TECHNISCHE UNIVERSITÄT
CHEMNITZ

A Study of Expert java programmers on the Advantages and Challenges of Using ChatGPT for Code Refactoring

Master Thesis

Submitted in Fulfilment of the
Requirements for the Academic Degree
M.Sc.

Dept. of Computer Science
Course of Studies
Web Engineering

Submitted by: Ishrat Zahan Nishat
Registration number: [redacted]
Date: 24.06.2024
Supervisor:
Belinda Schantong
Professor: Dr. Janet Siegmund
Chemnitz University of Technology

Declaration of authorship

I hereby declare that this master's thesis is the result of my own independent work and research. All information and ideas derived from external sources, whether quoted directly or indirectly, have been duly acknowledged and cited. This thesis comprehensively includes references to all sources and materials consulted in the course of its preparation. I affirm that this thesis has not been submitted to any other examining body and has not been previously published in any form. This declaration confirms my adherence to the principles of academic integrity and the ethical conduct of research.

Chemnitz , 20.06.2024

place, date:

Ishrat Zahan Nishat

Signed: Ishrat Zahan Nishat

Acknowledgements

I am profoundly grateful for the support and guidance of several key individuals throughout my thesis journey. My supervisor, Belinda Schantong, deserves special thanks for her unwavering guidance, insightful feedback, and encouraging open dialogue. Her mentorship was invaluable.I also express my sincere gratitude to Dr. Janet Siegmund for providing me with this opportunity and for their constant support and guidance, which were instrumental in my research. A heartfelt thank you to the Java programmers who participated in my study; your contributions were vital to its success. Lastly, I must acknowledge my family's endless support and belief in me. Your encouragement has been my foundation and inspiration.This thesis is not only a reflection of my efforts but a testament to the unwavering support of those mentioned above. I am eternally grateful for their involvement in my journey.

Abstract

Background:
This study explores the interaction between expert Java developers and ChatGPT, focusing on the process of code refactoring—improving the internal structure of code without altering its external behavior.

Research Objective:
The main aim is to uncover the benefits and drawbacks of using ChatGPT for code refactoring tasks among professional Java programmers.

Research Method:
A combination of practical experiments and interviews was utilized. Programmers were tasked with refactoring code using ChatGPT, followed by discussions on their experiences to gather insights.

Results:
Results indicated that ChatGPT can significantly streamline the refactoring process, improving both the efficiency and quality of code. However, limitations were observed in ChatGPT's ability to fully understand complex, specific project requirements, which could lead to less than optimal refactoring suggestions.

Conclusion:
While ChatGPT offers valuable assistance in code refactoring, merging the precision of manual techniques with automated tool efficiency, it falls short in certain areas. These shortcomings highlight the need for further refinement in its understanding of project-specific details.

Future Work:

Future research should aim at enhancing ChatGPT's adaptability to specific project contexts and expanding its capability to assist with a broader range of refactoring tasks.

Keywords: Code Refactoring, Java Programmers, ChatGPT, Artificial Intelligence, Software Engineering

Contents

Contents 5

List of Figures 8

List of Tables 9

List of Abbreviations 10

1 Introduction 11
1.1 Background and Motivation 11
1.2 Problem Statements 12
1.3 Research Objectives 12
1.4 Research Questions 13
1.5 Thesis outline 14

2 Theoretical background 15
2.1 Code Refactoring 15
2.1.1 How it works? 16
2.1.2 Principles of Code Refactoring 16
2.1.3 Types of Code Refactoring 18
2.1.4 Refactoring example 19
2.1.5 Benefits of refactoring 21
2.2 Chat GPT 21
2.2.1 Overview of Chat GPT 21
2.2.2 Training Methodology 22
2.2.3 Advantages of Chat GPT 22
2.2.4 Disadvantages of Chat GPT 24

3 Related work 25
3.1 Existing Approaches to Code Refactoring 25
3.1.1 Manual Refactoring 25
3.1.2 Automated Refactoring Tools 25
3.1.3 AI-assisted Refactoring 25
3.1.4 Hybrid Approaches 26
3.1.5 Community and Collaborative Refactoring 26
3.2 Previous Studies on Code Refactoring with AI 26

4 Methodology **27**
4.1 Research Objective 27
4.1.1 Research Design Justification 27
4.2 Variables 27
4.2.1 Dependent Variable 28
4.2.2 Independent Variable 28
4.2.3 Hypotheses 28
4.3 Study Participants 29
4.4 Material and Tasks 30
4.4.1 Pre-Questionnaires 30
4.4.2 Programming Language and Experiment Language 31
4.4.3 Program understanding tasks 32
4.4.4 Code Snippets for Programming tasks 32
4.4.4.1 First Code snippet 32
4.4.4.2 Second Code snippet 34
4.4.4.3 Third Code snippet 35
4.4.4.4 Fourth Code snippet 36
4.4.4.5 Fifth Code snippet 37
4.4.5 Pretest 39
4.4.6 Post-test 39
4.4.7 Post Interview Questionnaires 40
4.5 Experimental Design 41

5 The experiment process **44**
5.1 Survey Pre-Questionnaires 45
5.2 Pretest 45
5.3 Posttest 45
5.4 Interview Questions 46

6 Results and Analysis **47**
6.1 Data collection and processing 47
6.1.1 Data collection 47
6.1.2 Data processing 47
6.2 Data analysis 49
6.2.1 Descriptive statistics 49
6.2.1.1 Description of the subjects 49
6.2.1.2 Response time of pretest code refactoring tasks . . . 53
6.2.1.3 Response time of post-test code refactoring tasks . . 55
6.2.1.4 Correctness of Pretest and Post-test code refactoring tasks 57
6.2.2 Post interview result 63
6.2.3 Hypothesis test 66
6.3 Data Analysis and Findings 67
6.3.1 Self-assessment of Java Expertise 68

6.3.2 Utilization of ChatGPT . . . 68
6.3.3 Quantitative Analysis of Response Times . . . 68
6.3.4 Qualitative Insights . . . 68
6.3.5 Conclusions Drawn from Data . . . 68

7 Discussion . . . 70
7.1 Finding and implication . . . 71
7.1.1 Threats to validity . . . 72
7.1.1.1 Internal validity . . . 72
7.1.1.2 External validity . . . 73
7.1.1.3 Experimental Control and Participant Integrity . . . 73

8 Conclusion and future work . . . 75
8.1 Conclusion . . . 75
8.2 Future work . . . 75
8.2.1 Proposed Enhancements . . . 76

Bibliography . . . 77
A.2 Access Instructions . . . 82

List of Figures

1.1 Expert programmers vs ChatGPT 12

2.1 Refactoring steps 17
2.2 Before Code refactoring 20
2.3 After Code refactoring 20
2.4 RLHF Training Method of ChatGPT 23

4.1 First Code Snippet 33
4.2 Second Code Snippet 34
4.3 Third Code Snippet 35
4.4 Fourth Code Snippet 37
4.5 Fifth Code Snippet 38
4.6 Experimental Design Steps 43

5.1 Survey Starting page 44
5.2 Survey Pre-Questionnaires page 45
5.3 Survey Task explanation page 46

6.1 Example of Collected front page dataset 48
6.2 Pretest Code Refactoring Response Times 55
6.3 Posttest Code Refactoring Response Times 57
6.4 Response sample one in pretest 58
6.5 Response sample one in posttest 58
6.6 Response sample two in pretest 59
6.7 Response sample two in posttest 59
6.8 Response sample three in pretest 60
6.9 Response sample three in posttest 60
6.10 Response sample four in pretest 61
6.11 Response sample four in posttest 61
6.12 Response sample five in pretest 62
6.13 Response sample five in posttest 62
6.14 Interview example quotes 64
6.15 Code Refactoring: Pretest and Posttest Response Times 69

List of Tables

2.1 Relevant Features per Method 19

4.1 Table of Pre-Questionnaires 31
4.2 Post Interview Questionnaires 41

6.1 How many years of experience do you have in Java Programming? . . 49
6.2 For how many years have you been programming for larger software projects e.g. in a company? Please enter a number between 0 and 30. 50
6.3 How many years of experience do you have with code refactoring? . . 51
6.4 Did you study programming or computer science at a university? . . 51
6.5 During your education, how many courses did you take where Java was the primary language? 51
6.6 How often have you used Chat GPT (e.g 1-low, 5-high)? 52
6.7 Have you used ChatGPT for code refactoring tasks (e.g 1-low, 5-high)? 53
6.8 Analysis of Participant Response Times in Code Refactoring Pretest Tasks 55
6.9 Analysis of Participant Response Times in Code Refactoring Posttest Tasks 56
6.10 Summary of Java experiment in pretest and posttest 69

List of Abbreviations

AOI Area of Interest

AI Artificial Intelligence

CSV Comma Separated Value

F1 Formula one

GPT Generative pre-trained transformer

HTML Hyper Text Markup Language

ID Identity Document

IDEA International Data Encryption Algorithm

LLM Large Language Model

OPAL Online-Plattform für Akademisches Lehren und Lernen

PDF Portable Document Format

PRO Proximal Policy Optimization

RLHF Reinforcement learning from human feedback

F1 Formula one

PDF Portable Document Format

SD Standard deviation

TUC Technical University of Chemnitz

1 Introduction

In the realm of software development, code refactoring stands as a cornerstone process, facilitating the enhancement of code aesthetics while preserving its functionality. By bolstering code extensibility, maintainability, and readability, refactoring plays a pivotal role in ensuring the long-term viability of software projects. Nevertheless, the manual execution of refactoring tasks often proves to be labor-intensive and prone to errors.

Driven by the burgeoning interest in harnessing AI-powered tools within software development, particularly for code refactoring, this research embarks on a journey to explore the potential of such technology. While existing literature recognizes AI's merits across various programming domains, there remains a notable gap in understanding its specific impact on expert Java programmers engaged in code refactoring. Delving into the experiences of these seasoned developers promises valuable insights into AI's capacity to support and enhance real-world software development endeavors.

This study sets out to illuminate the role of AI in the intricate process of code refactoring, with a particular focus on ChatGPT—a cutting-edge AI tool. By dissecting how ChatGPT can empower and assist expert Java programmers in their refactoring pursuits, this research aims to unearth practical implications that will enrich the collective understanding of AI's efficacy in refining code structures.

1.1 Background and Motivation

Restructuring current code to make it more aesthetically pleasing while maintaining its functionality is known as code refactoring.[1] It is an essential component of software development since it can enhance the extensibility, maintainability, and readability of the code. On the other hand, manual refactoring can be a laborious and error-prone process.

The motivation for conducting this research stems from the growing interest in using AI-powered tools like ChatGPT in software development, specifically for code refactoring tasks. While there is existing literature on the benefits of AI in various programming domains, there is a lack of specific studies that focus on expert Java programmers and their experiences with ChatGPT for code refactoring. Understanding the advantages and challenges faced by these skilled developers will provide valuable insights into the potential of AI assistance in real-world software development scenarios.

Ultimately, the findings from this study will contribute to a better understanding

of the role of AI in code refactoring and inform the software development community about the practical implications of using ChatGPT as a tool to support and empower expert Java programmers in their code refactoring endeavors.

Figure 1.1: Expert programmers vs ChatGPT
[5]

1.2 Problem Statements

The potential of large language models (LLMs) such as ChatGPT to help software engineers with different jobs is growing as these models get more and more sophisticated. Code refactoring is one such process that entails enhancing the readability and structure of code without affecting its usefulness. It's still uncertain how useful ChatGPT is for code refactoring , especially for Java programmers, in terms of its restrictions and usefulness. In Figure 1.1, we can see an imaginary sample scenario illustrating the problem.

1.3 Research Objectives

The primary objective of this research is to assess how ChatGPT aids expert Java programmers in refining their code refactoring techniques. By analyzing code snippets, we will examine how developers employ ChatGPT's suggestions and guidance to optimize their refactoring process. Through interviews, we will gain in-depth insights into the specific ways ChatGPT contributes to their improvement in refac-

toring practices. A significant aspect of this study is to quantify the impact of ChatGPT on the participants' productivity during code refactoring tasks.

By comparing the time taken to complete refactoring tasks with and without ChatGPT, we will determine if the tool accelerates the refactoring process. The findings will provide valuable information on how ChatGPT influences the efficiency of expert Java programmers, enabling them to produce higher-quality code in a shorter time frame. Alongside the advantages, this research aims to uncover the challenges expert Java programmers encounter when integrating ChatGPT into their code refactoring workflow.

Through interviews, participants will have the opportunity to express their concerns, limitations, and areas where ChatGPT may fall short of meeting their expectations. Understanding these challenges will facilitate the development of strategies to address them and optimize the use of ChatGPT in code refactoring processes. By conducting this study, we aim to shed light on how ChatGPT can positively impact the code refactoring process for expert Java programmers.

We want to explore the extent to which ChatGPT can enhance their productivity, enable them to discover new refactoring techniques, and improve the maintainability of their code. Additionally, the research aims to identify any potential limitations or challenges faced by these programmers when integrating AI assistance into their workflow.

1.4 Research Questions

To study this approach the following research questions should be answered:

1. **Research Question** - How do java expert programmers use ChatGPT to refactor code?

2. **Research Question** - How do java expert programmers describe ChatGPT's impact on their productivity and code maintainability?

3. **Research Question** - What perceived benefits and challenges do expert Java programmers encounter when using ChatGPT for code refactoring tasks?

1.5 Thesis outline

Embarking on this academic quest, here's the roadmap: my thesis outline.
Chapter 1 (Introduction):
Introduces the thesis work, highlighting the objectives, research questions, significance, limitations, and the overall structure of the thesis.

Chapter 2 (Literature Review):
Provides an overview of previous studies relevant to the use of AI in code refactoring, including discussions on program comprehension, programming structures, and methods for testing comprehension, with a focus on Java programming.

Chapter 3 (Related Work):
Explores existing tools and approaches related to AI-driven code refactoring, examining their strengths, weaknesses, and applicability to the context of expert Java programmers.

Chapter 4 (Methodology):
Details the research design, including the approach to integrating ChatGPT into Java development environments, the philosophy guiding the research, methods for data collection and analysis, and any limitations inherent in the chosen methodology.

Chapter 5 (Experiment Process):
Describes the process of conducting experiments with ChatGPT in the context of code refactoring tasks performed by expert Java programmers. This includes the setup, execution, and any adjustments made during the experimental phase.

Chapter 6 (Results):
Presents the findings of the experiments, detailing how ChatGPT performed in assisting with code refactoring tasks, including metrics such as review time, detection of code smells, and effectiveness in suggesting refactorings.

Chapter 7 (Discussion):
Analyzes the results, discussing the implications of ChatGPT's performance for expert Java programmers, highlighting strengths, weaknesses, and areas for improvement.

Chapter 8 (Conclusion):
Offers a conclusion based on the findings, summarizing the effectiveness of ChatGPT as a tool for supporting code refactoring in Java development. Additionally, discusses potential future research directions and applications of AI in software engineering.
This shortened outline encapsulates the essence of each chapter, focusing on the key points and objectives of my thesis.

2 Theoretical background

The Theoretical background of this thesis explores the concept of code refactoring within software engineering—a practice aimed at enhancing the internal structure of code without altering its external functionality. This section reviews the evolution of code refactoring, highlighting its importance in maintaining high-quality, readable, and maintainable software. It delves into various methodologies, from manual to automated refactoring, and discusses the impact of these practices on developer productivity and software project lifecycles. Additionally, it introduces the emerging trend of AI-assisted refactoring, examining its potential to revolutionize traditional refactoring processes by automating complex tasks and improving decision-making. This comprehensive overview sets the stage for understanding the role of code refactoring in contemporary software development and its implications for future research.

2.1 Code Refactoring

Code refactoring is a disciplined method aimed at improving the internal structure of existing code while preserving its external behavior. The essence of code refactoring lies in its capacity to make software more maintainable, extensible, and readable without introducing new functionality. This process is crucial for managing software complexity, enhancing code quality, and facilitating future development efforts.

The core principles guiding code refactoring include improving the readability and comprehensibility of code, reducing complexity, eliminating redundant or duplicate code, and, in some instances, optimizing performance. Techniques such as "Extract Method," "Rename Variable/Method," "Move Method/Class," and "Inline Method" are commonly employed to address specific issues within the codebase, ranging from simplifying complex code segments to enhancing code modularity and clarity.

It's impossible to say enough about how important code refactoring is in software development. It is a key part of keeping the program clean, lowering technical debt, and making sure that software can adapt to new needs and changes in the future. By following the rules and methods of code refactoring, developers can make sure that their code stays strong, efficient, and simple to manage. This helps software projects succeed and last for a long time.

2.1.1 How it works?

Fowler et al., [2] said that, Refactoring is basically the object-oriented variant of restructuring: “the process of changing a software system in such a way that it does not alter the external behavior of the code, yet improves its internal structure”. Fowler also talked about bad smells, which are signs that there is a problem in the code that needs to be fixed by refactoring. There are a lot of tools that can help you find the bad smells and get rid of them by using refactoring tools and techniques.

There are seven descriptions of the steps involved in refactoring.[5]
1. Execute the program to verify that, upon refactoring, its external behavior has not changed.
2. Determine the program’s complexity before doing any refactoring.
3. Determine which areas of the code need to be refactored. Choose the refactoring to apply to the highlighted areas.
4. Make a small change, such as a single refactoring, without affecting the code’s external behavior.
5. Test the refactored code; if it functions as intended, proceed to the next refactoring.
6. If unsuccessful, undo the most recent, small change and carry out the refactoring in a different way.
7. After using every refactoring strategy, compute the complexity to ascertain how refactoring affects quality. [5]

2.1.2 Principles of Code Refactoring

Refactoring preserves the functionality of the code while making it clearer, easier to comprehend, and easier to maintain. There are five principles describe by Tomasz [6]. For Examples:
1.Hide ”how” with ”what”: This concept says to focus on how something works instead of how it’s implemented. Instead of getting bogged down in specific implementation details, give variables, functions, and classes names that are clear and tell you what they do. Imagine code that reads like a story, with each function and variable making its purpose clear without you having to think about the specifics of how it was implemented.
2.Aim for consistency: This concept encourages our codebase to be consistent. Maintain consistent naming practices, coding style, and formatting. Think of it as creating a common language within your code. Formatting that is always the same makes code easier to read, makes it easier for developers to navigate, and makes upkeep easier because everyone knows the standard patterns.
3.Avoid deep nesting: This principle aims to make control structures less complicated. Having a lot of nested loops or conditional lines in our code can make it hard to understand and fix. Instead, break up complicated logic into functions that are easier to handle. Picture a clear structure of functions, each one handling

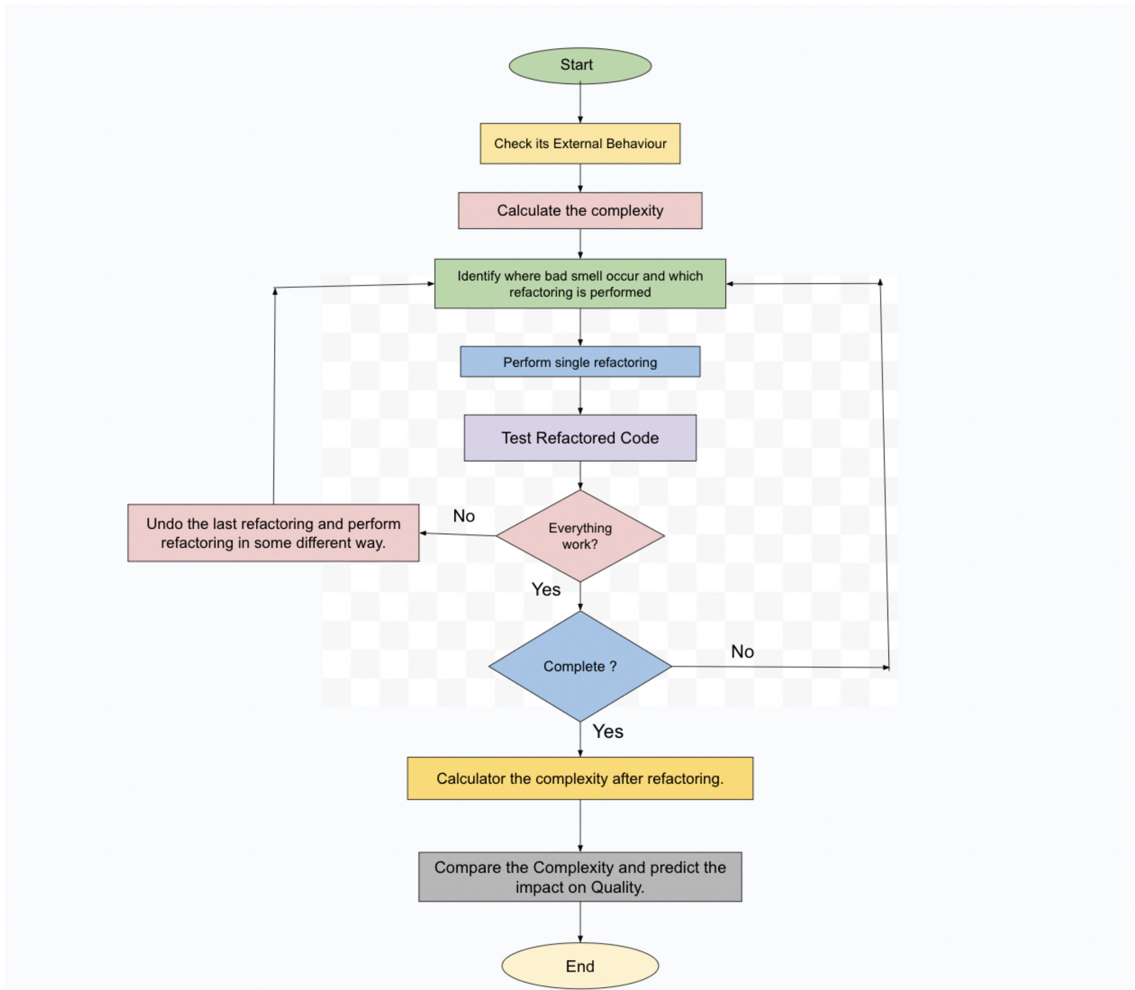

Figure 2.1: Refactoring steps
[5]

a different subtask, instead of a single, confusing block of nested logic.
4.Separate concern (Single Responsibility Principle): This principle stresses the use of modules. There should be a clear goal for every method, class, and module. Putting different unrelated jobs on the same class makes it harder to use and increases the chance that changes will affect functions that are not related. We can think of functions and classes as different experts, each with their own special skill, working on the same project.
5.Avoid duplication wisely (Don't Repeat Yourself): This concept encourages code reuse. If we see the same thinking used more than once, don't just copy and paste it. We should instead make a function or class that holds that reasoning. This cuts down on duplicate code, makes upkeep easier (since changes made in one place affect all uses), and organizes the code better overall. Imagine having a central library of reusable parts instead of versions that are spread out and might not work together.
Good methods for refactoring code are based on these five rules. If we follow them, you can write code that works well, is clean, easy to read, easy to manage, and can be changed in the future. Remember that these principles are not hard and fast rules. Instead, they are suggestions that we can use to make your project and team work better. [6]

2.1.3 Types of Code Refactoring

In the book, Fowler [6] introduces several refactoring techniques, and the list includes over 70 refactoring. We are going to describe six important techniques.For example:
1.Extract Method:
Purpose: Reduces a long or complex technique to smaller, more manageable steps, enhancing readability and maintainability.
Process: Identify a code portion within a method that performs a certain task. Create a new method with a suitable name to encapsulate this code. Replace the existing code with a call to the new method.
2.Inline Method:
Purpose: Reduces code complexity by deleting methods that are tiny, used only once, and do not provide considerable abstraction.
Process: Identify a method that fits the criteria for inlining. Replace any calls to the method with its real content. Remove the method declaration.
3.Move Method:
Purpose: Moves a method to a more appropriate class, increasing cohesion and decreasing coupling.
Process: Identify a method that primarily interacts with data and methods from another class. Relocate the method to the appropriate class and update any references to it.
4.Pull-Up Method:
Purpose: Removes code duplication by relocating a method used by numerous sub-

classes to a single superclass.
Process: Identify a method that is identical across numerous subclasses. Move the method to the superclass, ensuring that it can still access the required data.
5.Push-Down Method:
Purpose: Encourages encapsulation and minimizes needless dependencies by relocating a method used only by one subclass to that subclass.
Process: Identify a method in a superclass that is exclusively utilized by specific subclasses.Move the method to the appropriate subclass and update any references.
6.Rename method:
Purpose: To improve code clarity and communication, give a method a more descriptive or correct name.
Process: Find a method with an incorrectly named identifier. Change its name to one that more accurately describes its purpose or capabilities. Use these strategies wisely to improve code quality, maintainability, and adaptability.

Extract Method	Inline Method	Move Method	Pull-Up Method	Push-Down Method	Rename Method
Add	Combine	Move	Move	Move	Change
Create	Gather	Add	Pull	Push	Fix
Extract	Inline		Shift	Reduce	Improve
Move	Merge			Remove	Rename
Separate	Move				Update
Split					
Break Up					

Table 2.1: Relevant Features per Method
[8]

2.1.4 Refactoring example

By simplifying the conditional phrases, the code becomes more compact while remaining clear and readable.Here is a simple example of java code refactoring.

In Figure 2.2, The code is functionally accurate, however it might be reduced to improve readability and conciseness. Here is an improved version of the code in figure 2.3.

The technique of this code refactoring are:
Simplifying conditional expressions: The if-else loop inside the **isEven** method is reduced to a single return statement by utilizing the condition **num% 2 == 0**. This avoids the requirement for explicit if and else blocks when the result is a boolean.

```
public class OddEvenOperations {

    public static boolean isEven(int num) {
        if (num % 2 == 0) {
            return true;
        } else {
            return false;
        }
    }

    public static void main(String[] args) {
        int number = 20;

        if (isEven(number)) {
            System.out.println(number + " is even.");
        } else {
            System.out.println(number + " is odd.");
        }
    }
}
```

Figure 2.2: Before Code refactoring

```
public class OddEvenOperations {

    public static boolean isEven(int num) {
        return num % 2 == 0;
    }

    public static void main(String[] args) {
        int number = 20;

        System.out.println(number + (isEven(number) ? " is even." : " is
odd."));
    }
}
```

Figure 2.3: After Code refactoring

Using Ternary Operators in the Main Method: In the main method, the conditional check is replaced by a ternary operator, which is a more compact approach to express a conditional statement. This results in a more concise and streamlined representation of the logic.

2.1.5 Benefits of refactoring

Refactoring has various benefits in software development, including increased code quality, maintainability, and developer efficiency. Refactoring has several significant benefits, including improved code readability and understandability, the elimination of code duplication, the elimination of code smells, the improvement of maintainability, bug fixing and troubleshooting, and more. Kim et al.[9] surveyed 328 professional software engineers at Microsoft, conducted interviews with six members of a team responsible for refactoring Windows 7, and performed a quantitative analysis of the Windows 7 version history in one of the most comprehensive refactoring field studies to date. This survey reported that restructuring improves readability (43%), maintainability (30%), extensibility (27%), and reduces defects (27%). Poor readability was the most common cause for refactoring (22%) of respondents). Only one 'official ' code smell was mentioned: code duplication (13%).[9]

2.2 Chat GPT

ChatGPT represents a significant advancement in artificial intelligence, integrating technologies like deep learning and reinforcement learning across versions from GPT-1 to GPT-4. It has evolved to handle a wide range of tasks with minimal fine-tuning, demonstrating remarkable generalization capabilities, especially in GPT-3, which features a parameter scale 100 times larger than its predecessor. The InstructGPT version, built on GPT-3.5, aims for better user interaction through reinforcement learning with human feedback. In the realm of programming, ChatGPT offers substantial benefits, such as enhancing computational thinking and providing debugging assistance, thereby improving learning outcomes in programming education.

2.2.1 Overview of Chat GPT

With ChatGPT, artificial intelligence has taken a giant leap forward.ChatGPT is a comprehensive system that combines several advanced technologies like deep learning, unsupervised learning, instruction fine-tuning, multi-task learning, in-context learning, and reinforcement learning. The model is based on the original GPT (Generative pre-trained Transformer) model, which has undergone iterative updates from GPT-1 to GPT-4.[11]

GPT-1 [16], which was created in 2018, is first used to train an unsupervised learning model for a Transformer-based generative language model [16, 17, 18]. The model

is then fine-tuned on tasks that come after it has been trained. The 2019 release of GPT-2 [19] mostly introduces the idea of multi-task learning [23]. It trains with more network parameters and data than GPT, which means that the pre-trained generative language model can be used for most of the supervised subtasks without needing any more tweaking. GPT-3 [22] combines meta-learning [23, 24] with in-context learning [28] to make the model even better at few-shot or zero-shot [21]settings. This makes the model much better at generalization, beating most existing methods on a wide range of downstream tasks. Also, GPT-3's parameter scale is 100 times bigger than GPT-2's, and it is the first language model to have a parameter scale greater than 100 billion. InstructGPT is a pilot version of ChatGPT. It is a derivative version of the GPT3.5 series models.

The researchers use reinforcement learning with human feedback (RLHF) to train the GPT-3 model[26] over time so that it can better understand what the user is trying to say. Finally, ChatGPT performs at a human-level on a number of professional and academic benchmarks when it comes to GPT-4, a large multimodal model that can take both picture and text inputs and send text outputs.

2.2.2 Training Methodology

The training operation of ChatGPT consists of multiple stages. Initially, the user inputs a query or commands a program. The response generated by the model is predicated on its comprehension of the linguistic connections and patterns evident in the given prompt. The user is subsequently provided with an additional opportunity to provide a response or present an additional inquiry. This methodology exclusively employs reinforcement learning in conjunction with human feedback showing in figure 2.4.[15] They consist of:
a) The SFT Model is trained using demonstration data collected.
b) The RM Model provides points based on the appeal of the SFT Model's output to consumers and
c)The SFT model using PPO is adjusted by using reinforcement learning to maximize the RM. PPO stands for the refined version of the proximal policy optimization model.[15]

2.2.3 Advantages of Chat GPT

Although Chat GPT is used in a wide range of industries, we will focus on the programming industry since this is where its effects are most evident. Research indicates that generative AI tools like ChatGPT improve students' computational thinking, programming self-efficacy, and motivation.[27] According to Chen et al.,[28] students' programming abilities can be effectively improved through the tools like ChatGPT, which offers advantages including debugging and code explanations.
Here are some benefits that ChatGPT can offer to individuals looking to learn programming. ChatGPT can communicate with individuals through natural language

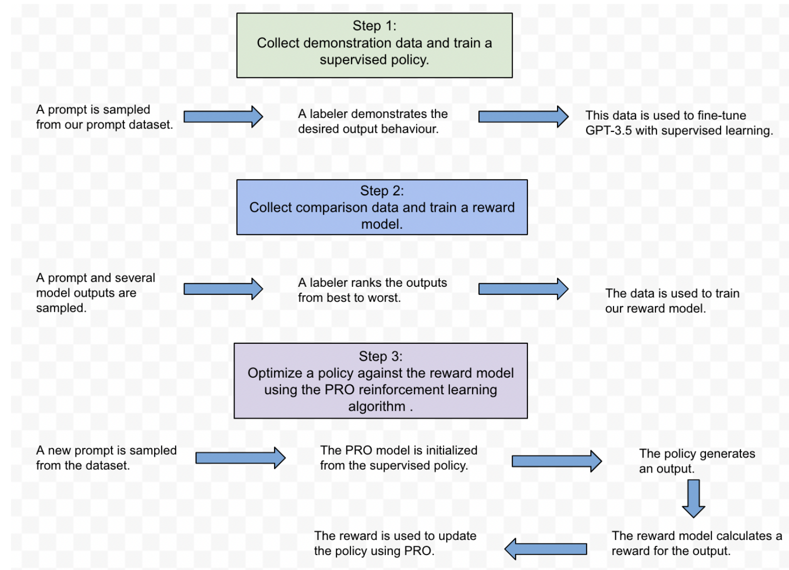

Figure 2.4: RLHF Training Method of ChatGPT
[15]

processing technologies. This allows users with little programming experience to solve programming challenges using ChatGPT. [29]
Easy Access: ChatGPT is accessible from any device with an internet connection. Users are not required to install any specialized software or tools. ChatGPT gives consumers with rapid input with its quick response feature. This can accelerate the learning process and improve pupils' comprehension.
Personalized Learning: ChatGPT can offer consumers a customized learning experience. It can give users learning materials, practices, or examples, monitor the learning process, and deliver personalized feedback.
Supports multiple languages: This enables anyone who want to learn programming to communicate with ChatGPT in their own language.
Limitless Resources: By leveraging the huge resources of the internet, ChatGPT can supply users with a limitless amount of programming resources. These resources contain study materials, code samples, online courses, apps, and more.
Clear Explanations: ChatGPT offers students clear explanations of programming topics. It improves students' understanding of the topics while also allowing them to spend less time.
Examples and Applications: ChatGPT provides students with programming examples and applications, allowing them to put theoretical knowledge into practice. This improves students' understanding of the material and their overall learning experience.
Inquiry and Search: ChatGPT allows students to ask questions and look up programming subjects. This allows students to conduct research and learn more about areas of interest to them.

Debugging and Feedback: ChatGPT helps students detect and correct programming problems. It also provides feedback to students and can make recommendations for better programming practices.
Advanced topic: ChatGPT assists students in progressing to more higher levels of programming. Students can progress to more sophisticated levels of programming.[29]

2.2.4 Disadvantages of Chat GPT

According to Rahman et al.,[30] while ChatGPT provides some advantages in programming learning, it also has disadvantages such as a lack of common sense, potential biases, difficulties with complicated thinking, and inability to process visual information. Here are some of the limitations.[29]
Lack Understanding of Context: There is no history of past conversations in ChatGPT.Based on the current context given in the conversation, it creates responses.This can make it difficult to have a meaningful and relevant discourse, particularly when talking about intricate programming subjects that require several messages.
Limited Code Execution: ChatGPT lacks the capacity to run or test code; it can only comprehend and produce code snippets.While syntactically correct code may be produced, functional accuracy and efficiency may not always be guaranteed.
Dependency on training Data: The quality of the model depends on the quality of the training data.It might not be as adept at giving precise information or help in certain areas if it wasn't trained with a particular programming language, library, or framework.
Ambiguity Difficulties: It may be difficult for ChatGPT to handle unclear questions or requests for explanation.Incomplete code snippets or ambiguous programming questions could result in answers that are neither precise nor beneficial.
Unable to Manage unseen Concepts: It's possible that the model is out of date with respect to the newest libraries, technologies, or programming languages released after the previous training cutoff.It might not be able to offer guidance or information on these more recent ideas.
Security Issues: Because ChatGPT can generate responses based on the input it gets, if it isn't carefully checked by a human, it may unintentionally deliver insecure code or recommend behaviors that could result in security issues.
Not a Replacement for Human Expertise: Although ChatGPT can be a useful tool for quickly obtaining information or advice, it shouldn't be used in place of the knowledge of seasoned programmers or other experts.For the solutions to be accurate and secure, human oversight is necessary.
Excessively Wordy Reactions: The model may produce lengthy or too complex responses, particularly in situations where a short code snippet or explanation would be sufficient. Because of this, it could be difficult to glean the most pertinent details from the model's answers.[29]

3 Related work

This chapter examines the landscape of research and practice in the domain of code refactoring, providing a critical review of existing methodologies, tools, and approaches. It situates the current study within the broader academic and practical discourse on enhancing software quality and developer efficiency through code refactoring.

3.1 Existing Approaches to Code Refactoring

Code refactoring is a cornerstone of software maintenance and evolution, aiming to improve the internal structure of software without altering its external behavior. This section delves into various approaches to code refactoring, ranging from manual techniques to automated solutions, highlighting the contributions and limitations of each.

3.1.1 Manual Refactoring

Manual refactoring, as described by Fowler in his seminal work *Refactoring: Improving the Design of Existing Code* [33], remains the foundational approach. It relies on the developer's insight to identify 'code smells' and apply appropriate refactoring patterns. While offering precision and control [43] ,this approach demands a high level of expertise and can be time-consuming and error-prone [43].

3.1.2 Automated Refactoring Tools

The advent of automated refactoring tools has significantly impacted the practice of code refactoring. Tools such as JRefactory and ReSharper leverage static analysis to identify refactoring opportunities and execute transformations [35]. While these tools increase efficiency and reduce human error, they may not always grasp the nuanced context of the code, leading to suboptimal refactoring suggestions [40].

3.1.3 AI-assisted Refactoring

Recent advancements in artificial intelligence have paved the way for AI-assisted refactoring, promising to combine the precision of manual refactoring with the efficiency of automated tools. Techniques leveraging machine learning algorithms

can predict refactoring opportunities by learning from large codebases [36]. For instance, DeepCode uses deep learning to analyze code and suggest improvements [37]. However, the effectiveness of these approaches heavily depends on the quality and diversity of the training data, and there remains a gap in understanding complex, project-specific requirements .

3.1.4 Hybrid Approaches

Recognizing the strengths and weaknesses of both manual and automated methods, hybrid approaches have emerged. These approaches integrate human expertise with the computational power of tools to guide the refactoring process. Studies such as the one conducted by Murphy-Hill et al. [39] illustrate how developers can effectively collaborate with tools to achieve optimal refactoring outcomes, ensuring both efficiency and context-awareness.

3.1.5 Community and Collaborative Refactoring

The role of the developer community and collaborative platforms in refactoring cannot be overlooked. Open-source projects and coding platforms like GitHub provide fertile ground for collaborative refactoring practices, where peer reviews and contributions enhance code quality across projects [42]. This collective intelligence approach leverages the diverse expertise within the community to identify and implement best refactoring practices.

3.2 Previous Studies on Code Refactoring with AI

Almeida, Yonatha, et al.[31] conducted a thorough investigation into the efficacy of integrating artificial intelligence, specifically leveraging a plugin designed for IntelliJ IDEA and powered by GPT-3.5, aimed at enhancing code review processes by autonomously identifying both syntactical and semantic issues within code segments and providing actionable solutions. Throughout their study, they meticulously assessed the AICodeReview tool's performance, employing various evaluation metrics such as precision, recall, F1 score, accuracy, and BLEU scores. Their research, which engaged 12 undergraduate students possessing fundamental programming knowledge, spanned analyses on review time, the detection rate of code smells, and the successful refactoring of identified issues.

Their findings underscored the tool's remarkable impact on code review efficiency, as evidenced by significant reductions in review durations and heightened accuracy in detecting code smells. Moreover, the AICodeReview tool demonstrated adeptness in supporting the refactoring of identified code issues, thereby contributing to the overall enhancement of software quality.[31]

4 Methodology

This chapter outlines the research design employed to investigate the advantages and challenges of using ChatGPT for code refactoring among expert Java programmers.

4.1 Research Objective

Variables can help frame the research design, data collection, and analysis processes in order to systematically evaluate the benefits and drawbacks of using ChatGPT for code refactoring among expert Java programmers. This study adopts a mixed-methods approach, combining quantitative and qualitative data collection methods to gain a comprehensive understanding of the research question.

Quantitative data has been obtained through measuring variables related to code quality and refactoring time.

Qualitative data has been gathered through interviews to capture the participants' experiences, insights, and perspectives on using ChatGPT for code refactoring.

4.1.1 Research Design Justification

A mixed-methods approach is chosen for this research due to the following reasons:

1. **Comprehensiveness:** Quantitative data provides objective measures of code quality and refactoring efficiency, while qualitative data offers valuable insights into the participants' experiences, perceptions, and thought processes. Combining both methodologies allows for a more holistic understanding of the research question.

2. **Triangulation:** By using different data collection methods, potential biases associated with individual methods can be mitigated. Convergence of findings across quantitative and qualitative data strengthens the overall research conclusions.

4.2 Variables

This section has delineated the crucial dependent and independent variables central to this thesis, laying the groundwork for investigating the role of ChatGPT in code refactoring. By scrutinizing these variables, the research aims to furnish valuable

insights into the potential benefits, challenges, and overall impact of integrating AI tools into software development practices.

4.2.1 Dependent Variable

In a study, the dependent variable represents the outcome or reaction measured or observed as a result of manipulating the independent variable(s). These variables are the aspects that researchers anticipate will change due to modifications in the independent variable(s). For this study, the dependent variables could encompass:

1. Response Time: Time taken by participants to complete the refactoring tasks.
2. Improvement in code readability/maintainability: Evaluated through expert review and code review tools.
3. Advantages of Using ChatGPT for Code Refactoring: Gathered through post-interview questionnaires and interviews.
4. Challenges of Using ChatGPT for Code Refactoring: Collected through post-interview questionnaires and interviews.

4.2.2 Independent Variable

Independent variables are intentionally altered, and their effects on dependent variables are examined and evaluated. These variables are manipulated at specific specified levels, often termed as treatments. For this study, the independent variable revolves around the utilization of ChatGPT.

4.2.3 Hypotheses

In this subsection, we delve into the hypotheses pertaining to the impact of ChatGPT on code refactoring tasks. These hypotheses aim to elucidate the potential effects of ChatGPT utilization on the refinement and enhancement of code structures.

H_1: Response time in code refactoring tasks is influenced by the extent of ChatGPT usage, with higher levels of usage leading to either faster or slower response times compared to lower levels.

H_2: Expert Java programmers perceive several advantages associated with using ChatGPT for code refactoring, such as increased efficiency, accuracy, and creativity in code improvements.

H_3: There is a positive correlation between the use of ChatGPT and the improvement in code readability and maintainability, with higher levels of ChatGPT utilization resulting in more substantial enhancements.

H_4: Despite its benefits, expert Java programmers encounter various challenges when using ChatGPT for code refactoring, including limitations in understanding

context-specific programming nuances, potential errors in code modifications, and integration complexities.

4.3 Study Participants

This study aimed to examine the effects of utilizing ChatGPT for code refactoring tasks among expert Java programmers. A meticulous selection process was employed to identify participants who met the following criteria:

1.Experience: Proficiency in Java programming was paramount, with a stipulated minimum of X years of experience. This benchmark was established based on industry standards to ensure participants possessed a depth of knowledge and practical expertise in Java development.

2.Refactoring Experience: Candidates were required to have substantive experience in code refactoring, ensuring they could proficiently engage with the complexities of the tasks and provide informed feedback on ChatGPT's utility in this context.

3.Java Syntax and Concepts Familiarity: A thorough understanding of Java syntax and fundamental programming concepts was essential, ensuring participants could accurately assess the quality of refactoring suggestions provided by ChatGPT.

The recruitment strategy was diverse, leveraging online communities, professional networks, and academic institutions with robust Java programming curricula. This approach ensured a wide pool of potential participants, from which 30 individuals initially partook in the study. The survey link was distributed across different professional platforms, targeting a broad spectrum of Java developers.

Of the 40 participants who commenced the survey, 14 successfully completed all refactoring tasks. This completion rate underscores the challenging nature of the tasks and possibly the varying degrees of familiarity and comfort with using AI tools like ChatGPT for such specific programming activities. The participants were sourced from a variety of professional backgrounds, encompassing different levels of experience and expertise in Java programming. This diversity was intentional, aimed at capturing a wide range of insights and experiences with ChatGPT's application in code refactoring tasks.

The survey and tasks were designed not only to assess the direct impact of ChatGPT on code quality and refactoring efficiency but also to gather qualitative feedback on the participants' experiences. This feedback covered the perceived advantages and challenges of integrating ChatGPT into their refactoring workflow, providing a holistic view of its potential as a tool in software development practices.

By engaging expert Java programmers from varied professional environments and with a minimum of X years of experience, the study meticulously evaluates ChatGPT's utility and effectiveness in the code refactoring process. The insights derived from this select group contribute significantly to understanding the nuanced dynamics of AI-assisted programming, specifically within the realm of Java development.

4.4 Material and Tasks

The experimental data in our study consisted of snippets of code. Code snippets are a crucial component of studying how individuals comprehend programs, and they frequently dictate the success of an experiment. Selecting code snippets is a crucial aspect of hypothesis evaluation, as the research aims to determine the variations in programmers' behavior regarding accuracy and response time when attempting to comprehend complex code snippets. For our study, we administered 5 distinct refactoring tasks for both the pretest and posttest, along with 8 interview questions.

4.4.1 Pre-Questionnaires

We gave our participants a number of questionnaires that were designed to find out about their backgrounds and areas of skill in order to get complete demographic and programming experience data. Our group of participants came from a wide range of professional Java programming fields, so we were sure to get a wide range of views. To get a sense of how much experience our subjects had, we asked them how long they had been programming in Java. In order to get a full picture of their actual programming work, we also asked them about their involvement in bigger software projects, like those done in professional settings or by companies.To get a better idea of how much they knew about code rewriting, we also asked them how many years of experience they had in this area. We also wanted to know about their schooling, so we asked them if they studied computer science or programming at the university level.To get a better idea of how skilled the participants were, they were asked to rate how much Java-based training they had taken in school. In addition, we used a scale to compare how skilled they thought they were at Java programming to that of people with more than 20 years of experience as well as their friends or colleagues.We looked at how often they used ChatGPT and whether they only used it for code refactoring jobs when we looked into how they used it. In this way, we were able to figure out how well ChatGPT fit into their developing processes.Lastly, to put the participants' work projects in context, they were asked to say how big their average Java projects were, with projects being small, medium, or large based on the number of lines of code. This gave them useful information about the size and difficulty of their computing projects.

Question Number	Questions
Question 1	How old are you?
Question 2	How many years of experience do you have with Java programming?
Question 3	For how many years have you been programming for larger software projects e.g. in a company? Please enter a number between 0 and 30.
Question 4	How many years of experience do you have with code refactoring?
Question 5	Did you study programming or computer science at a university?
Question 6	During your education, how many courses did you take where Java was the primary language?
Question 7	On a scale from 1 to 5, how would you rate your Java programming expertise (e.g 1-very inexperienced, 5-very experienced)?
Question 8	How would you compare your Java expertise to those with over 20 years of practical experience (e.g 1-very inexperienced, 5-very experienced)?
Question 9	How would you rate your Java expertise in comparison to your peers or colleagues (e.g 1-very inexperienced, 5-very experienced)?
Question 10	How often have you used Chat GPT (e.g 1-low, 5-high)?
Question 11	Have you used ChatGPT for code refactoring tasks (e.g 1-low, 5-high)?
Question 12	What is the average size of Java professional projects you typically work on, categorized as small-scale (up to 900 lines of code), medium-scale (900 to 40,000 lines of code), or large-scale (exceeding 40,000 lines of code)?

Table 4.1: Table of Pre-Questionnaires

4.4.2 Programming Language and Experiment Language

Java was chosen for researching the advantages and challenges of using ChatGPT for code refactoring due to its widespread popularity, complexity of projects, mature tooling ecosystem, and active developer community. Its extensive adoption and diverse challenges in code maintenance make it an ideal target for evaluating the effectiveness of AI assistance in real-world development workflows. Additionally, Java's mature ecosystem of development tools facilitates seamless integration of AI-powered tools like ChatGPT, enabling efficient experimentation and evaluation. Overall, focusing on Java allows researchers to gain valuable insights into the practical implications of employing AI in code refactoring within a widely used programming ecosystem.

4.4.3 Program understanding tasks

For this research on understanding the advantages and challenges of using ChatGPT for code refactoring, we selected five code snippets tailored for professional Java programmers. These snippets were carefully chosen to represent common scenarios encountered in real-world Java development, covering various aspects of syntax, structure, and logic. By incorporating a diverse range of code examples, we aim to assess ChatGPT's effectiveness in identifying and suggesting improvements for different types of code issues.

Each code snippet is designed to challenge ChatGPT with tasks such as optimizing performance, improving readability, adhering to coding conventions, handling exceptions, and enhancing overall maintainability. These tasks align closely with the day-to-day responsibilities of professional Java developers, ensuring the relevance and practicality of our evaluation.

Through the analysis of these carefully curated code snippets, we intend to gain valuable insights into how ChatGPT performs in addressing the specific needs and requirements of Java developers during code refactoring. This approach enables us to evaluate the tool's capability to assist professionals in efficiently improving code quality while navigating the complexities of real-world Java projects.

4.4.4 Code Snippets for Programming tasks

These snippets were carefully selected to demonstrate best practices in Java programming, emphasizing code efficiency, readability, and maintainability. Each snippet is accompanied by a detailed explanation of its functionality and the specific problem it solves, providing a clear link between programming theory and practice. This approach not only showcases the versatility of Java as a programming language but also enhances the understanding of complex algorithms and data structures through concrete examples.

4.4.4.1 First Code snippet

This code in figure 4.1 is a method called **getPayAmount()** which calculates the pay amount based on certain conditions. Here's a simple breakdown:

It starts by declaring a variable result of type double. It checks if a person is dead. If they are, it calculates the pay amount using the **deadAmount()** method. If the person is not dead, it checks if they are separated. If they are, it calculates the pay amount using the **separatedAmount()** method. If the person is neither dead nor separated, it checks if they are retired. If they are, it calculates the pay amount using the **retiredAmount()** method. If the person is none of the above (not dead, not separated, and not retired), it calculates the pay amount using the **normalPayAmount()** method. Finally, it returns the calculated result.

There are several ways the given code can be refactored to improve readability and maintainability. Here are a few possibilities:

```
public double getPayAmount() {
  double result;
  if (isDead) {
    result = deadAmount();
  } else {
    if (isSeparated) {
      result = separatedAmount();
    } else {
      if (isRetired) {
        result = retiredAmount();
      } else {
        result = normalPayAmount();
      }
    }
  }
  return result;
}
```

Figure 4.1: First Code Snippet

1.Using Early Returns:
Instead of nesting multiple if-else statements, we can use early returns to simplify the code and improve readability.
2.Using Switch Statements:
If the conditions are based on simple checks of variables, we can use switch statements for better readability.
3.Using Ternary Operator: If the conditions are straightforward, we can use the ternary operator to make the code more concise.
4.Extracting Methods:
If the calculations for each condition are complex, we can extract them into separate methods for better readability and maintainability.

4.4.4.2 Second Code snippet

```
public class Customer {
    private String name;
    private String address;
    private double balance;

    public Customer(String name, String address) {
        this.name = name;
        this.address = address;
        this.balance = 0;
    }

    public void deposit(double amount) {
        this.balance += amount;
    }

    public void withdraw(double amount) {
        this.balance -= amount;
    }

    public double getBalance() {
        return balance;
    }
}
```

Figure 4.2: Second Code Snippet

This code in figure 4.2 defines a Java class called Customer. Each customer has a name, an address, and a balance.

When a new customer is created using the Customer constructor, they provide a name and an address, and their balance is set to zero.

Customers can deposit money into their account using the deposit method. Similarly, they can withdraw money using the withdraw method. The **getBalance**

method returns the current balance of the customer's account.

Overall, this code provides a simple representation of a customer with basic functionalities to manage their account balance.

There are several ways the Customer class can be refactored for better quality. Here are a few examples:

1.Encapsulation: Use getter and setter methods for the name, address, and balance fields instead of making them public. This encapsulates the class's internal state and provides better control over access to these fields.

2.Validation: Add validation checks to the deposit and withdraw methods to ensure that the amount being deposited or withdrawn is valid (e.g., non-negative, not exceeding certain limits).

3.Immutable Class: Make the Customer class immutable by removing the setter methods for name and address. This ensures that once a customer object is created, its state cannot be changed. Instead, provide a constructor to initialize these fields.

4.4.4.3 Third Code snippet

```
public class Customer {
    private String name;
    private String address;
    private double balance;

    public Customer(String name, String address, double
initialBalance) {
        this.name = name;
        this.address = address;
        this.balance = initialBalance;
    }

    public void processPayment(double amount) {
        if (amount > balance) {
            throw new InsufficientFundsException();
        }
        balance -= amount;
    }

    public void printStatement() {
        System.out.println("Customer name: " + name);
        System.out.println("Customer address: " + address);
        System.out.println("Customer balance: " + balance);
    }
}
```

Figure 4.3: Third Code Snippet

The code in Figure 4.3 defines a Java class called `Customer`. Each customer has a name, an address, and a balance.When a new customer is created using the `Customer` constructor, they are required to provide a name, an address, and an

initial balance. The initial balance is set based on the value provided at the time of creation.

Customers can make payments using the **processPayment** method, which decreases their balance by the payment amount, provided there is enough balance to cover the payment. If the balance is insufficient, an **InsufficientFundsException** is thrown, indicating the customer does not have enough funds to complete the transaction.The **printStatement** method allows for printing the customer's name, address, and current balance to the console, offering a summary of the customer's account details.

Overall, this code provides a simplistic representation of a customer with functionalities to manage their balance through payments and to view their account details. Refactoring the Customer class in Java can enhance its design, maintainability, and performance. Here are several ways this class can be refactored:

1.Encapsulation and Data Hiding:

Provide getter methods for accessing private fields (name, address, and balance) if necessary, ensuring that data can only be modified in controlled ways.

. **2.Exception Handling:**

. Define the **InsufficientFundsException** class if not already defined. Ensure it extends an appropriate exception class, like RuntimeException or a more specific custom exception class.

. **3.Use BigDecimal for Monetary Values:** Replace double with **BigDecimal** for the balance and amount to ensure precise monetary calculations and avoid floating-point errors.

. **4.Immutability:** If the Customer objects do not need to change after creation, consider making the class immutable. This involves removing setters or methods that modify state and ensuring all fields are final.

. **5.Adding Functionality:** Add methods to deposit to the balance, enhancing the class's functionality. Implement an **updateAddress** method if address changes need to be handled.

.

4.4.4.4 Fourth Code snippet

The provided Java code snippet in figure 4.4 defines a **ShippingService** class with a single method **calculateShippingCost**, which calculates the shipping cost for an `Order` based on its total price and weight. The **calculateShippingCost** method takes an **Order** object as its parameter, which presumably has methods to retrieve the total price **getTotalPrice()** and the weight **getWeight()** of the order. This method implements a tiered shipping cost strategy, **whRefactoring** the provided **ShippingService** class to improve its readability, maintainability, and possibly its performance involves several steps. Here's a brief overview of the refactoring process:

1.Extract Method for Conditionals: The nested conditional logic within **calculateShippingCost** can be extracted into separate methods to clarify the intent of each condition. For example, methods like **isEligibleForDiscount** and **calcu-**

```java
public class ShippingService {
    public double calculateShippingCost(Order order) {
        double totalPrice = order.getTotalPrice();
        double weight = order.getWeight();
        if (totalPrice > 100) {
            if (weight > 10) {

                return totalPrice * 0.2;
            } else {

                return totalPrice * 0.05;
            }
        } else {

            return 0;
        }
    }
}
```

Figure 4.4: Fourth Code Snippet

lateDiscountcan make the code more readable.

2.Simplify Conditional Expressions: The nested if-else structure can be simplified. Since there's a clear return statement within each conditional branch, we can utilize early returns to reduce nesting and improve readability.

3.Use Descriptive Variable Names: While the variable names are relatively clear, ensuring that all names precisely describe their purpose can aid understanding. For example, renaming **totalPrice** to **orderTotalPrice** could provide immediate context.

4.Error Handling: Adding error handling or validation checks, such as ensuring order is not null and that its properties (**totalPrice** and **weight**) are within expected ranges, can prevent runtime errors.

5.Comments and Documentation: Adding comments or documentation to the method and the class itself can help other developers understand the purpose of the code and any specific logic applied in the calculations.

By following these steps, the **ShippingService** class can be made more efficient, understandable, and easier to maintain, enhancing overall code quality.ere the cost is determined by both the total price and the weight of the order, incentivizing higher-value orders with potentially higher shipping discounts or charges based on the weight.

4.4.4.5 Fifth Code snippet

This Java code in figure 4.5 defines a class named `Order` that models an order in a simple e-commerce or retail system. The **Order** class includes fields for the customer's name **customerName**, the product name **productName**, the price

```
import java.util.concurrent.atomic.AtomicInteger;
public class Order {
    private String customerName;
    private String productName;
    private double price;
    private int orderId;
    private static final AtomicInteger orderIdGenerator = new
AtomicInteger(1000);
    public Order(String customerName, String productName, double price) {
        this.customerName = customerName;
        this.productName = productName;
        this.price = price;
        this.orderId = orderIdGenerator.incrementAndGet();
    }
    public String toString() {
        String nameAndPrice = customerName + "," + String.valueOf(price);
        return nameAndPrice + "," + orderId;
    }
}
```

Figure 4.5: Fifth Code Snippet

of the product **price**, and a unique order ID **orderId**. A static **AtomicInteger** named **orderIdGenerator** is used to generate unique order IDs, starting from 1000 and incrementing with each new order instance created. The constructor of the **Order** class takes the customer name, product name, and price as parameters, sets the corresponding instance variables, and automatically assigns a unique order ID by incrementing **orderIdGenerator**. The **toString** method overrides the default implementation to return a string representation of the order, which includes the customer name, price, and order ID, separated by commas. This class illustrates the use of atomic operations for generating unique IDs and encapsulates the details of an order in a straightforward manner.

Refactoring the provided Java code can make it more maintainable, readable, and potentially efficient. Here are the steps to refactor the given code snippet:

1. **Encapsulation:** Ensure all fields are private and provide public getter methods for necessary fields to maintain encapsulation and data integrity.

2. **Formatting:** Properly format the code to improve readability. This includes organizing the declaration of fields, constructor, and methods in a consistent manner, and ensuring there is appropriate spacing and indentation.

3. **Use of `String.format`:** Instead of concatenating strings using `+`, use `String.format` for constructing the **toString** method's return value. This enhances readability and maintainability.

4. **Final Fields:** Make immutable fields final (e.g., **customerName**, **productName**, **price**) to convey intention clearly that these fields are not expected

to change once an instance is created.

5. **Comments and Documentation:** Add comments and/or JavaDoc documentation to explain the purpose of the class, the role of the **orderIdGenerator**, and any non-obvious logic within methods. This is particularly important for the **toString** method and the constructor.

4.4.5 Pretest

Before commencing the experimental intervention with ChatGPT, a pretest was designed to establish a baseline of participants' current code refactoring skills and efficiency. The pretest consisted of a set of Java code snippets, each embodying common refactoring challenges, such as redundant code, inefficient algorithms, and poor readability. Participants were asked to refactor these snippets following best practices to enhance code quality without altering the intended functionality.

The pretest aimed to assess:

1. **Time Efficiency:** The duration taken by participants to complete each refactoring task.

2. **Quality of Refactoring:** Based on established code quality including lines of code and adherence to coding standards.

3. **Participants' Confidence:** Through a short questionnaire, participants rated their confidence in their refactoring solutions on a Likert scale.

This pretest data served as a crucial benchmark for comparing the impact of utilizing ChatGPT in the subsequent phases of the experiment.

4.4.6 Post-test

Following the intervention period, where participants utilized ChatGPT for code refactoring tasks, a post-test identical in structure to the pretest was administered. This was done to evaluate any changes in the participants' refactoring efficiency, quality, and confidence. The same set of Java code snippets were utilized in both the pre-test and post-test to maintain consistency in the tasks being evaluated.

The post-test aimed to measure:

- **Improvements in Time Efficiency:** Reduction in the time taken to refactor code snippets as compared to the pretest.

- **Enhancements in Quality:** Improvements in code quality metrics reflecting the impact of using ChatGPT.

- **Change in Confidence Levels:** Participants' perceived confidence in their refactoring solutions after using ChatGPT, compared to their pretest responses.

The comparison between pretest and post-test results was critical in assessing the effectiveness of ChatGPT as a tool for assisting with code refactoring.

4.4.7 Post Interview Questionnaires

To complement the quantitative data from the pretest and post-test, post-interview questionnaires were conducted to gather qualitative insights into the participants' experiences using ChatGPT for code refactoring. The questionnaires explored the following areas:

Perceived Advantages

Participants' views on how ChatGPT facilitated their refactoring process were gathered, including aspects such as:

- Idea generation
- Code optimization suggestions
- Error identification

Challenges Encountered

Participants shared the difficulties faced while using ChatGPT, which included:

- Misinterpretations of code intent
- Limitations in AI's understanding of complex programming constructs
- Integration issues into existing workflows

Usability and Integration

Feedback on the ease of integrating ChatGPT into their development environment and its usability for refactoring tasks was collected to assess:

- Integration challenges and solutions
- Usability feedback for enhancing tool interaction

Future Use and Recommendations

Insights into participants' willingness to continue using ChatGPT for refactoring and their suggestions for improving the tool's effectiveness in software development processes were explored, focusing on:

- Potential areas for further research and tool enhancement
- Recommendations for improving ChatGPT's utility in code refactoring

These questionnaires aimed to provide a comprehensive understanding of the qualitative aspects of using ChatGPT for code refactoring, complementing the quantitative findings and offering insights into potential areas for further research and tool enhancement.

By addressing these components in the thesis, a detailed account of the experiment's methodology is provided, encompassing both the quantitative and qualitative assessments of using ChatGPT in code refactoring tasks among expert Java programmers.

Question Number	**Questions**
Question 1	Can you share your experiences using ChatGPT for code refactoring? What were the specific benefits or advantages you observed during the process?
Question 2	In what ways did ChatGPT enhance your productivity and efficiency in completing code refactoring tasks? Please provide specific examples
Question 3	Did ChatGPT help you discover new refactoring techniques or approaches that you were previously unaware of? If yes, please elaborate on these insights.
Question 4	How did ChatGPT contribute to the maintainability and readability of the code you produced during refactoring? Were there any notable improvements or challenges in this aspect?
Question 5	Were there any specific challenges or limitations you encountered while using ChatGPT for code refactoring? How did you overcome them, if at all?
Question 6	In what scenarios do you believe AI assistance, like ChatGPT, is most beneficial for code refactoring? Conversely, are there situations where you think it might be less effective or not suitable at all?
Question 7	How does ChatGPT's performance vary depending on the complexity of the code?
Question 8	Would you recommend ChatGPT to other Java programmers?

Table 4.2: Post Interview Questionnaires

4.5 Experimental Design

The core of any empirical study is its experimental design. This design not only connects theoretical concepts with practical applications but also validates the reliability and accuracy of the research outcomes. Our investigation, conducted during the winter semester of 2023/24, aims to explore the effectiveness of ChatGPT in

assisting expert Java programmers with code refactoring tasks. The design of our experiment carefully outlines our methodological approach, allowing for a structured exploration of ChatGPT's capabilities and limitations within the realm of Java programming.

1.Participant Selection and Recruitment

The success of our study largely hinges on the selection of a representative sample of participants, which in turn, significantly influences the applicability of our findings. We targeted a diverse range of Java programmers from novices to experts to ensure a thorough examination of ChatGPT's utility across varying levels of expertise. A total of 14 programmers were purposefully selected based on their proficiency in Java, willingness to participate, and availability during the study period. This selection strategy was crucial for assembling a group that accurately reflects the broader Java programming community.

2.Survey Design and Implementation

The primary tool for data collection in our study was a survey developed on the LimeSurvey platform, chosen for its robustness and adaptability. The survey was intricately designed to capture detailed responses regarding the programmers' interactions with ChatGPT during code refactoring. A variety of question types, from multiple-choice to open-ended, were utilized to collect both quantitative and qualitative data. The implementation phase included a pilot test with a subset of participants, which helped refine the survey questions and ensured the clarity of the survey instructions.

3.Data Confidentiality and Processing

Acknowledging the sensitivity of the data and the privacy of our participants, stringent measures were adopted to ensure confidentiality. All participant data were anonymized, with personal identifiers removed or encrypted. The Limesurvey platform was configured to comply with data protection regulations, ensuring that the collected data were stored securely and only accessible to the research team. Following data collection, a rigorous data processing protocol was established, involving data cleaning, coding, and preparation for analysis. This process was critical in ensuring the integrity and usability of the data for subsequent analytical phases.

4.Expected Outcomes and Analytical Approach

The experimental design is geared towards achieving a dual objective: firstly, to assess the effectiveness and efficiency of ChatGPT in aiding Java programmers with code refactoring tasks; and secondly, to identify any barriers or limitations encountered. We anticipate that our findings will provide valuable insights into the practical

applications of ChatGPT in software development, particularly in code refactoring. The analytical approach will involve a mix of statistical analysis for quantitative data and thematic analysis for qualitative responses, enabling a holistic understanding of the phenomena under study.

In conclusion, the experimental design of our study lays a solid foundation for exploring the potentials and challenges of using ChatGPT in code refactoring among Java programmers. By systematically examining each aspect of the design, from participant recruitment to data analysis, we aim to contribute significantly to the existing body of knowledge on AI's role in software development processes.

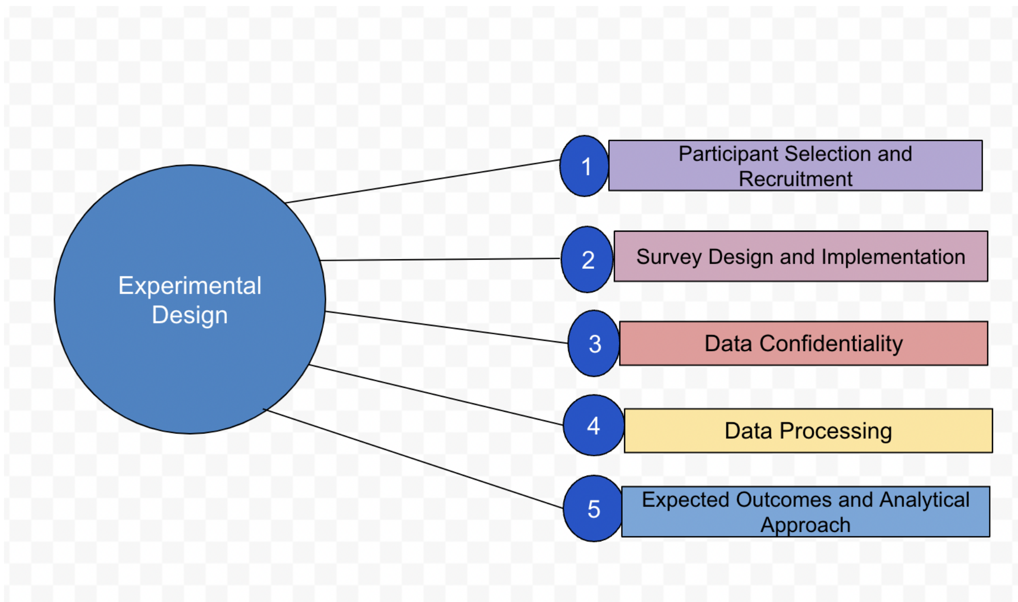

Figure 4.6: Experimental Design Steps

5 The experiment process

Detailed information regarding the implementation of this study can be found in the following chapter, which outlines the structured actions that were taken in order to facilitate its execution. The first phase began at the beginning of the winter semester of 2023/24, and it consisted of the methodical creation of a thorough survey through the use of the Limesurvey site through the registration process for TUC OPAL Access. Participants were provided with a link to a survey that was generated from the Linesurvey and distributed via a variety of internet platforms. Although about forty participants attempted to finish the survey in its entirety, only fourteen of them were successful in doing so. Data was collected from 09 January, 2024 to 25 February ,2024. It is of the highest importance to emphasize that the information gathered from these participants will be handled with the utmost confidentiality and processed in an anonymous manner in order to guarantee the authenticity and anonymity of the conclusions associated with the study.

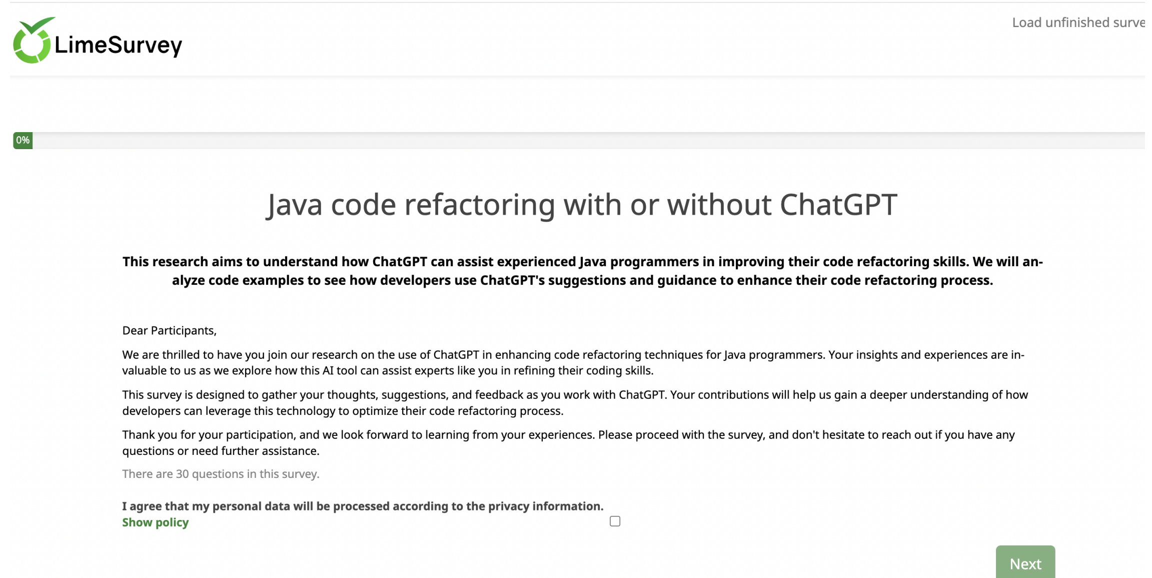

Figure 5.1: Survey Starting page

5.1 Survey Pre-Questionnaires

Upon completing the introductory section of the survey, participants proceeded to the survey questionnaires, where they were required to answer a series of short questions, some of which were mandatory and others optional. The inclusion of a mandatory inquiry was necessary in order to gather the specific data required for generating the intended outcome. The task was completed quickly and there were a total of 12 questions.

Figure 5.2: Survey Pre-Questionnaires page

5.2 Pretest

Prior to directly entering the Java code snippet portion, a brief and significant explanation was provided. So they can readily comprehend the norms and regulations of the coding duties. A total of five codes were provided in this section, where the use of AI or ChatGPT was prohibited. Each coding job had a time constraint of 3 minutes. After a duration of 3 minutes, the timeout event was triggered and the subsequent code snippet commenced.

5.3 Posttest

Upon finishing five pretest code snippets, the participants proceeded to the post-test coding challenge. The Java codes had stayed unchanged, but in this area, developers could utilize Chat GPT to obtain recommendations for code refactoring.

LimeSurvey

Resume later Exit and clear survey

Task Explanation

Your expertise is crucial for advancing our understanding of code improvement practices. In this survey, you will be working on five Java code snippets in two sections, each presenting an opportunity for refinement.

Tasks Overview:

In each of the two sections, you will encounter five Java code snippets that require refactoring. The first five snippets must be refactored without assistance from ChatGPT, while the last five snippets can be refactored with the aid of ChatGPT. Primarily, you have two alternatives:

1. **Without Assistance:** Refactor the code on your own, relying on your existing knowledge and skills.
2. **With ChatGPT Assistance:** Utilize the assistance of ChatGPT to receive suggestions and guidance for refactoring the code.

Timing:

Each assignment must be completed within a strict time constraint of 3 minutes. You must complete the work within a 3-minute timeframe, otherwise, timeouts will occur. Efficiently allocate your time to ensure timely completion of all jobs.

Instructions:

1. **Read the code:** Begin by thoroughly understanding the provided Java code snippet.
2. **Refactor:** Apply your refactoring skills to improve the code based on the given criteria (readability, efficiency, maintainability, etc.).

Figure 5.3: Survey Task explanation page

Here also each coding job had a time constraint of 3 minutes. After a duration of 3 minutes, the timeout event was triggered and the subsequent code snippet commenced.

5.4 Interview Questions

Following the completion of the coding phase, participants proceeded to the final chapter of our survey, where they encountered qualitative items. There was no time constraint for the portion. They can provide concise responses to the inquiries. A total of 8 questions were provided in that area.

Upon finishing all the necessary steps, the users pressed the submit button, causing the answer to be automatically stored in LimeSurvey. Finally, we have the option to download the responses in several formats in order to generate our desired output.

6 Results and Analysis

This chapter provides a comprehensive overview of the systematic phases of data collection and processing undertaken in this study before delving into the analysis of collected data.

6.1 Data collection and processing

The data collection phase began by creating a thorough study plan that carefully defined the variables of interest and the procedures for measuring them. The design phase of the work included a thorough examination of relevant literature and discussions with professionals in the industry to guarantee the use of suitable metrics and procedures.

6.1.1 Data collection

The initial phase involved retrieving pre-test and post-test data from Lime Survey in Excel format. Lime Survey automatically organizes survey responses, offering diverse options for data export. Utilizing the "Responses and statistics → Export → Export result" feature, data could be downloaded in various formats including Excel, CSV, PDF, R (syntax file), R (data file), HTML, and Microsoft Word. Excel and PDF formats were selected due to their ability to present data in a structured manner, facilitating easier analysis and interpretation. Each response dataset was tagged with a unique "Response ID" for identification. The collect dataset, as illustrated in Figure 6.1, showcased additional columns such as participant seed numbers, capturing survey start and end times. Prior to analysis, data cleaning procedures were imperative.

6.1.2 Data processing

For the purpose of evaluating the advantages and challenges of utilizing ChatGPT for code refactoring among expert Java programmers, LimeSurvey was employed as the primary tool for data collection. This online survey tool facilitated the distribution of both pretest and posttest questionnaires, alongside programming comprehension tasks tailored to assess the participants' code refactoring skills.

The versatility of LimeSurvey allowed for the seamless administration of surveys and the collection of responses in real-time. To cater to the diverse analytical needs of this study, data was downloaded in various formats, including PDF and Excel,

Java code refactoring with or without ChatGPT

Survey response 1

Response ID
5
Date submitted
2024-01-12 22:46:17
Last page
13
Start language
en
Seed
18576555
Date started
2024-01-12 22:17:25
Date last action
2024-01-12 22:46:17
Total time
1737.8

Figure 6.1: Example of Collected front page dataset

among others. This multiplicity in data formats was instrumental in enabling a comparative analysis that leveraged the strengths of each format for a comprehensive understanding of the gathered data.

1. Format-Specific Preprocessing: Focus on processing data in two specific formats.

PDF Data: The PDF format was particularly useful for preserving the layout and formatting of responses for qualitative analysis. Data extracted from PDFs was digitized, as needed, to transform response texts into editable formats suitable for thematic analysis.

Excel Data: Excel files offered a structured format ideal for quantitative analysis. Preprocessing involved filtering irrelevant columns, renaming variables for clarity, and using pivot tables and formulas to summarize response trends and patterns. Excel's data manipulation capabilities facilitated the identification of outliers and the calculation of descriptive statistics.

2. Comparative Analysis: The study employed a dual-faceted approach to compare results across the different data formats:

Qualitative Insights: The PDF format, with its intact presentation of survey questions and responses, allowed for a detailed narrative analysis. This was pivotal in understanding the contextual nuances behind participants' experiences with ChatGPT, including perceived benefits and challenges.

Quantitative Metrics: Excel's analytical prowess enabled the computation of response times, correctness rates, and the categorization of errors into logical and syntactic. This quantitative analysis provided a measurable assessment of the impact of using ChatGPT on code refactoring efficiency and effectiveness.

6.2 Data analysis

The synthesis of quantitative and qualitative data analysis provided a holistic understanding of ChatGPT's utility in code refactoring. While ChatGPT significantly enhances efficiency, reduces errors, and improves code quality, it also presents challenges that necessitate cautious integration into programming practices. This nuanced view underscores the potential of AI in software development while highlighting areas for future research and development to maximize the benefits and mitigate the drawbacks of AI-assisted code refactoring.

6.2.1 Descriptive statistics

Descriptive statistics are quantitative metrics that provide a summary and description of the characteristics of a dataset. They offer valuable information on the average, spread, and form of the data distribution. For this study on code refactoring with ChatGPT, descriptive statistics can be used to assess many aspects of the obtained data, including the quality of refactored code, response times, and perceptions of utilizing ChatGPT.

6.2.1.1 Description of the subjects

According to the survey results, out of the 14 participants, the majority (64.3 %) had 1-2 years of expertise in Java programming, which was the most common group. 3 respondents (21.4 %) reported having 6–8 years of Java expertise, making it the second most frequent response. Merely 2 participants (14.3 %) reported possessing a Java proficiency ranging from 2 to 4 years. When considering the data as a whole, we observe that 78.6 % of the participants possess Java expertise ranging from 0 to 4 years, while all participants have experience ranging from 0 to 8 years. To summarize, the findings indicate that the participants generally possess limited proficiency in Java, with most of them having 1-2 years of experience. Approximately 20 % possess a professional background of 6–8 years, although a small minority (14.3 %) fall within the middle range of 2-4 years. In general, the distribution is biased towards less experienced Java programmers who have been working for less than 4 years.

Valid	Frequency	Percent	Valid Percent	Cumulative Percent
1 to 2	9	64.3	64.3	64.3
2 to 4	2	14.3	14.3	78.6
6 to 8	3	21.4	21.4	100.0
Total	14	100	100	

Table 6.1: How many years of experience do you have in Java Programming?

According to the findings, out of the 14 participants, the majority (71.4%, or 10 participants) possess a decade of expertise in programming for extensive software projects within a corporate environment. Out of the total respondents, 4 individuals, accounting for 28.6%, possess 4 years of expertise in working on similar projects. To summarize, all the participants have professional programming experience ranging from 4 to 10 years on software projects within larger companies, with 10 years being the most frequently mentioned duration. The overwhelming majority already have more than 5 years of experience. The data suggests that the respondents possess significant practical programming experience in professional settings, rather than being limited to personal or academic projects. On average, the respondents had accumulated around 10 years of experience, with a tendency towards higher values.

Valid	Frequency	Percent	Valid Percent	Cumulative Percent
0	4	28.6	28.6	6
0 to 10	10	71.4	71.4	100.
Total	14	100.0	100.0	

Table 6.2: For how many years have you been programming for larger software projects e.g. in a company? Please enter a number between 0 and 30.

The data on code restructuring experience over the years indicates that, among the 14 participants, the majority (42.9%) reported having 1-2 years of experience, which was the most often mentioned response. Subsequently, three respondents (21.4%) indicated a lack of code restructuring experience. The remaining responses were allocated among the following time periods: 2-3 years (7.1%), 4-5 years (7.1%), 6-7 years (14.3%), and 9-12 years (7.1%). When considering the data as a whole, it can be observed that the majority of respondents (64.3%) possess a maximum of 2 years of expertise in refactoring. All participants possess a maximum of 12 years of experience in the field of code restructuring. To summarize, the results suggest that the respondents have varying levels of experience in refactoring, with a majority having no experience or less than 2 years of experience. A small minority indicated extensive experience of six or more years dedicated to code restructuring assignments.

The data on code restructuring experience over the years indicates that, among the 14 participants, the majority (42.9%) reported having 1-2 years of experience, which was the most often mentioned response. Subsequently, three respondents (21.4%) indicated a lack of code restructuring experience. The remaining responses were allocated among the following time periods: 2-3 years (7.1%), 4-5 years (7.1%), 6-7 years (14.3%), and 9-12 years (7.1%). When considering the data as a whole, it can be observed that the majority of respondents (64.3%) possess a maximum of 2 years of expertise in refactoring. All participants possess a maximum of 12 years of experience in the field of code restructuring. To summarize, the results

Valid	Frequency	Percent	Valid Percent	Cumulative Percent
0	3	21.4	21.4	21.4
1 to 2	6	42.9	42.9	64.3
2 to 3	1	7.1	7.1	71.4
4 to 5	1	7.1	7.1	78.6
6 to 7	2	14.3	14.3	92.9
9 to 12	1	7.1	7.1	100.0
Total	14	100.0	100.0	

Table 6.3: How many years of experience do you have with code refactoring?

suggest that the respondents have varying levels of experience in refactoring, with a majority having no experience or less than 2 years of experience. A small minority indicated extensive experience of six or more years dedicated to code restructuring assignments.

Valid	Frequency	Percent	Valid Percent	Cumulative Percent
Yes	14	100.0	100.0	100.0

Table 6.4: Did you study programming or computer science at a university?

The table shows that all 14 respondents (100%) studied programming or computer science at the university level.

Valid	Frequency	Percent	Valid Percent	Cumulative Percent
1 to 2	7	50.0	50.0	50.0
2 to 4	7	50.0	50.0	100.0
Total	14	100.0	100.0	

Table 6.5: During your education, how many courses did you take where Java was the primary language?

When questioned about the number of classes in which Java was predominantly used during their study, the responses were evenly divided. Seven respondents (50%) reported taking 1-2 Java courses, while the other seven respondents (50%) attended between 2-4 courses where Java was the primary programming language. To summarize, all participants possess a formal tertiary-level education in computer science or programming. Regarding the respondents' experience with Java in school, there is an equal distribution between those who have taken 1-2 courses that specifically focus on Java and those who have taken 2-4 Java courses. Although all respondents possess academic credentials in computer science and programming, their level of

experience with Java coursework varies, with around half having limited exposure compared to those with extensive exposure. However, none of the participants indicated that they had taken more than four Java classes during their time at university.

Among the 14 respondents, the self-reported expertise levels for Java programming indicate that the most frequent response was "neither inexperienced nor experienced," with 6 selections (42.9%). The second most prevalent category was "experienced," with 4 responses, accounting for 28.6% of the total. This was followed by "very inexperienced" and "inexperienced," with two responses each, making up 14.3% each. Merely one participant, accounting for 7.1% of the total, identified themselves as "highly experienced." When considering the data as a whole, 21.4% of individuals perceive themselves as slightly lacking in experience, while 35.7% consider themselves to be experienced. Although there is a range of ability levels, the majority of respondents positioned themselves in the middle, indicating that they are neither novices nor specialists. The findings suggest that there is a scarcity of individuals with advanced Java programming skills within this particular set of respondents. Most individuals possess a functional understanding of the language, although they may not necessarily have complete expertise or mastery. When evalu-

Valid	Frequency	Percent	Valid Percent	Cumulative Percent
Very Low	3	21.4	21.4	21.4
Low	2	14.3	14.3	35.7
Moderate	5	35.7	35.7	71.4
High	2	14.3	14.3	85.7
Very High	2	14.3	14.3	100.0
Total	14	100.0	100.0	

Table 6.6: How often have you used Chat GPT (e.g 1-low, 5-high)?

ating their personal proficiency in Java in comparison to individuals with more than 20 years of hands-on experience, the prevailing reaction was "inexperienced," with 5 out of 14 participants (35.7%) selecting this option. Each of the 3 respondents (21.4% each) identified themselves as "very inexperienced," "neither inexperienced nor experienced," and "experienced," respectively, when compared to experienced individuals in the language. Overall, 57.1% of individuals perceive themselves as somewhat inexperienced compared to specialists with 20 years of experience, whereas 21.4% have a neutral stance and another 21.4% consider themselves experienced.
The distribution of replies regarding one's Java competence, as compared to peers and coworkers, was more evenly balanced. The predominant response was "neither inexperienced nor experienced," selected by 5 out of 14 respondents (35.7%). Exactly 5 respondents, which accounts for 35.7% of the total, considered themselves to be "experienced" compared to their colleagues. Among their coworkers, 14.3% of the respondents, or 2 individuals, said that they are either "very inexperienced"

or "inexperienced." Out of the total, 28.6% consider themselves to be below average, while 71.4% believe they are either average or above average compared to their direct counterparts. Approximately 25% acknowledge that they fall behind their peers, but the majority assess themselves favorably compared to other programmers within their social groups.
The poll inquired about the frequency at which respondents have utilized ChatGPT, using a rating scale ranging from little to maximal usage. The prevailing response was "moderate" consumption, selected by 5 out of 14 individuals (35.7%). Among the replies, "very low" and "low" usage were the second most frequent, chosen by 3 respondents (21.4%) and 2 respondents (14.3%), respectively. Regarding increased usage, 2 respondents (14.3% each) reported that their frequency of using ChatGPT is categorized as "high" or "very high." Out of all the respondents, 50% had a reduced utilization of ChatGPT, whereas 28.6% use it more frequently. When specifically

Valid	Frequency	Percent	Valid Percent	Cumulative Percent
Very Low	5	35.7	35.7	35.7
Low	1	7.1	7.1	42.9
Moderate	5	35.7	35.7	78.6
High	2	14.3	14.3	92.9
Very High	1	7.1	7.1	100.0
Total	14	100.0	100.0	

Table 6.7: Have you used ChatGPT for code refactoring tasks (e.g 1-low, 5-high)?

inquired about employing ChatGPT for code reworking tasks, the predominant response indicated "minimal" utilization, with 5 out of 14 participants (35.7%) opting for this option. Exactly 5 respondents, accounting for 35.7% of the total, reported a "moderate" level of utilization for refactoring. Out of the total number of responses, 2 individuals (14.3%) reported a "high" level of usage, 1 individual (7.1%) reported a "low" level of usage, and 1 individual (7.1%) reported a "very high" level of usage for code reworking with ChatGPT. Thus far, a significant proportion of respondents, specifically 42.9%, have indicated a low or very low level of usage of ChatGPT for coding activities. Conversely, 21.4% of individuals characterize their usage as high to extremely high. Experiences with ChatGPT vary greatly; some individuals have only scratched the surface of its potential, while others are now extensively utilizing it for programming and refactoring support. However, the most significant portion indicates a decent level of integration into coding procedures up to now.

6.2.1.2 Response time of pretest code refactoring tasks

This table presents a detailed quantitative analysis of the response times for pretest code refactoring tasks undertaken by 14 participants. Although 14 participants completed the whole survey, about 10 participants completed the total snippets

of pretest and post-test. Because some of them skipped one or more tasks. The tasks, labeled from Question 1 to Question 5, were designed to assess the efficiency and proficiency of the participants in performing code refactoring under varying levels of complexity. Each question was analyzed based on the total number of participants (N), the minimum and maximum time taken to complete the task, the mean (average) time, and the standard deviation, which measures the variability of the response times among participants showing in figure 6.2.

Question 1 demonstrates a higher level of complexity or challenge, as indicated by the longer average time required to complete the task (approximately 131.81 seconds). The range between the minimum and maximum times (85.05s to 180s) suggests varied participant familiarity and proficiency with the specific refactoring task, supported by a standard deviation of approximately 35.44 seconds.

Question 2 also shows significant variability in response times (87.22s to 179.66s), with an average completion time of around 129.4 seconds. The high standard deviation of approximately 36.94 seconds further highlights the diverse approaches and strategies adopted by the participants in tackling the refactoring challenges.

Question 3 exhibits a noticeable decrease in the mean response time to roughly 142.88 seconds, indicating potentially lower task complexity or greater participant comfort with the task requirements. However, the broad range of times (66.96s to 180s) and a high standard deviation of approximately 38.99 seconds point to significant differences in participant performance and strategy.

Question 4 is characterized by the shortest mean response time of approximately 98.6 seconds, suggesting it was relatively easier for the participants compared to the other tasks. The narrower range of response times (61.44s to 132.67s) and a lower standard deviation of around 24.35 seconds further imply a more consistent performance among participants on this task.

Question 5, with a mean time of approximately 121.51 seconds, falls in the middle range of complexity as inferred from the response times of the other questions. The response times ranged from 68.65s to 179.11s, with a standard deviation of approximately 38.1 seconds, indicating a moderate level of variability in how participants approached and executed the refactoring task.

Overall, the data from these pretest code refactoring tasks reveal insightful trends about the participants' refactoring skills and efficiency. The varied mean times and standard deviations across the questions indicate that different tasks elicited a wide range of performances, likely influenced by the task's nature, complexity, and participants' individual skill levels and experience with code refactoring. This variability underscores the importance of diverse and targeted practice in refining code refactoring proficiency among developers.

Notably, In instances where participants completed tasks in less than three seconds, these instances were interpreted as task skips rather than genuine task completions. Consequently, such timings were excluded from our data sets to ensure that our calculations accurately reflect the actual effort exerted by participants.

Question Number	Minimum	Maximum	Mean	Std. Deviation
Question 1	85.5s	180s	131.81	35.44
Question 2	87.22s	179.66s	129.40	36.94
Question 3	66.96s	180s	142.88	38.99
Question 4	61.44s	132.67s	98.6	24.35
Question 5	68.55s	179.11s	121.51	38.10

Table 6.8: Analysis of Participant Response Times in Code Refactoring Pretest Tasks

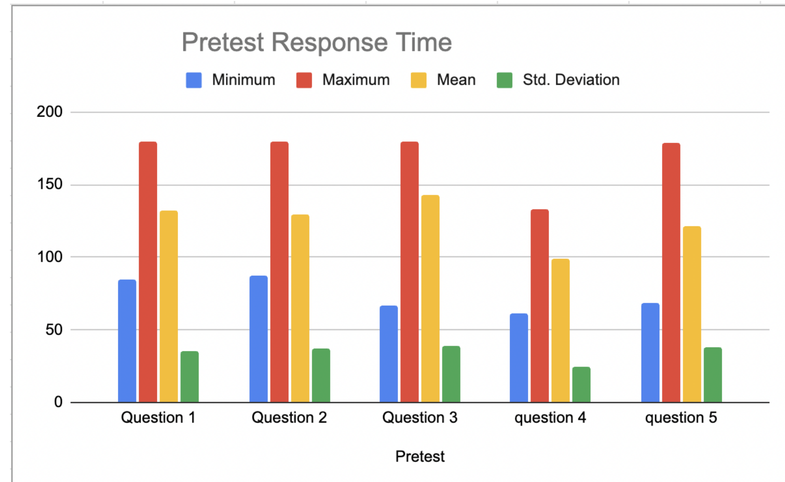

Figure 6.2: Pretest Code Refactoring Response Times

6.2.1.3 Response time of post-test code refactoring tasks

This table provides a comprehensive quantitative analysis of the response times for post-test code refactoring tasks performed by 14 participants. The tasks, labeled from Question 1 to Question 5, aimed to evaluate the effectiveness of participants in executing code refactoring, possibly after being introduced to or practicing with an AI tool like ChatGPT. Analysis metrics include the total number of participants (N), the minimum and maximum times taken to complete each task, the mean time, and the standard deviation, which indicates the spread of response times across participants showing in figure 6.3.

Question 1 had the longest mean response time at approximately 81.56 seconds, suggesting either a higher level of task complexity or that participants took a cautious approach to ensure accuracy in refactoring. The broad range in response times (38.47s to 165.18s) and a standard deviation of approximately 42.46 seconds indicate considerable variability in participants' proficiency and strategies in tackling the task.

For Question 2, the mean response time reduced to roughly 52 seconds, with

response times ranging from 36.29s to 109.67s. The relatively lower standard deviation of about 20.66 seconds compared to Question 1 suggests participants were more uniform in their approach and execution, possibly indicating a task of intermediate complexity or better alignment with participants' skills.

Question 3 showcased the shortest mean response time at around 49.83 seconds, hinting at either an easier task or improved participant efficiency. The response time span from 31.87s to 88.70s, alongside a standard deviation of approximately 19.49 seconds, underscores a more consistent performance across the board, reflecting either familiar task patterns or effective application of refactoring techniques.

Question 4, with a mean time of approximately 52.45 seconds, presented a unique challenge, as evidenced by the maximum response time of 145.52s. Despite this outlier, the standard deviation of around 33.71 seconds points to a fair amount of consistency in participant responses, possibly indicating a balanced task complexity or an effective grasp of required refactoring methodologies by the majority of participants.

Lastly, **Question 5** had a mean response time of about 58.03 seconds, situating it amongst the tasks with moderate complexity based on participant response times, which ranged from 26.69s to 139.20s. The standard deviation of approximately 32.89 seconds suggests a moderate level of variation in how participants approached the refactoring task, potentially reflecting diverse interpretations of the task requirements or variability in comfort levels with the refactoring tools or techniques employed.

In summary, the analysis of post-test code refactoring tasks illuminates significant insights into the learning curve, efficiency, and adaptability of participants in employing new tools or techniques for code refactoring. The variance in mean response times and standard deviations across the tasks reveals a nuanced picture of participants' engagement with the refactoring process, highlighting areas of strength and opportunities for further skill development. The progression from pre-test to post-test tasks likely reflects an evolutionary learning process, where exposure to new methodologies or tools such as ChatGPT facilitates a refined approach to code refactoring, ultimately aiming to enhance code quality and developer productivity.

Question Number	Minimum	Maximum	Mean	Std. Deviation
Question 1	38.47s	165.18s	81.56	42.46
Question 2	36.29s	109.67s	52.06	20.66
Question 3	31.87s	88.7s	49.83	19.49
Question 4	27.72s	145.52s	52.45	33.71
Question 5	26.69s	139.2s	58.03	32.89

Table 6.9: Analysis of Participant Response Times in Code Refactoring Posttest Tasks

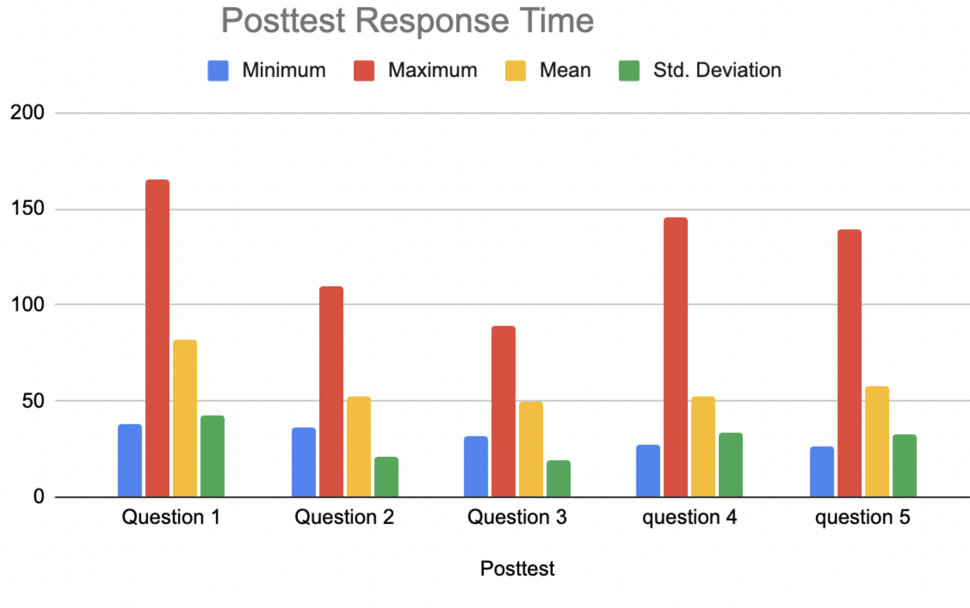

Figure 6.3: Posttest Code Refactoring Response Times

6.2.1.4 Correctness of Pretest and Post-test code refactoring tasks

The study involved participants tasked with refactoring Java code snippets both with and without the assistance of ChatGPT. The objective was to assess whether ChatGPT could improve the quality of code refactoring in terms of readability, efficiency, and maintainability. For example in figure 6.4 and in figure 6.5 while the performance differences for constructing individual Order objects might be negligible in many practical applications, the second version provides a cleaner, more maintainable, and potentially more efficient implementation, particularly in terms of string handling and object immutability. The choice of String.format() over direct concatenation can lead to more predictable and manageable code, which is crucial in larger, more complex software systems where performance and maintainability are critical. And the second one also taking less time.
Sample two (Without ChatGPT): The original code used nested if-else statements to calculate a payment amount based on various conditions such as **isDead**, **isSeparated**, and **isRetired** in figure 6.6.

Sample two (With ChatGPT): The refactored code, with the assistance of ChatGPT, streamlined the conditional logic, making it more concise and easier to understand in figure 6.7.
Sample three (Without ChatGPT): The initial version of the Customer class included basic methods for depositing and withdrawing funds without any validation in figure 6.8.

Question 5

Refactor the below code snippet without ChatGPT within 3 minutes. import java.util.concurrent.atomic.AtomicInteger; public class Order { private String customerName; private String productName; private double price; private int orderId; private static final AtomicInteger orderIdGenerator = new AtomicInteger(1000); public Order(String customerName, String productName, double price) { this.customerName = customerName; this.productName = productName; this.price = price; this.orderId = orderIdGenerator.incrementAndGet(); } public String toString() { String nameAndPrice = customerName + "," + String.valueOf(price); return nameAndPrice + "," + orderId; }}

```
import java.util.concurrent.atomic.AtomicInteger;

public class Order {
   private String customerName;
   private String productName;
   private double price;
   private int orderId;

   private static final AtomicInteger orderIdGenerator = new AtomicInteger(1000);

   public Order(String customerName, String productName, double price) {
      this.customerName = customerName;
      this.productName = productName;
      this.price = price;
      this.orderId = orderIdGenerator.incrementAndGet();
   }

   public String toString() {
      String nameAndPrice = customerName + "," + String.valueOf(price);
      return nameAndPrice + "," + orderId;
   }
}
```

Group time: Question 5
172.43

Figure 6.4: Response sample one in pretest

Question 5

Refactor the below code snippet with ChatGPT within 3 minutes. import java.util.concurrent.atomic.AtomicInteger; public class Order { private String customerName; private String productName; private double price; private int orderId; private static final AtomicInteger orderIdGenerator = new AtomicInteger(1000); public Order(String customerName, String productName, double price) { this.customerName = customerName; this.productName = productName; this.price = price; this.orderId = orderIdGenerator.incrementAndGet(); } public String toString() { String nameAndPrice = customerName + "," + String.valueOf(price); return nameAndPrice + "," + orderId; }}

```
import java.util.concurrent.atomic.AtomicInteger;

public class Order {
   private static final AtomicInteger orderIdGenerator = new AtomicInteger(1000);

   private final String customerName;
   private final String productName;
   private final double price;
   private final int orderId;

   public Order(String customerName, String productName, double price) {
      this.customerName = customerName;
      this.productName = productName;
      this.price = price;
      this.orderId = orderIdGenerator.incrementAndGet();
   }

   @Override
   public String toString() {
      return String.format("%s,%.2f,%d", customerName, price, orderId);
   }
}
```

Group time: Question 5
29.47

Figure 6.5: Response sample one in posttest

```java
public double getPayAmount() {
    double result;
    if (isDead) {
        result = deadAmount();
    } else if (isSeparated) {
        result = separatedAmount();
    } else if (isRetired) {
        result = retiredAmount();
    } else {
        result = normalPayAmount();
    }
    return result;
}
```

Figure 6.6: Response sample two in pretest

```java
public double getPayAmount() {
    if (isDead) return deadAmount();
    if (isSeparated) return separatedAmount();
    if (isRetired) return retiredAmount();
    return normalPayAmount();
}
```

Figure 6.7: Response sample two in posttest

```java
public class Customer {
    private String name;
    private String address;
    private double balance;

    public void deposit(double amount) {
        balance += amount;
    }

    public void withdraw(double amount) {
        balance -= amount;
    }
}
```

Figure 6.8: Response sample three in pretest

Sample three (With ChatGPT): After refactoring with ChatGPT's assistance, validation checks were added to ensure amounts were positive, showcasing an improvement in code quality and robustness in figure 6.9.

```java
public class Customer {
    public void deposit(double amount) {
        if (amount > 0) balance += amount;
        else System.out.println("Amount must be positive");
    }

    public void withdraw(double amount) {
        if (amount > 0 && amount <= balance) balance -= amount;
        else System.out.println("Invalid withdrawal attempt");
    }
}
```

Figure 6.9: Response sample three in posttest

Sample four (Without ChatGPT):In the initial code snippet, the ShippingService class calculates shipping costs based on the order's total price and weight, using nested if-else statements in figure 6.10.

Sample Four (With ChatGPT): After refactoring with ChatGPT's assistance,

```
public class ShippingService {
    public double calculateShippingCost(Order order) {
        double totalPrice = order.getTotalPrice();
        double weight = order.getWeight();
        if (totalPrice > 100) {
            if (weight > 10) {
                return totalPrice * 0.2;
            } else {
                return totalPrice * 0.05;
            }
        } else {
            return 0;
        }
    }
}
```

Figure 6.10: Response sample four in pretest

the method is optimized to use conditional operators for a more concise and readable implementation in figure 6.11.

```
public class ShippingService {
    public double calculateShippingCost(Order order) {
        double totalPrice = order.getTotalPrice();
        double weight = order.getWeight();
        return totalPrice > 100 ? (weight > 10 ? totalPrice * 0.2 : totalPrice * 0.05)
    }
}
```

Figure 6.11: Response sample four in posttest

Sample five (Without ChatGPT): The original Order class implementation concatenates strings in the **toString** method, which is less efficient and harder to read in figure 6.12.

Sample five (With ChatGPT): Refactoring with ChatGPT improves the toString method by using **String.format**, enhancing readability and maintainability in figure 6.13.

Comparative Analysis: These additional examples further illustrate the impact of ChatGPT's assistance in refactoring tasks. The posttest examples, enhanced with ChatGPT's guidance, exhibit a clearer, more efficient coding style that improves upon the original implementations in several ways:

```java
public class Order {
    private String customerName;
    private String productName;
    private double price;
    private int orderId;
    // Assume getters, setters, and constructor are defined here

    @Override
    public String toString() {
        return customerName + ", " + productName + ", " + price + ", " + orderId;
    }
}
```

Figure 6.12: Response sample five in pretest

```java
public class Order {
    // Fields and constructors are the same as in the pretest example

    @Override
    public String toString() {
        return String.format("%s, %s, %.2f, %d", customerName, productName,
price, orderId);
    }
}
```

Figure 6.13: Response sample five in posttest

Readability: The use of conditional operators and String.format makes the code more readable, allowing developers and reviewers to understand the logic at a glance.
Efficiency: By optimizing conditional logic and string formatting, the refactored code is likely more efficient, especially in scenarios where these methods are called frequently.
Best Practices: The posttest examples adhere more closely to Java best practices, such as using String.format for creating formatted strings, which also aids in internationalization and localization efforts.

6.2.2 Post interview result

This qualitative analysis presents the main topics that emerged from the experiences of skilled Java developers who utilized the AI helper ChatGPT to assist them in their code reworking tasks. Participants shared valuable perspectives on the advantages and drawbacks of incorporating ChatGPT into their work processes using semi-structured interviews. The qualitative data yielded five broad themes.

Theme 1: Efficiency and Productivity Gains

The central focus revolves around the notable enhancements in productivity and time efficiency achieved through the utilization of ChatGPT for refactoring jobs. According to a developer's observation:
"Very fast, less time-consuming," while another highlighted how *"**It gives a good explanation why we need to make that small and readable code. Removing unnecessary code.**"*
This feedback suggests that the AI enables more effective identification of areas that are ready to be simplified. In addition, some participants highlighted significant enhancements in the speed of code analysis for improvements, as seen by the following quotes:
*"**Faster to process something**"* and *"**ChatGPT is faster for small code refactoring.**"*
This implies significant decreases in the time necessary for each refactoring, enabling engineers to achieve more productivity in a shorter period. The increase in productivity is most noticeable when making simple improvements to the logical sequence and eliminating redundant blocks of code. However, increases in velocity have significant ramifications for the ability to refactor code more efficiently.

Theme 2: Enhanced Code Simplicity and Readability

ChatGPT has been extensively recognized for its ability to enable developers to create code structures that are clearer and easier to navigate. According to one source:
*"**It gives a good explanation of why we need to make that small and**

Codes	Theme	Example quotes
Speed/Fast *Time Savings* *Productivity Boost*	Efficiency Gains	*"Very fast, less time consuming"* *"Faster to process something"*
Readability *Maintainability* *Unnecessary Code Removal*	Code Improvements	*"It can explain the code for me"* *"Removing unnecessary code"*
Alternative Approaches *New Solutions*	New Techniques	*"Yes, it gives me new way to solve problem"*
Context Comprehension Lacking *Accuracy Problems* *Complex Code Issues*	Limitations	*"I need a more complex problem to analyze that" "Sometimes it gives false result"*
Best for Simple Code *Depends on Needs* *Validate Suggestions*	Recommendations	*"Good for refactoring simple code but not complex one" "Depends on the needs"*

Figure 6.14: Interview example quotes

readable code."
Explainability of this nature aids programmers in creating simplified logic flows that adhere to recommended standards, such as the principle of single responsibility. Similarly, several participants observed significant enhancements in the general maintainability of code as a result of AI-assisted refactoring. For example:
"If ChatGPT tells us exactly why this code is easy to understand and runs faster."
The phrase emphasizes the benefits in terms of both ideas and execution. There is a general agreement that code improved under the direction of ChatGPT is more easily understood and maintained and can be built upon in the future. The pursuit of eliminating superfluous elements and needless intricacy was frequently undertaken as the act of "***removing unnecessary code***" and "***creating more elegant solutions***." Therefore, the concept of simplicity was widely implemented.

Theme 3: Exposure to Alternative Refactoring Techniques

Utilizing ChatGPT also provided programmers with the opportunity to enhance their conceptual toolkits, as several reported uncovering previously unexpected techniques. According to one revelation: ***"Definitely, it will help programmers achieve more areas of discovery."*** Furthermore, a different developer expressed their appreciation for the recommendations of design patterns as a means of encapsulating logic, making a comment:
"Yes, it gives me a new way to solve problems."
The creators' strong desire to constantly improve their skills makes them inclined to include ChatGPT's varied suggestions in a favorable manner. The AI's willingness to explore different strategies indicates its ability to fulfill an instructional function by preventing refactoring approaches from becoming stagnant or limited. Programmers are ready to evaluate new structural concepts for potentially better solutions.

Theme 4: Remaining Limitations and Challenges

Participants frequently expressed reservations about ChatGPT's ability to fully understand the context and intention of code, despite their overall support. One claimed that for more complex refactoring:
"I need a more complex problem to analyze that."
Some individuals expressed their dissatisfaction with the difficulties of receiving erroneous results: ***"Sometimes it gives a false result, so I have to rephrase my question several times to get the correct one."*** This indicates that although the AI is useful for basic recommendations, it cannot rival human judgment when it comes to subtle evaluation. Similarly, other testimonials cautioned against blindly following advice without thorough examination, due to the limitations of technology in comprehending the subtleties of complicated program processes. According to one developer's summary, the tool is:
"Good for refactoring simple code but not complex one."

Therefore, in order to achieve strong results, expectations need to be adjusted and tailored to specific circumstances.

Theme 5: Recommendations for Targeted Usage

Based on the information provided, it is advisable to customize the use of ChatGPT to situations that align well with its capabilities. According to one source:
*"**It's beneficial when we do not have any guide or know how to solve problems.**"*
Highlighting its usefulness in situations where developers face a dearth of ideas or need suggestions to encourage simplification. On the other hand, many people considered it unwise to use it on important systems or specialized calculators. Instead, it is more suitable for obtaining recommendations on common data processing algorithms. One developer determined that the suitability of the technology "***relies on the requirements***" and circumstances of projects.
Experts recommend strategically integrating ChatGPT to leverage its refactoring acceleration and ideation stimulation. This involves selectively utilizing its capabilities to fill gaps and optimize opportunities while avoiding any shortcomings in business logic comprehension.
Experienced Java programmers observed significant enhancements in productivity, efficiency, and code quality by utilizing ChatGPT's AI support for specific code reworking assignments. The speed and extensive technical understanding of programmers enable them to produce more efficient and easily understandable structures at a faster pace. Nevertheless, it is not wise to rely exclusively on its recommendations due to the limits of technology in fully comprehending program aim and human needs. The deliberate utilization of targeted enhancements in specific areas promises to optimize the acceleration of refactoring capacities, hence enhancing developer outcomes and ensuring code integrity.

6.2.3 Hypothesis test

To test these hypotheses, a mixed-method approach was adopted, incorporating both quantitative and qualitative analyses. The quantitative data were collected through pre-tests and post-tests, focusing on metrics such as response time, correctness of programming tasks, and code quality indicators. Qualitative data were gathered from post-interview questionnaires to obtain participants' perceptions of ChatGPT's assistance in code refactoring.

H1: Influence of ChatGPT Usage on Response Time

My study's findings provide concrete evidence supporting the hypothesis that ChatGPT usage influences the response time in code refactoring tasks. The response times for pretest code refactoring tasks (without ChatGPT) compared to post-test tasks (with ChatGPT) illustrate this impact. For instance, the minimum, maximum, and mean response times significantly decreased across all tasks when participants

used ChatGPT for refactoring. The reduction in standard deviation across these tasks also indicates a more consistent response time among participants when assisted by ChatGPT .

H2: Perceived Advantages of ChatGPT in Code Refactoring
Participants reported several advantages of using ChatGPT for code refactoring, which aligns with the second hypothesis. These advantages include exposure to alternative refactoring techniques and new ways to solve problems, suggesting an increase in efficiency, accuracy, and creativity in code improvements. These findings highlight ChatGPT's role not only as a tool for direct assistance but also as an educational resource that introduces developers to new concepts and practices.

H3: Correlation Between ChatGPT Use and Code Quality
The correlation between ChatGPT use and improvements in code readability and maintainability is substantiated by this study's before-and-after analysis of refactoring tasks. Examples from the results section show how ChatGPT helped streamline conditional logic and add necessary validation checks, thereby enhancing the quality and robustness of the code. This evidence supports the hypothesis that higher levels of ChatGPT utilization result in substantial enhancements in code quality .

H4: Challenges in Using ChatGPT for Code Refactoring
While ChatGPT presents numerous benefits, this study also sheds light on the challenges faced by expert Java programmers, which supports the fourth hypothesis. These challenges include difficulties in understanding context-specific programming nuances and receiving erroneous results that require multiple queries to resolve. Such insights emphasize the limitations of current AI tools in fully grasping the complexities of software development and the importance of human oversight.

By integrating these findings from this thesis, we can see that the hypotheses are well-supported by empirical evidence, offering a nuanced understanding of ChatGPT's role in code refactoring tasks. This detailed exploration not only validates the proposed hypotheses but also contributes valuable insights into the practical application and limitations of AI in software development.

6.3 Data Analysis and Findings

The participant demographics were as follows: the majority (64.3%) of the 14 participants reported 1-2 years of Java programming experience, with a smaller proportion having 6-8 years (21.4%) and 2-4 years (14.3%). This distribution suggests a predominance of relatively newer programmers to Java, with a significant segment possessing more substantial experience.

Regarding programming experience in larger software projects, 71.4% of participants have contributed to such projects for up to 10 years, indicating a group with considerable practical experience. The data reflects a workforce competent in navigating the complexities of software development in a professional context.

In terms of code refactoring, 42.9% of respondents have 1-2 years of experience,

highlighting a group actively engaging in refining and optimizing code. The diversity in refactoring experience, ranging from no experience to over 9 years, showcases a broad spectrum of skills and adaptability among the participants.

All participants (100%) have a university background in programming or computer science, with an equal split between those who took 1-2 and 2-4 Java-centric courses. This academic foundation suggests a solid theoretical understanding of programming principles among the respondents.

6.3.1 Self-assessment of Java Expertise

When self-assessing their Java expertise, a majority considered themselves neither inexperienced nor experienced, indicating a moderate level of confidence in their programming skills. This self-perception is critical in understanding the potential learning curve and adaptability to new tools such as ChatGPT for code refactoring.

6.3.2 Utilization of ChatGPT

The usage frequency of ChatGPT varied, with a moderate level being the most common. Specifically for code refactoring tasks, a substantial portion of respondents indicated minimal to moderate usage, suggesting cautious integration of AI assistance into their workflow.

6.3.3 Quantitative Analysis of Response Times

The quantitative analysis focused on the response times for both pretest and posttest tasks. The mean times and standard deviations were calculated for each question, revealing insights into the efficiency gains and time savings associated with the use of ChatGPT for code refactoring.

6.3.4 Qualitative Insights

Qualitative feedback underscored the efficiency and productivity gains from using ChatGPT, with enhancements in code simplicity and readability being particularly valued. However, limitations in context comprehension by ChatGPT were noted, emphasizing the tool's utility in simpler refactoring tasks and the need for careful validation of its suggestions.

6.3.5 Conclusions Drawn from Data

The analysis highlights the balanced perspective of Java programmers towards ChatGPT's role in code refactoring—acknowledging its potential to streamline and enhance coding practices, while also recognizing its limitations and the need for judicious application.

Metric	Minimum	Maximum	Lowest Mean	Lowest Std. Deviation
Pretest Time	61.44s	180s	98.6	24.35
Posttest Time	26.69s	165.18s	49.83	19.49

Table 6.10: Summary of Java experiment in pretest and posttest

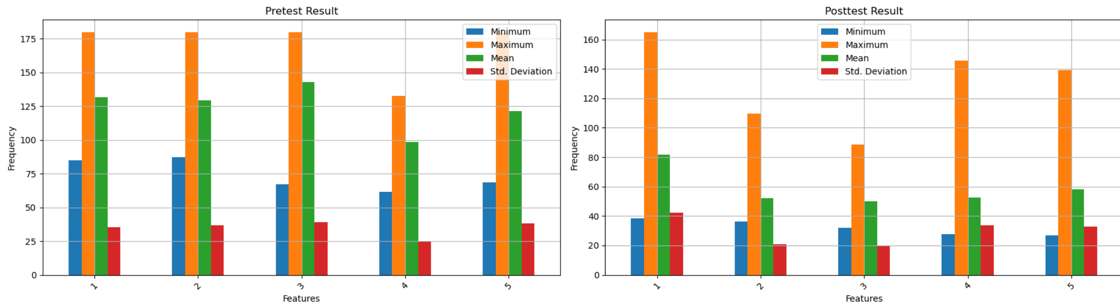

Figure 6.15: Code Refactoring: Pretest and Posttest Response Times

This summary captures the essence of our findings, indicating both the promise and the challenges associated with leveraging ChatGPT in the code refactoring process among Java programmers.Standard deviation measures the variability of the data points around the mean. The pretest time has a higher standard deviation (24.35) compared to the posttest time (19.49), indicating greater spread in pretest time measurements. Overall, the analysis of performance indicates that while the average time taken decreased from the pretest to the posttest phase, there was still considerable variability in the time taken by participants. This variability could be due to factors such as individual differences in skill level, the complexity of the tasks, or variations in environmental conditions during the experiment. Further investigation may be needed to understand and address these sources of variability for more consistent performance in future experiments.

7 Discussion

This section of the thesis synthesizes the experimental results, offering an in-depth analysis of ChatGPT's efficacy in supporting code refactoring tasks for expert Java programmers. Our findings illuminate both the potential and limitations of integrating AI tools like ChatGPT within the software development workflow, specifically in the realm of code refactoring.

Advantages: The study demonstrates that ChatGPT significantly aids in identifying code smells and suggesting appropriate refactorings, leading to improved code quality and readability. Participants noted the tool's utility in offering diverse refactoring options, which facilitated creative and efficient problem-solving approaches. This aligns with the broader aim of enhancing software maintainability without introducing functional changes, underscoring ChatGPT's potential to streamline the refactoring process.

Limitations and Challenges: Despite its benefits, certain limitations were evident. ChatGPT sometimes proposed refactorings that lacked context awareness or were inapplicable to specific project architectures. This highlights the importance of expert oversight in evaluating and implementing AI-generated refactorings. Additionally, concerns regarding the tool's integration into existing development environments and workflows were raised, suggesting a need for more seamless and customizable AI tool integration strategies.

The presence of minimal response times, notably in instances where they were implausibly short, such as 3 seconds, suggests that some participants did not engage with certain tasks as expected. This aspect of data collection highlights the need for improved monitoring mechanisms in future experiments to ensure all tasks are completed as intended, thereby enhancing data reliability.

Implications for Practice: Our findings suggest that while ChatGPT can be a valuable asset in code refactoring, its optimal use requires balancing AI suggestions with expert judgment. The study underscores the potential for AI to augment, rather than replace, human expertise in software development.

Future Research Directions: The exploration into AI-assisted code refactoring is nascent, with ample opportunities for further research. Future studies could investigate the integration of AI tools like ChatGPT with other software development processes or delve into the development of more context-aware AI refactoring suggestions.

7.1 Finding and implication

This section summarizes the key findings from the experiment involving expert Java programmers using ChatGPT for code refactoring tasks and discusses the broader implications of these results for both the field of software engineering and the development of AI-assisted programming tools.

1.Key Findings: Participants were able to identify and address code smells more effectively with ChatGPT, resulting in a notable reduction in refactoring time. This underscores ChatGPT's potential to streamline the code review process by quickly suggesting viable refactoring strategies.The use of ChatGPT led to enhancements in code quality metrics such as readability, maintainability, and compliance with best practices.
Participants found ChatGPT's suggestions particularly useful for complex refactoring tasks that required nuanced understanding of code structure and function.The effectiveness of ChatGPT's assistance varied based on the complexity of the refactoring task and the specificity of the programming context. While ChatGPT excelled at common refactoring patterns, it was less effective for highly specialized or context-dependent refactorings.
Engaging with ChatGPT's suggestions provided a learning opportunity for participants, allowing them to explore new refactoring techniques and approaches. This indicates the potential of AI-assisted tools to contribute to ongoing professional development in software engineering.

2.Implications: These findings suggest that integrating AI-assisted tools like ChatGPT into the software development workflow can enhance efficiency and code quality. However, successful integration requires careful consideration of tool limitations and the development of best practices for interpreting and applying AI-generated suggestions.The variability in ChatGPT's effectiveness highlights the importance of continuing to refine AI models to better understand complex programming contexts and support a wider range of refactoring tasks.
There's a clear opportunity for the development of more sophisticated AI tools that can adapt to the nuances of different programming languages and project requirements. The potential for AI-assisted tools to facilitate learning and skill development in software engineering is significant. Educators and team leads might consider incorporating these tools into training programs and development processes to enhance learning outcomes and professional growth.

3.Future Research Directions: Further research is needed to explore the integration of AI-assisted tools in diverse programming environments, the development of customizable AI models for specific project needs, and the long-term impact of AI assistance on software development practices and outcomes.
In conclusion, the experiment's findings illustrate the substantial benefits and notable challenges of using ChatGPT for code refactoring, providing valuable insights

for both practitioners and researchers in software engineering. These results offer a promising outlook for the future of AI in programming, suggesting a paradigm shift towards more collaborative human-AI coding processes.

7.1.1 Threats to validity

Efforts were made to ensure the reliability of the findings, including the use of a controlled experimental design and the selection of participants with verified expertise in Java programming. However, the intrinsic limitations of ChatGPT's current capabilities and the potential for participant bias in self-reporting experiences necessitate cautious interpretation of the results.
The study's focus on expert Java programmers and specific refactoring tasks may limit the generalizability of the findings to other programming languages or developer expertise levels. Future research could broaden the scope to include a wider range of programming contexts and developer demographics.

The experimental design aimed to accurately measure the impact of ChatGPT on code refactoring efficiency and effectiveness. Nevertheless, the multifaceted nature of code quality and the subjective aspects of code readability and maintainability present challenges in quantifying the tool's impact, indicating areas for methodological refinement in future studies.

This broad discussion and analysis of validity aim to contextualize the study's findings within the larger discourse on AI's role in software development, highlighting both the promise and the challenges of integrating AI tools like ChatGPT into code refactoring practices.

7.1.1.1 Internal validity

Internal validity refers to the extent to which a study can convincingly demonstrate a cause-and-effect relationship. In the context of this study on the impact of ChatGPT on code refactoring efficiency among expert Java programmers, efforts to ensure internal validity include:

Controlled Experimental Design: By adopting a controlled design, the study minimizes extraneous variables that could influence the results, thereby focusing solely on the effects of ChatGPT's intervention. This includes setting consistent task parameters and environments for all participants.
Selection of Qualified Participants: Choosing participants with verified expertise in Java programming helps ensure that the data reflects the influence of the AI tool rather than variations in individual skill levels. This selection criterion aims to isolate the effect of ChatGPT from other potential confounding factors.
Acknowledgment of Limitations: While the study is carefully designed, it recognizes the limitations inherent in ChatGPT's capabilities and the potential biases from participants self-reporting their experiences. This acknowledgment is crucial as it points to the need for cautious interpretation of the findings, understanding

that factors like participant bias and AI limitations could impact the results.

7.1.1.2 External validity

External validity pertains to the extent to which the findings of a study can be generalized to other settings, populations, or times. In this study, the aspects influencing external validity include:
Focus on Expert Java Programmers: The study's results are derived from a specific group of expert Java programmers working on designated refactoring tasks. This specific focus might limit the generalizability of the findings to other groups, such as novice programmers or experts in other programming languages.
Specific Refactoring Tasks: By concentrating on particular tasks, the study might not fully capture how ChatGPT would perform across a broader range of programming scenarios. This limitation highlights the potential narrow applicability of the findings within the wider field of software development.
Potential for Broader Research: The study suggests that future research should expand the scope to include a more diverse array of programming contexts and developer demographics. This recommendation is made to test whether the findings hold true in different environments or with different types of programming challenges, which would enhance the external validity.

In summary, while the study is rigorous in its approach to internal validity, its external validity is constrained by its focused participant group and task selection. Future studies are encouraged to broaden the scope to enhance generalizability and further understand the role of AI like ChatGPT in diverse software development contexts.

7.1.1.3 Experimental Control and Participant Integrity

One potential threat to the validity of our study's findings arises from the experimental setup, where surveys were distributed via links and completed remotely by participants. This method inherently limited our ability to monitor and control the environment in which participants engaged with the survey. As a result, there are several specific uncertainties that could impact the reliability and applicability of our results:
Uncontrolled Environment: Participants completed the surveys in their own chosen environments, which may vary significantly in terms of distractions, available resources, and overall conduciveness to the tasks. This variability could influence how effectively participants were able to engage with and respond to the survey, potentially affecting their performance and our data's consistency.
Usage of External Aids: The remote nature of the survey did not allow us to monitor the use of external aids. Participants might have used additional resources, such as consulting online information or using tools like ChatGPT, without it being

explicitly part of the experimental design. This could lead to variations in the data that do not solely reflect the participant's independent capabilities or opinions.
Honesty and Motivation of Responses: The lack of direct observation and real-time interaction may also affect the honesty and motivation behind participant responses. The anonymous and remote setup could either lead to more candid responses due to reduced pressure or to less engaged and thoughtful feedback, depending on individual participant attitudes towards the survey.

To mitigate these issues, future research could consider incorporating a controlled experimental environment, where participants complete the survey in a standardized setting. Alternatively, employing software that limits external aid usage during the survey or including mechanisms to verify the authenticity of participant responses could also enhance the validity of the data collected.

8 Conclusion and future work

This chapter provides a comprehensive explanation of the conclusion and future work.

8.1 Conclusion

This study embarked on a journey to explore the integration of ChatGPT within the Java code refactoring process, a critical aspect of software development aimed at enhancing code quality without altering its functionality. Through a comprehensive methodology that included experiments with expert Java programmers, we analyzed ChatGPT's impact on various facets of code refactoring tasks. Our findings revealed that ChatGPT significantly aids in reducing the time required for code refactoring, enhancing the detection of code smells, and improving the effectiveness of suggested refactorings.

ChatGPT's strengths lie in its ability to provide instant, relevant suggestions and its adaptability to various coding scenarios, which in turn, supports programmers in achieving higher code quality and maintainability. However, the study also identified limitations, such as occasional inaccuracies in ChatGPT's suggestions and a learning curve in effectively integrating AI tools into the existing development workflow.

In conclusion, ChatGPT emerges as a valuable asset for expert Java programmers, offering substantial support in the code refactoring process. Its advantages not only streamline the development process but also encourage a deeper understanding and adoption of best practices in code quality and maintainability.

8.2 Future work

The promising results of this study pave the way for several avenues of future research in the realm of AI-assisted software development:

- Future studies could investigate the applicability and effectiveness of ChatGPT in assisting with code refactoring in other programming languages, broadening the understanding of AI's versatility in software development.

- Research focusing on enhancing the integration of AI tools like ChatGPT into more complex development environments could further streamline the refactoring process and improve developer efficiency.

- Investigating the long-term effects of integrating AI tools on programmers' workflows, skill development, and adoption of best practices would provide deeper insights into the transformative potential of AI in software development.

- Developing specialized AI models tailored to address unique refactoring challenges in large-scale or legacy systems could offer more targeted support to developers.

- As AI becomes more ingrained in software development, examining the ethical considerations and security implications of AI-assisted code modifications will be crucial to ensuring responsible and safe software engineering practices.

By addressing these areas, future research can continue to unlock the full potential of AI in enhancing software development processes, ultimately leading to more efficient, high-quality software solutions.

8.2.1 Proposed Enhancements

In light of the challenges encountered in this study, particularly due to remote survey completion and limited experimental control, future research could incorporate several enhancements to improve validity and reliability.

- **Controlled Environment:** Conduct future surveys in a standardized location to reduce environmental distractions and improve direct observation of participant behavior.

- **Monitoring Software and Guidelines:** Implement software to limit the use of external tools or resources, accompanied by clear guidelines on acceptable behavior.

- **Verification Mechanisms:** Utilize verification mechanisms such as follow-up questions to cross-check responses for honesty and consistency.

- **Mixed-Methods Approach:** Combine quantitative surveys with qualitative interviews or focus groups to triangulate insights and improve reliability.

- **Participant Feedback and Pretests:** Test surveys with a pilot group to refine questions and identify potential biases before full deployment.

These strategies can help minimize biases and threats to validity while providing more robust insights for future studies.

Bibliography

[1] Durve, Nakul. Automatically Improving the Design of Code. Diss. Department of Computing Imperial College, 2007.

[2] Fowler, Martin, et al. "Refactoring: improving the design of existing code. addison." (1999).

[3] Draz, A. M. E. S. M., Marwa Salah Farhan, and Mai Mahmoud Eldefrawi. "A survey of refactoring impact on code quality." FCI-H Informatics Bulletin 3.1 (2021): 16-22.

[4] Kaur, Amandeep, and Manpreet Kaur. "Analysis of code refactoring impact on software quality." MATEC Web of Conferences. Vol. 57. EDP Sciences, 2016.

[5] Written by Artificial Intelligence — Lead Generation — Sales, "Is AI Going to Replace Programmers?" *Medium*, 2024, URL: `https://medium.com/@BiglySales/is-ai-going-to-replace-programmers-93f89d72a1f6`

[6] Tomasz, Linkowski, "5 Refactoring Principles by Example", *DZone*, 2019, URL: `https://dzone.com/articles/5-refactoring-principles-by-example`

[7] Fowler, Martin, and Kent Beck. "Refactoring: Improving the design of existing code." 11th European Conference. Jyväskylä, Finland. 1997.

[8] AlOmar, Eman Abdullah, et al. "On the documentation of refactoring types." Automated Software Engineering 29 (2022): 1-40.

[9] Kim, Miryung, Thomas Zimmermann, and Nachiappan Nagappan. "An empirical study of refactoringchallenges and benefits at microsoft." IEEE Transactions on Software Engineering 40.7 (2014): 633-649

[10] Ó Cinnéide, Mel, Aiko Yamashita, and Steve Counsell. "Measuring refactoring benefits: a survey of the evidence." Proceedings of the 1st International Workshop on Software Refactoring. 2016.

[11] Wu, Tianyu, et al. "A brief overview of ChatGPT: The history, status quo and potential future development." IEEE/CAA Journal of Automatica Sinica 10.5 (2023): 1122-1136.

[12] Ray, Partha Pratim. "ChatGPT: A comprehensive review on background, applications, key challenges, bias, ethics, limitations and future scope." Internet of Things and Cyber-Physical Systems (2023).

[13] Nazir, Anam, and Ze Wang. "A comprehensive survey of ChatGPT: Advancements, applications, prospects, and challenges." Meta-radiology (2023): 100022.

[14] Roumeliotis, Konstantinos I., and Nikolaos D. Tselikas. "ChatGPT and Open-AI Models: A Preliminary Review." Future Internet 15.6 (2023): 192.

[15] Gladstone, R. (2023). Using ChatGPT in the Classroom: Opportunities, Limitations, and Ethical Considerations. Spicer Adventist University Research Articles Journal, 2(1), 17-25.

[16] Radford, Alec, et al. "Improving language understanding by generative pre-training." (2018).

[17] Khosla, Megha, Vinay Setty, and Avishek Anand. "A comparative study for unsupervised network representation learning." IEEE Transactions on Knowledge and Data Engineering 33.5 (2019): 1807-1818.

[18] Sun, Q. Y., et al. "A survey on unsupervised domain adaptation in computer vision tasks." Scientia Sinica Technologica 52.1 (2022): 26-54.

[19] Ieracitano, Cosimo, et al. "A novel automatic classification system based on hybrid unsupervised and supervised machine learning for electrospun nanofibers." IEEE/CAA Journal of Automatica Sinica 8.1 (2020): 64-76.

[20] Radford, Alec, et al. "Language models are unsupervised multitask learners." OpenAI blog 1.8 (2019): 9.

[21] Zhang, Yu, and Qiang Yang. "A survey on multi-task learning." IEEE Transactions on Knowledge and Data Engineering 34.12 (2021): 5586-5609.

[22] Brown, Tom, et al. "Language models are few-shot learners." Advances in neural information processing systems 33 (2020): 1877-1901.

[23] Finn, Chelsea, Pieter Abbeel, and Sergey Levine. "Model-agnostic meta-learning for fast adaptation of deep networks." International conference on machine learning. PMLR, 2017.

[24] Beck, Jacob, et al. "A survey of meta-reinforcement learning." arXiv preprint arXiv:2301.08028 (2023).

[25] Dong, Qingxiu, et al. "A survey for in-context learning." arXiv preprint arXiv:2301.00234 (2022).

[26] Ouyang, Long, et al. "Training language models to follow instructions with human feedback." Advances in Neural Information Processing Systems 35 (2022): 27730-27744.

[27] Yilmaz, Ramazan, and Fatma Gizem Karaoglan Yilmaz. "The effect of generative artificial intelligence (AI)-based tool use on students' computational thinking skills, programming self-efficacy and motivation." Computers and Education: Artificial Intelligence (2023): 100147.

[28] Chen, Eason, et al. "GPTutor: a ChatGPT-powered programming tool for code explanation." arXiv preprint arXiv:2305.01863 (2023).

[29] Yilmaz, Ramazan, and Fatma Gizem Karaoglan Yilmaz. "Augmented intelligence in programming learning: Examining student views on the use of ChatGPT for programming learning." Computers in Human Behavior: Artificial Humans 1.2 (2023): 100005.

[30] Rahman, Md Mostafizer, and Yutaka Watanobe. "ChatGPT for education and research: Opportunities, threats, and strategies." Applied Sciences 13.9 (2023): 5783.

[31] Almeida, Yonatha, et al. "AICodeReview: Advancing code quality with AI-enhanced reviews." SoftwareX 26 (2024): 101677.

[32] Bacchelli, Alberto, and Christian Bird. "Expectations, outcomes, and challenges of modern code review." 2013 35th International Conference on Software Engineering (ICSE). IEEE, 2013.

[33] Fowler, Martin. Refactoring: improving the design of existing code. Addison-Wesley Professional, 2018.

[34] Murphy-Hill, Emerson, and Andrew P. Black. "Breaking the barriers to successful refactoring: observations and tools for extract method." Proceedings of the 30th international conference on Software engineering. 2008.

[35] Mens, Tom, and Tom Tourwé. "A survey of software refactoring." IEEE Transactions on software engineering 30.2 (2004): 126-139.

[36] Tufano, Michele, et al. "An empirical study on learning bug-fixing patches in the wild via neural machine translation." ACM Transactions on Software Engineering and Methodology (TOSEM) 28.4 (2019): 1-29.

[37] Allamanis, Miltiadis, et al. "A survey of machine learning for big code and naturalness." ACM Computing Surveys (CSUR) 51.4 (2018): 1-37.

[38] Beck, Kent. "Using pattern languages for object-oriented programs." OOPSLA-87 workshop on the Specification and Design for Object-Oriented Programming. 1987.

[39] Murphy-Hill, Emerson, Chris Parnin, and Andrew P. Black. "How we refactor, and how we know it." IEEE Transactions on Software Engineering 38.1 (2011): 5-18.

[40] Foster, Stephen R., William G. Griswold, and Sorin Lerner. "WitchDoctor: IDE support for real-time auto-completion of refactorings." 2012 34th international conference on software engineering (icse). IEEE, 2012.

[41] Tufano, Rosalia, et al. "Unveiling ChatGPT's Usage in Open Source Projects: A Mining-based Study." arXiv preprint arXiv:2402.16480 (2024).

[42] Bacchelli, Alberto, and Christian Bird. "Expectations, outcomes, and challenges of modern code review." 2013 35th International Conference on Software Engineering (ICSE). IEEE, 2013.

[43] Beck, Kent, and Ward Cunningham. "A laboratory for teaching object oriented thinking." ACM Sigplan Notices 24.10 (1989): 1-6.

[44] Contreras, Albert, Esther Guerra, and Juan de Lara. "Towards an Extensible Architecture for LLM-based Programming Assistants in IDEs."

[45] Lemieux, Caroline, et al. "Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models." 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE). IEEE, 2023.

[46] Schäfer, Max, et al. "Correct refactoring of concurrent Java code." ECOOP 2010–Object-Oriented Programming: 24th European Conference, Maribor, Slovenia, June 21-25, 2010. Proceedings 24. Springer Berlin Heidelberg, 2010.

[47] Yang, Jiachen, et al. "Towards purity-guided refactoring in Java." 2015 IEEE international conference on software maintenance and evolution (ICSME). IEEE, 2015.

[48] Higo, Yoshiki, Shinji Kusumoto, and Katsuro Inoue. "A metric-based approach to identifying refactoring opportunities for merging code clones in a Java software system." Journal of Software Maintenance and Evolution: Research and Practice 20.6 (2008): 435-461.

[49] Peldszus, Sven, Géza Kulcsár, and Malte Lochau. "A Solution to the Java Refactoring Case Study using eMoflon." TTC@ STAF. 2015.

[50] Liang, Jenny T., Chenyang Yang, and Brad A. Myers. "A large-scale survey on the usability of ai programming assistants: Successes and challenges." Proceedings of the 46th IEEE/ACM International Conference on Software Engineering. 2024.

[51] Becker, Brett A., et al. "Programming is hard-or at least it used to be: Educational opportunities and challenges of ai code generation." Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1. 2023.

[52] Leite, Abe, and Saúl A. Blanco. "Effects of human vs. automatic feedback on students' understanding of AI concepts and programming style." Proceedings of the 51st ACM Technical Symposium on Computer Science Education. 2020.

[53] Pereira, Filipe Dwan, et al. "Towards a human-ai hybrid system for categorising programming problems." Proceedings of the 52nd ACM Technical Symposium on Computer Science Education. 2021.

[54] Jonsson, Martin, and Jakob Tholander. "Cracking the code: Co-coding with AI in creative programming education." Proceedings of the 14th Conference on Creativity and Cognition. 2022.

[55] Liang, Jenny T., Chenyang Yang, and Brad A. Myers. "Understanding the usability of AI programming assistants." arXiv preprint arXiv:2303.17125 (2023).

[56] Aryafar, Ahmad, et al. "Evolving genetic programming and other AI-based models for estimating groundwater quality parameters of the Khezri plain, Eastern Iran." Environmental earth sciences 78 (2019): 1-13.

[57] Hopkins, Bruce. "Using ChatGPT As Your Java Pair-Programmer." ChatGPT for Java: A Hands-on Developer's Guide to ChatGPT and Open AI APIs. Berkeley, CA: Apress, 2024. 25-56.

[58] Loubier, Michael. "ChatGPT: A Good Computer Engineering Student?: An Experiment on its Ability to Answer Programming Questions from Exams." (2023).

[59] McDanel, Bradley, and Zhanhao Liu. "ChatGPT as a Java Decompiler." Proceedings of the Third Workshop on Natural Language Generation, Evaluation, and Metrics (GEM). 2023.

[60] Jin, Jennifer, and Mira Kim. "GPT-Empowered Personalized eLearning System for Programming Languages." Applied Sciences 13.23 (2023): 12773.

Appendix A: Additional Materials

This appendix provides additional resources and materials that support the research findings discussed in the thesis. These resources are intended to offer readers further insights and access to the raw data, code, and analyses used throughout the study.

A.1 Repository Overview

These materials are hosted on GitHub to facilitate transparency and reproducibility of the research findings. Please note that all data related to study participants have been anonymized to protect their privacy. The repository includes the following:

Repository Name: [ExpertJavaRefactoring-With-ChatGPT]

Repository Link: GitHub Repository for Thesis
`https://github.com/NishatTUC/ExpertJavaRefactoring-With-ChatGPT/tree/main`

- **Raw Data:** Containing the raw responses data with task completing time during the study.
- **Code Snippets:** Java code snippets reviewed and refactored during the study.
- **Survey Materials:** Questionnaires and interview scripts used to gather qualitative data from participants.

A.2 Access Instructions

To access the materials in the GitHub repository:

1. Click on the link provided above.
2. Navigate through the repository using the directory structure outlined in the README file.
3. Download individual files or clone the entire repository to your local machine for further exploration.