. ChatGPT's unpredictability

The second main source of limitations in our study was the variability of ChatGPT. You can ask ChatGPT a question with the exact same wording 100 times and it will give you a different answer every time. The answers will be similar but not the same. We did our best to combat this problem by doing prompt engineering for each question and picking prompts partly based on how consistent its answers were.

Similarly, another problem arises from the way ChatGPT builds off of the previous answers in the chat log. This means that when asking ChatGPT to refactor code fragments it tends to use previous techniques on future fragments. So you can ask the exact same question, once in a new window and once in a log previously used to refactor code, and will get completely different responses. We overcome these limitations by creating a new chat log every time we asked a new prompt to minimize how much the previous answer impacted future ones.

These two problems and our experiences with ChatGPT's variability as a whole throughout our research are similar to what (Ma et al., 2023) experienced during their RQ2. The researchers in that study noted that ChatGPT tended to "hallucinate" when they asked the AI about code semantic structures. ChatGPT would analyze the structures it was asked to interpret and then make up facts to explain the code. The AI would create and refer to non-exist statements and nodes which the other researchers identified as a threat to ChatGPT's reliability.

6.3. ChatGPT ability to refactor the code

Across all 40 code segments for each of the 8 quality attributes, ChatGPT successfully refactored each one with the exception of one. However, the nature of the changes varied. In some cases, the modifications primarily involved renaming variables and reformatting the code to adhere to Java conventions. In other instances, the suggested changes extended to transforming enhanced loops into for loops and switching data structures. Nevertheless, ChatGPT consistently provided valuable and applicable suggestions, ranging from minor to more substantial alterations, for almost all code segments. While ChatGPT can complete the tasks assigned to it, there are limitations to ChatGPT's responses to each research question. The major limitations noticed after completing each question are ChatGPT's complexity and size of data, ChatGPT's unpredictability, and ChatGPT's changes based on the quality attribute used in the prompt. The complexity of the code segments provided to ChatGPT was simple for ChatGPT to do much refactoring since, for the most part, the code segments were in their simplest form. Additionally, with RQ2, ChatGPT had a limit to the size of the data since the prompt used required us to enter the prompt to be entered two times to make a comparison between the original code and the refactored code. Furthermore, ChatGPT's unpredictability was a limitation because there were cases where the answers between quality attributes would be the same. After all, ChatGPT would use the answer from the previous attribute to answer the new one. This meant that a new "chat" or tab had to be used to ensure that ChatGPT did not try to use the answers from previous attributes.

6.4. ChatGPT reliance on its training data set and limitations for addressing complex errors

The efficacy of ChatGPT is intricately linked to the quality of its training data set. Drawing from the patterns and code structures present in its training data, ChatGPT can recommend refactoring solutions. Nevertheless, when the data set lacks diversity and complexity in code segments, ChatGPT may face challenges in suggesting refactoring techniques for intricate user input. Inadequate and less complex data can limit the overall performance of ChatGPT. During our investigation for RQ1 and RQ2, we observed that the suggested changes by ChatGPT remained generally consistent across all 40 files for the eight different quality attributes. These changes primarily revolved around variable renaming, code reformatting, and data type modifications. Also, for RQ3 we recognized that ChatGPT only determined that 8 quality attributes were improved upon after refactoring and the 8 quality attributes were not the same as the 8 we refactored the code for. The two primary attributes were readability and maintainability which suggests that ChatGPT has a limited ability to refactor code segments to improve more sophisticated quality attributes like cohesion and coupling because ChatGPT was not able to recognize their improvement in the refactored code. Although our provided code segments were not highly intricate, this suggests that ChatGPT's training data set may not have been exceptionally complex or extensive. As a result, ChatGPT frequently failed to identify larger-scale changes that could optimize code refactoring more effectively.

6.5. ChatGPT documentation accuracy

While ChatGPT was able to provide concise goal(s)/motivation(s), refactor change(s)/instruction(s), and impact(s)/quality attribute(s) regarding the refactored code, there were some documentation inaccuracies. For example, when ChatGPT was provided with the prompt for RQ3, it would return long responses for some of the quality attributes such as performance, cohesion, coupling, etc while short ones for others including complexity, design size, etc. There is no specific reason why ChatGPT provided such variations. However, ChatGPT was able to accurately determine the goal of the refactored code, the refactored changes made, and the quality attribute(s) impacted by these changes. While the main refactoring change was renaming, ChatGPT was able to correctly identify these changes, which is vital because it portrays the fact that ChatGPT is capable of identifying ways to refactor code. This does not apply to the impact of quality attributes. When ChatGPT was

asked to commit messages regarding the impact(s)/quality attribute(s), we were looking to see if it could identify the type of quality attribute the original code was refactored for. For instance, we provided ChatGPT with the original code and the refactored code with improved quality and performance. We wanted to see if ChatGPT would realize that the code was refactored to improve performance. This was repeated for all the other quality attributes. However, ChatGPT did not perceive the prompt from this perspective, and as a result, it was able to identify that the overall refactored code was impacted by readability and maintainability. We conjecture that this is a limitation to ChatGPT's understanding, but the results could have been different if the prompt was written to specify this. For now, it is enough to know that ChatGPT is able to correctly identify that a code has been refactored and the changes made to it.

7. Ethics implications of using ChatGPT for code refactoring

While ChatGPT can be a very useful tool to begin the refactoring process, there are several ethical implications one should be aware of before making the decision to use said tool. One of the main ethical concerns around using AI generative technology is the potential for it to lead to job losses. At its current state, ChatGPT poses no significant threat to job security in the software industry; however, as the technology progresses programmers need to be careful not to become too reliant on AI. For instance, if developers consistently delegate crucial coding decisions to ChatGPT, there is a risk of diminishing their hands-on skills and the automation of jobs that were traditionally performed by humans.

Another concern to be aware of with generative AI technology is ownership. If an AI like ChatGPT is asked to refactor code and make significant changes to the original segment, it raises questions about crediting for these alterations. Consider a scenario where ChatGPT suggests a groundbreaking refactor that significantly improves the code structure. The debate arises over whether the credit should go to the dataset ChatGPT was trained on, ChatGPT itself, or the creators of ChatGPT. Resolving ownership and attribution becomes a complex challenge in such situations.

The final ethical implication one should be aware of before deciding to use ChatGPT is the security concerns of such technology. OpenAI, ChatGPT's parent company, saves your data and history from the conversations you have with the AI. So, any code provided to the AI will be saved and used in future responses, meaning any original code you provided to ChatGPT may be reused and recommended to other users limiting the security around your data. Developers should be cautious about sharing sensitive or proprietary code with ChatGPT, considering the potential implications of data security and confidentiality.

These ethical implications are not intended to entirely dissuade users from using ChatGPT to help refactor code but to make users aware of the possible downsides of using generative technology and encourage responsible and informed usage.

8. Future work

For future studies, there are four aspects that we believe could be further explored to strengthen the conclusions from this study and further develop a deeper comprehension of ChatGPT's capabilities. The first is that while reviewing the results of the PMD tool, we noticed that urgent violations were the most common violation types found in all of the quality attributes. Thus, understanding which parts of the refactored code caused the PMD tool to identify the code as urgent violations would be the next step in this project. The results of this study will allow us to understand what caused the code segment to have urgent violations and how we can change the code segment to have warning violations or no violations. Therefore, by focusing on the urgent violations we can gain a better understanding of what ChatGPT lacks in refactoring as it either causes more of these urgent violations or fails to address them.

Furthermore, the second idea we propose is to use metric values to determine if code refactored by ChatGPT can be used in software development. Metric values can provide quantitative data to discern to what extent ChatGPT's refactorization is improving the code segments. Metrics such as lack of cohesion of methods, coupling between objects, cyclomatic complexity, weighted method count, and lines of code are only a couple of possible examples. First, the metrics would be measured in the original code segments and then compared to the metric values for the refactored code segments. This allows us to objectively measure the benefit of ChatGPT's refactoring techniques instead of using our opinions. However, this can only be tested with more complex code segments in order to easily evaluate these values. Due to the limitations we faced regarding the size and complexity of our dataset, our findings highlight the importance of further research into the refactoring capabilities of ChatGPT with a larger and more complex dataset to better understand and quantify the changes between the original and refactored code segments using metrics.

Another possible topic to explore is studying ChatGPT's effect on producing code that improves security. After the results from the PMD tool were analyzed, we recognized a lack of violations from the security category due to our data set not violating security initially. Thus, using the PMD tool, ChatGPT can be tested to see whether or not it can produce code that improves security on a variety of more complex code segments. This study will allow society to better understand whether or not ChatGPT is capable of refactoring complex code relating to security or if this is its limit.

We also believe that the same approach conducted in this research can be used to understand how ChatGPT can improve the quality of code in different languages, such as Python since it is one of the most popular languages. Future work can explore how ChatGPT would be able to identify what refactoring changes need to be made to the code based on the quality attribute(s). After completing this study with different languages, we can determine if ChatGPT has the same abilities in refactoring code as it does in all programming languages or if there are certain languages in which it excels more.

9. Related works

Recent studies utilized ChatGPT for different software engineering tasks. Due to the importance of the topic, several domains for the use of ChatGPT have been proposed, including code quality, programming, testing, bug management, security, and program comprehension. In this section, we are only interested in research on the use of ChatGPT to solve software engineering tasks,

(Ma et al., 2023) evaluated ChatGPT's ability to interact with code segments. Specifically, the paper investigated how the AI was able to comprehend different software engineering tasks. The research focused on ChatGPT's understanding of semantic structures and code syntax by asking the AI to evaluate call graphs, control flow graphs, and abstract syntax trees. The researchers found that ChatGPT was able to understand abstract syntax trees with ease but struggled to understand semantics, especially dynamic ones. They also found that ChatGPT tended to fabricate variables and facts when asked to interpret dynamic semantic structures.

(Haque & Li, 2023) focused on ChatGPT's ability to summarize code and provide an appropriate comment for a given code segment. In their paper, Haque and Li underwent vigorous prompt engineering to find the question that would provide the most accurate and consistent results. After finding a sufficient prompt, they compared the ChatGPT prompt results for a Python dataset to the results given by popular code summarization tools. They evaluated the results on 3 main metrics and found that based on 2 out of the 3 metrics ChatGPT significantly underperformed when compared to current methods of code summarization.

(AlOmar et al., 2023) focused on using static analysis tools like PMD to assist students in improving the quality of code. The researchers

conducted an experiment across 3 academic periods and reviewed 65 submissions on 690 rules. They found that tools like PMD influenced a student's decision to acknowledge or ignore issues in code segments, especially with violations that take a longer amount of time to revise.

(Sun et al., 2023) also evaluated ChatGPT's ability to interact with code segments. In this paper, the researchers seek to find the best way to implement ChatGPT in a way to make the software development process easier. They discovered that while the AI can be used to ease some of the debugging process, ChatGPT has several hard limitations that make it impossible to fully rely on its debugging capabilities. The researchers conclude that ChatGPT can be used as a starting point in the debugging process for software developers but that human intervention is still absolutely necessary.

(Tian et al., 2023) analyzed ChatGPT's ability to be a programming assistant based on its code summarization, code generation, and program repair capabilities. They learned that the AI can handle most of the typical software problems but reaches its limits when it comes to tasks that involve ChatGPT's attention span, especially comprehensive descriptions. The researchers conclude their paper by drawing special attention to the way ChatGPT provides insight into a creator's original intentions when asked to evaluate incorrect code and point out the potential for future work in this area.

(Xia & Zhang, 2023) focused on creating a fully automated conversation-driven program repair using a large language model. The researchers tested this out by feeding ChatGPT different code segments and then evaluating the corrections the AI made. Incorrect patches were combined with the reason for failure and then this information was used to create a new prompt to ask ChatGPT to re-correct the code segment. In the instances where ChatGPT correctly fixed a bug, the AI was asked to generate different variations of the fix. By doing this, the researchers found that they could actually build off of ChatGPT's previous responses and increase the likelihood of receiving an error-free correction for a given code segment.

In summary, while recent studies have delved into ChatGPT's application in various software engineering tasks (e.g., code generation, quality, testing, repair, etc), there is a gap in the exploration of its refactoring capabilities across different internal and external quality attributes (e.g., performance, readability, complexity, etc). Our study aims to complement existing research by scrutinizing ChatGPT's performance in ensuring the correctness and dependability of its outputs for diverse software engineering tasks.

10. Conclusion

ChatGPT is a new form of technology that will allow society to expand in all directions: education, technology, etc. Therefore, it is essential that we understand what it is capable of and what its limitations are. Through this project, we were able to better understand ChatGPT's code refactoring capabilities, ethics, and limits in order to determine whether or not it can be useful to society. Based on our findings for RQ1 and RQ2, we can conclude that ChatGPT is able to refactor and preserve the behavior of the code. For most of the code segments, ChatGPT was able to suggest minor to major modifications to improve the quality of the code. The most common refactoring techniques were adhering to Java conventions by renaming class, variable, and method names, reformatting the code, and adjusting import statements for clarity. This portrays that developers can use ChatGPT to take existing code and refactor it to improve its quality and other attributes. ChatGPT is proficient at providing quick and helpful tips for debugging and improving one's code, particularly regarding syntax, naming, and organization of the code. Additionally, after making these changes, ChatGPT preserves the behavior of the original code which demonstrates its ability to comprehend the code's function and purpose and choose modifications that do not alter it. Even though ChatGPT has limitations on the range of complexity it can interpret, it indicates how ChatGPT has a strong foundation of programming comprehension. Also, based on our findings for RQ3, it is evident that ChatGPT can properly identify the intent, instructions, and impact of refactored code segments when given the original to compare it to. While ChatGPT was not always able to recognize the improvement of the quality attribute that the code was refactored for, improvements for numerous quality attributes were always made. We noticed that the most common quality attributes ChatGPT identified as enhanced were readability and maintainability. While readability was among the 8 quality attributes we tested, maintainability was not. This suggests a limitation of ChatGPT's deep understanding of complex code. It is easier to identify and make improvements for readability and maintainability while it is more difficult for other quality attributes such as coupling and complexity. We know this because ChatGPT was often unable to recognize the refactoring techniques it applied to the original code to improve a certain quality attribute that was asked in the prompt, especially for the more complex quality attributes. This means that ChatGPT is more proficient in refining code for quality attributes regarding readability in comparison to more complicated quality attributes. Thus, ChatGPT is not perfect. Despite the fact that the same prompt was used by each author, we did get different results from ChatGPT when gathering data for our research questions. These unpredictable outcomes made it difficult to truly determine whether or not ChatGPT was capable of understanding the task we gave it. Overall, ChatGPT was able to refactor most of the code provided. It was also able to preserve the behaviors of these code segments and provide documentation. ChatGPT's capabilities have yet to be discovered and its full potential has not been reached. With a better understanding of what it is capable of, the better we will be able to understand the impact ChatGPT will have on society and future technology. After conducting this study, we conclude that ChatGPT has proved to be a beneficial tool for programming as it is capable of providing valuable suggestions even if it is on a small scale. However, human programmers are still needed to oversee these changes and determine their significance. ChatGPT should be used as an aid to programmers since we cannot completely depend on it yet. In summary, the main takeaways are:

- Keyword sensitivity of ChatGPT: Our key finding indicates that ChatGPT tends to perform significant refactoring when specific quality attribute keywords are enforced.
- Ability of ChatGPT to provide helpful refactoring suggestions: Our analysis reveals that ChatGPT consistently offers enhanced versions of provided code, often suggesting improvements to variable names for better clarity. This underscores ChatGPT's potential as a valuable tool for assisting programmers by proposing a range of modifications, from minor to substantial, to enhance their code.
- Effect of quality attribute on ChatGPT's refactoring suggestions: While our study found no significant variations in the modifications suggested by ChatGPT for each quality attribute in the prompt, discernible differences were observed.
- ChatGPT's ability to refactor the code: Across all 40 code segments for each of the eight quality attributes, ChatGPT successfully refactored each one, except for one instance. The nature of these changes varied, from simple renaming of variables and code reformatting to more complex transformations like converting enhanced loops to for loops and switching data structures.
- Limitations of ChatGPT in understanding context: Although ChatGPT demonstrates the ability to propose beneficial refactoring techniques by analyzing provided code segments, it exhibits limitations in comprehending the broader context in which these segments are utilized.

CRediT authorship contribution statement

Kayla DePalma: Investigation, Methodology, Validation, Visualization, Writing – original draft. **Izabel Miminoshvili:** Investigation, Methodology, Validation, Visualization, Writing – original draft. **Chiara**

Henselder: Investigation, Methodology, Validation, Visualization, Writing – original draft. **Kate Moss:** Investigation, Methodology, Validation, Visualization, Writing – original draft. **Eman Abdullah AlOmar:** Conceptualization, Investigation, Project administration, Supervision, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

The link is shared in the manuscript. https://sites.google.com/stevens.edu/chatgptdataanalysis/home.

References

Al Dallal, J., & Abdin, A. (2017). Empirical evaluation of the impact of object-oriented code refactoring on quality attributes: A systematic literature review. *IEEE Transactions on Software Engineering, 44*(1), 44–69.

AlOmar, E. A., AlOmar, S. A., & Mkaouer, M. W. (2023, July 13). *On the use of static analysis to engage students with software quality improvement: An experience with PMD.* arXiv.org.https://arxiv.org/abs/2302.05554.

AlOmar, E. A., Mkaouer, M. W., Newman, C., & Ouni, A. (2021). On preserving the behavior in software refactoring: A systematic mapping study. *Information and Software Technology, 140*, Article 106675.

AlOmar, E. A., Mkaouer, M. W., Ouni, A., & Kessentini, M. (2019, September). On the impact of refactoring on the relationship between quality attributes and design metrics. In *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)* (pp. 1-11). IEEE.

AlOmar, E. A., Peruma, A., Mkaouer, M. W., Newman, C., Ouni, A., & Kessentini, M. (2021). How we refactor and how we document it? On the use of supervised machine learning algorithms to classify refactoring documentation. *Expert Systems with Applications, 167*, Article 114176.

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

Chaparro, O., Bavota, G., Marcus, A., & Di Penta, M. (2014, September). On the impact of refactoring operations on code quality metrics. In *2014 IEEE International Conference on Software Maintenance and Evolution* (pp. 456-460). IEEE.

Chávez, A., Ferreira, I., Fernandes, E., Cedrim, D., & Garcia, A. (2017). September). How does refactoring affect internal quality attributes? A multi-project study. In *In Proceedings of the XXXI Brazilian Symposium on Software Engineering* (pp. 74–83).

Fernandes, E., Chávez, A., Garcia, A., Ferreira, I., Cedrim, D., Sousa, L., & Oizumi, W. (2020). Refactoring effect on internal quality attributes: What haven't they told you yet? *Information and Software Technology, 126*, Article 106347.

Fowler, M. (2018). *Refactoring: Improving the design of existing code*. Addison-Wesley Professional.

Haque, Md. A., & Li, S. (2023, March 5). *The potential use of CHATGPT for debugging and Bug Fixing.* EAI Endorsed Transactions on AI and Robotics. https://publications.eai.eu/index.php/airo /article/view/3276#:~:text=ChatGPT%20can%20also%20be%20used,promising%20solution%20to%20this%20challenge.

Kaur, G., & Singh, B. (2017, June). Improving the quality of software by refactoring. In *2017 International Conference on Intelligent Computing and Control Systems (ICICCS)* (pp. 185-191). IEEE.

Kim, S., & Ernst, M. D. (2007, September). Which warnings should I fix first?. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering* (pp. 45-54).

Kim, M., Zimmermann, T., & Nagappan, N. (2012, November). A field study of refactoring challenges and benefits. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering* (pp. 1-11).

Liu, K., Kim, D., Bissyandé, T. F., Yoo, S., & Le Traon, Y. (2018). Mining fix patterns for findbugs violations. *IEEE Transactions on Software Engineering, 47*(1), 165–188.

Ma, W., Liu, S., Wang, W., Hu, Q., Liu, Y., Zhang, C., Nie, L., & Liu, Y. (2023, May 20). *The scope of chatgpt in software engineering: A thorough investigation.* arXiv.org.https://arxiv.org/abs/2305.12138.

Mens, T., & Tourwé, T. (2004). A survey of software refactoring. *IEEE Transactions on Software Engineering, 30*(2), 126–139.

Moser, R., Sillitti, A., Abrahamsson, P., & Succi, G. (2006, June). Does refactoring improve reusability?. In *International conference on software reuse* (pp. 287-297). Berlin, Heidelberg: Springer Berlin Heidelberg.

Murphy-Hill, E., Parnin, C., & Black, A. P. (2011). How we refactor, and how we know it. *IEEE Transactions on Software Engineering, 38*(1), 5–18.

Pantiuchina, J., Zampetti, F., Scalabrino, S., Piantadosi, V., Oliveto, R., Bavota, G., & Penta, M. D. (2020). Why developers refactor source code: A mining-based study. *ACM Transactions on Software Engineering and Methodology (TOSEM), 29*(4), 1–30.

Plosch, R., Gruber, H., Hentschel, A., Pomberger, G., & Schiffer, S. (2008, October). On the relation between external software quality and static code analysis. In *2008 32nd annual IEEE software engineering workshop* (pp. 169-174). IEEE.

Romano, S., Zampetti, F., Baldassarre, M. T., Di Penta, M., & Scanniello, G. (2022). Do static analysis tools affect software quality when using test-driven development?. In *In Proceedings of the 16th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement* (pp. 80–91).

Silva, D., Tsantalis, N., & Valente, M. T. (2016). Why we refactor? confessions of github contributors. In *In Proceedings of the 2016 24th acm sigsoft international symposium on foundations of software engineering* (pp. 858–870).

Stroggylos, K., & Spinellis, D. (2007, May). Refactoring–does it improve software quality?. In *Fifth International Workshop on Software Quality (WoSQ'07: ICSE Workshops 2007)* (pp. 10-10). IEEE.

Sun, W., Fang, C., You, Y., Miao, Y., Liu, Y., Li, Y., Deng, G., Huang, S., Chen, Y., Zhang, Q., Qian, H., Liu, Y., & Chen, Z. (2023, May 22). *Automatic code summarization via CHATGPT: How far are we?.* arXiv.org. https://arxiv.org/abs/2305.12 865.

Tian, H., Lu, W., Li, T. O., Tang, X., Cheung, S.-C., Klein, J., & Bissyandé, T. F. (2023, April 24). *Is CHATGPT the ultimate programming assistant – how far is it?.* arXiv.org. https://arxiv.org/abs/2304.11938.

Xia, C. S., & Zhang, L. (2023, April 1). *Keep the conversation going: Fixing 162 out of 337 bugs for $0.42 each using ChatGPT.* arXiv.org. https://arxiv.org/abs/2304.00385.