Enhancing Code Refactoring in Python: Leveraging Large Language Models

Master’s Thesis in Computer science and engineering

ARPITA NAIK & RITHIKA RAJAN SHYLAJA

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2024

Master's Thesis 2024

Enhancing Code Refactoring in Python: Leveraging Large Language Models

ARPITA NAIK & RITHIKA RAJAN SHYLAJA

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2024

Enhancing Code Refactoring in Python: Leveraging Large Language Models
ARPITA NAIK & RITHIKA RAJAN SHYLAJA

Supervisor: Farnaz Fotrousi, Department of Computer Science and Engineering
Examiner: Hans-Martin Heyn, Department of Computer Science and Engineering

Master's Thesis 2024
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Gothenburg, Sweden 2024

Enhancing code refactoring in Python: Leveraging Large Level Modules
ARPITA NAIK & RITHIKA RAJAN SHYLAJA
Department of Computer Science and Engineering,
Chalmers University of Technology

Abstract

Software engineering is evolving rapidly, involving tasks in creating new reliable features. The challenges of maintaining the existing features are challenging due to the existence of code smells, resulting in technical debt and thus reducing the overall quality of the product. At the same time, manual refactoring is error-prone and time-consuming. LLMs are becoming increasingly prevalent across various parts of the software development life cycle. The study aims to improve code refactoring suggestions in Python using Large Language Models (LLMs) such as ChatGPT and GitHub Copilot. Code refactoring is an important aspect of software engineering to address code smells, which improves the code quality.

The research aims to investigate the refactoring suggestions provided by LLMs compared to those provided by humans, specifically focusing on Python programming language. To determine this objective, qualitative research methods are incorporated, semi-structural interviews with developers to identify the prioritized and ignored code smells and challenges faced during the refactoring process in the industry.

The result was analyzed using narrative analysis, pre-study on prompt engineering to get an optimized prompt comparing and analyzing code refactoring suggestions between two LLMs, the study also conducts a controlled experiment to compare refactoring suggestions between LLM's and developers, the study identifies common refactoring patterns the ability of LLMs to offer correct and understandable refactoring solutions.

The research further demonstrates that LLMs can assist in giving refactoring suggestions but human developers are better than LLMs when it comes to understanding complex code smells and addressing code structure. On the other hand, LLMs are capable of enhancing code structure consequently enhancing the process of refactoring. The results show the merits and demerits of code refactoring using LLMs and how they can be adopted in Python development code refactoring.

Acknowledgements

We want to express our deep gratitude to the people who made our thesis successful. Our supervisor, Farnaz Fotrousi, deserves our sincere thanks for her valuable guidance, great support, and beneficial feedback during our journey. Her thoughtful feedback and interactive discussions have played essential role in helping us overcome all the difficulties and achieve key progress in this thesis. We extend our deep appreciation to our examiner Hans-Martin Heyn for dedicating his time to provide us with valuable and constructive feedback and for evaluating our thesis.

Additionally, a sincere thank you to the interview participants and experts who gave their time slots even in their busy schedule and giving us most important insights and experiences, greatly benefiting our study.
I, Arpita Naik, am thankful to my husband and friends for their support during my thesis.
I, Rithika Rajan Shylaja, express my gratitude to my husband, kid, brother, and friends for their extensive support and motivation during the thesis.
A special thanks to the Chalmers University for giving us this opportunity to pursue our master's program.

Arpita Naik, Gothenburg, 2024 Rithika Rajan Shylaja, Gothenburg, 2024

Contents

List of Figures

List of Tables

1

Introduction

Software engineering frequently involves the task of creating new reliable and maintainable code in a context of constantly evolving contexts and expanding challenges. The existence of code smells increases these challenges, resulting in technical debt and reducing the overall quality of the code quality [1]. Finding and eliminating code smells are important tasks in the process of developing software that meets high standards. SonarQube, a widely used static code analysis tool, helps identify code smells [2]. Refactoring can eliminate identified code smells. The software engineering community has accepted the concept of refactoring as a methodical way to improve the design and structure of code without changing its external behaviour [1]. Refactoring is carried out either manually or using automated tools in both industrial and academic settings. Research shows that manual refactoring of code smells is time-consuming, error-prone, and costly [1].

Whereas in automated refactoring, developer's enter their code for automated refactoring recommendations, which provides a complete but potentially problematic solution. While tools are crucial for analysing the impact of code smells, a significant drawback emerges, many existing tools struggle to communicate errors resulting from refactoring. Additionally, automatic refactoring tools face challenges in providing custom refactoring solutions to specific user needs due to the unlimited possibilities of refactoring. Consequently, the use of these tools may occasionally have adverse effects on code quality. Despite their role in reducing developer involvement and addressing code smells, refactoring tools can introduce confusion, limit developer oversight, and lack the necessary adaptability [1][3].

In response to these challenges [4] researchers have included the usage of large language models (LLMs), more particularly ChatGPT into the software development process, particularly in the settings of code refactoring in Java Programming language. By providing updated versions of code, LLMs demonstrate their capability to improve maintainability as a quality attribute by understanding developer's different refactoring needs. use specific phrases to communicate refactoring requirements, while LLMs adeptly incorporate these intentions into their responses. During the survey, 43 patterns of refactoring documentation were distinguished: for example, extract*, mov*, and renam* which all indicate the ability of LLMs to understand and provide solutions to generic techniques used in software refactoring which enhances code maintainability. LLMs, and more so ChatGPT, observe both the internals (eg. Inheritance, complexity) and externals (eg. Readability, usability, performance) of quality, indicating awareness of the critical variables of code quality and maintainability. In the example of the developer's interaction with ChatGPT, where the developer provided portions of code in addition to a narrative explanation, LLMs were seen as, actively participating in the process of code refactoring [4].

The goal of this thesis is to explore the approaches of code refactoring by engaging large language models (LLMs) like ChatGPT or Copilot in the course of software development. The LLMs are capable of comprehending the specific instructions of the developer regarding how to change the code and hence providing a revised version of the code that is of increased maintainability and quality [4].

This thesis is focused on understanding and documenting both the code smells that have been priortised and the ones that have been neglected along with the challenges encountered by the Python and seeks to validate the refactored code sugestions done by using LLMs like ChatGPT and Copilot in particular, with regards to the quality of the codes done in the Python programming language. The purpose of the research is to investigate the correctness of LLM in code refactoring suggestions and how to overcome existing refactoring tools limitations [6].

The research methodology involves qualitative research techniques, using semi structured interviews, narrative analysis, carrying out a prompt engineering pre-study and experiments with LLMs, as well as comparing results obtained using LLM and and their detailed analysis. This thesis aims at investigating the factors that influence the adoption of refactoring solutions by practitioners, the most effective prompt patterns for considering the refactoring suggestions from LLMs, and the effectiveness of refactoring suggestions offered by available LLMs [5].

To conclude, the significance of this research lies on the use of LLMs towards enabling better code quality during refactoring activities thereby improving software development processes. Like many other studies, this one aims at addressing research questions in the works related to software engineering, but specifically concentrating on Python and its use cases instead.

1.1 Statement Of the Problem

In contrast to the available literature demonstrating extensive research on refactoring tools within the Java programming language, there is a lack of studies conducted on such tools for the case of Python. Currently, the tools available in Java for automated code refactoring show several issues, such as depersonalizing the process and decreasing the developer's control over refactoring, lack regulations for decision-making, and failing to allow customization for individual users. As a result, there exists a research problem about possibilities for code refactoring in Python where leveraging large language models (LLMs) will be adopted for the enhancement of code quality as well as maintaining the code while making it possible to fix code smells through active participation on the design of solutions. The purpose of this thesis is to improve the standard and performance of coding by making use of LLMs prompt engineering practices for proposed changes to code, and easier exclusion of coding efforts that are not aimed at the primary goals and objectives. In this respect, the thesis presents a systematic review, which in particular will focus on the stages of software development and the evolution of software products where LLMs help programmers and also how prompts can be used for enhancing code quality and maintainability [4][5][7].

1.2 Purpose of the Study

This thesis aims to fill in the important gaps in the current practice by exploring innovative approaches for code refactoring suggestions using LLMs, assessing their correctness and understandability using prompt engineering and compare those to professional Python developer's. The resulting research has the potential to improve code quality, simplify development workflows, and empower the software engineer with advanced tools and insights.

1.3 Research Questions

RQ1: What refactoring patterns do the Python programmers employ while dealing with various code smells?

RQ1.1: Which code smells are prioritized and which ones are ignored during the refactoring?
The objective of this question is to examine the refactoring process and how do they priortize and ignore code smells, keeping themselves focused on the refactoring task.

RQ1.2: What challenges do they encounter in the process of the refactoring?
The purpose of this question is to explore the code quality improvement challenges that developer's encounter in the industry concerning the refactoring process.

RQ2: How correct and understandable are LLM generated refactoring suggestions in addressing code smells?

RQ2.1: How do understandability and correctness of code refactoring suggestions provided by LLMs compare with various LLMs?
The purpose of this question aims to investigate the understandability and correctness in comparision of the refactoring suggestions provided by the two LLMs-Chatgpt4 and Github copilot to resolve code smells.

RQ2.2: How do understandability and correctness of code refactoring suggestions provided by LLMs compare with that of a human expert?
The purpose of this question aim to investigate the understandability and correctness of the refactoring suggestions provided by Large Language Models LLMs versus human experts in resolving code smells.

Note: correctness here refers to the understandability and readability of the code.

Our objective is to measure the quality of refactoring suggestions made by LLMs in a consistent and valid manner. A successful elimination of code smells is a part of this research, with no change in functionality, readability, and code structure.

2

Background and Related Work

2.1 Code smells

Code smells in software development are a very crucial aspect of maintaining code quality to improve readability and better maintainability. By addressing these problems we can avoid technical debt which in turn restricts to modify and maintain the code-base easily. Prioritising these code smells on various factors can impact the developers to concentrate on solving problems that may cause issues on the functionality. Moreover, early handling will help to avoid bugs improve performance, lower bug scores, and lower long term maintenance costs. Finally, organized and clean code leads to reliable software development processes which facilitates smoother collaboration and faster on-boarding of new team members.

Refactoring is crucial to software maintenance since it enhances system architecture and fixes bugs. To find the most popular refactoring techniques, programming languages, and ways to apply them in the software industry, a recent study [6] surveyed software practitioners online. The results showed that Java was the most widely used language (96.7%) and that the Eclipse IDE was the most often used (63.3%) to apply refactoring techniques. These findings provide empirical evidence to improve software development practices. Refactoring, which is regarded as a conventional method for enhancing software design while maintaining functionality, seems to be a solution to this problem. The results provide empirical support for better software development procedures, which helps researchers create reference models and practical tools that practitioners may use to improve the quality of software [6].

The research conducts a comparative study of automated refactoring tools, aiming to address the challenge faced by developers in selecting tools based on their specification. The study mainly evaluates four distinct refactoring tools MultiRefactor, JDeodorant, jSparrow, and Spartenizer belonging to different categories of refactoring approaches. The evaluation framework, based on the Design Evaluation and Selection of Methodologies and Tools methodology DESMET, systematically compares these tools using five open-source Java projects. The findings reveal that jSparrow as the effective tool, supporting a broad range of quantitative and qualitative features. On the other hand, Spartenizer exhibits less favorable outcomes, introducing new code smells after applying refactoring opportunities. This research highlights importance of empirical evaluations in helping developers and researchers in tool selection, providing insights into tool characteristics in Java domain [8].

2.2 Large Language Models (LLMs) and Prompt Engineering

The emergence of Language Models called LLMs has transformed the natural language processing (NLP) field and helped shed light on the impressive functions of understanding and writing human-like text. Analyzing Software Development LLMs like OpenAI's GPT-3 or its more recent versions have been applied to a variety of contexts, e.g. code completion, bug detection, etc. These models are pre-trained on huge amounts of open-sourced data that helps them to understand intricate patterns in natural language, and in turn, create responses that are relevant to what input is given.

Prompt Engineering: One of the key aspects of having a productive usage of LLMs Designing and optimizing prompts that drive specific, clear, and contextually relevant responses from LLMs constitutes prompt engineering. Well-designed prompts quickly realize that the performance of LLMs can be greatly improved by more optimal prompt engineering which can in turn make them incredibly more useful for code refactoring, code smell detection, etc.

Research Contributions and Techniques in Prompt Engineering:

The systematic survey of prompt engineering in LLMs includes prompt-engineering techniques and how they are applied in LLMs from other major sources. This gives us insight into how to design and tune our prompts for better performance of the LLMs across a wider range of tasks. The study groups prompt engineering approaches on a granular level: from prompt design — i.e., treatments that design individual prompts with distinct structures and language — to prompt tuning — i.e., treatments that iteratively refine prompts (usually in a data-driven way) based on the system's responses. Prompt engineering needs to be done earliest for optimal unrealization of LLM possibilities — survey [14].
Prompt Engineering with ChatGPT- This body of research is a written guide about prompt engineering in academic writing of ChatGPT. It contains best practices for crafting and tuning prompts to make the most of ChatGPT across different writing use cases. The study provides a glimpse at strategies for creating prompts that direct ChatGPT to generate accurate and contextually suitable text. In order to make ChatGPT more useful for the academic purposes as well, it highlights the need for timely engineering [15].
By considering LLMs are human-level prompt engineers, we show how LLMs can engage in rapid prototyping at the human level; that is in effect able to read and respond to well-written prompts. Studies detail the ways in which LLMs are trained to create certain results. It shows how LLMs can produce outputs of such a great quality and application matching the prompts, which highlights, how LLMs can be used as a potent asset to a company in a different area [16].
In one of the research called An Evaluation on GitHub Copilot, they analyze the dialogue between prompt engineering and TODO comments, in light of a specific actual context GitHub Copilot. It examines how effectively Copilot responds to prompts

embedded in TODO comments. The paper discusses steps to optimize prompts to enhance the quality of code suggestions generated by Copilot. The article shows both the merits and difficulties of incorporating prompt engineering methods into software development workflows [17].
To understand more about code quality using LLM we referred to Frustrated with Code Quality Issues. LLMs can Help this paper which analyzes the likelihood of LLMs improving the standard of the code by discovering and fixing issues that relate to the standard of the code, such as code smells. This study looks at different ways of making LLMs detect and refactor code smells. It points out practical uses of LLMs in programming, demonstrating how effective prompt engineering helps in generating actionable ideas for the improvement of codes [18].

To conclude, blending advanced prompt manufacturing techniques with LLMs offers great hope of improving code quality and maintainability. Better prompting makes sure that LLMs can sense code smells while offering reliable modification proposals which enhances code readability and makes it easier to maintain it. Different research projects have demonstrated that high-quality and immediate initial prompt design can significantly increase the effectiveness of LLMs across a range of tasks, such as code refactoring. Configuring correct prompts results in improved outputs in terms of accuracy as well as relevance which means that LLMs are very essential for software development and maintenance. As Large Language Models continue to become better, trigger engineering remains a critical region of inquiry and application, pushing for more improved language models. In the end, the incorporation of LLMs into systems that generate programming codes gives software developers the ability to eliminate code smells in good time and more effectively create more durable, high-quality software programs.

3

Methods

In this section, we conducted our research in a structured way by collecting data, and examining our findings. The specified tools and strategies were employed to resolve the research questions. The section contains data collection, data analysis as well as different steps to be undertaken to answer our research questions. This section involves two methodologies: We conducted a systematic qualitative data analysis to address RQ1. Systematic Qualitative Data Analysis uses narrative analysis for finding useful patterns in interview transcripts. Pre-study by doing an exploratory study on creating prompts for LLMs's and Qualitative method as well as controlled experiment to analyze the results addressing RQ2. Prompt engineering patterns are used to get responses for the prompts in LLMs such as ChatGPT and Github Copilot so they are more straightforward and targeted towards specific outcomes. This will help maintain uniformity as well as enhance prompt generation depending on the learning data, which can be used for things like refactoring code.

3.1 Methods for Research Question 1

RQ1: What refactoring patterns do the Python programmers employ while dealing with various code smells?
RQ1.1: Which code smells are prioritized and which ones are ignored during the refactoring?
RQ1.2: What challenges do they encounter in the process of the refactoring?

To answer the above-mentioned RQ1, we used a semi-structure interview strategy as we wanted to get a deep understanding of Python developer's perceptions. Qualitative studies collect and analyze data using multiple sources e.g., interviews, observations, and data analysis. In our case, we opted to conduct semi-structured interviews [11] with Python developer's because we wanted to get close to the developer's context so we could observe and gather more accurate information about code smells and refactoring practices directly from them. During the interviews, we asked developer's to describe their understanding of code smells and how they refactor code. Figure 3.1 shows step-by-step process of the research method that we followed.

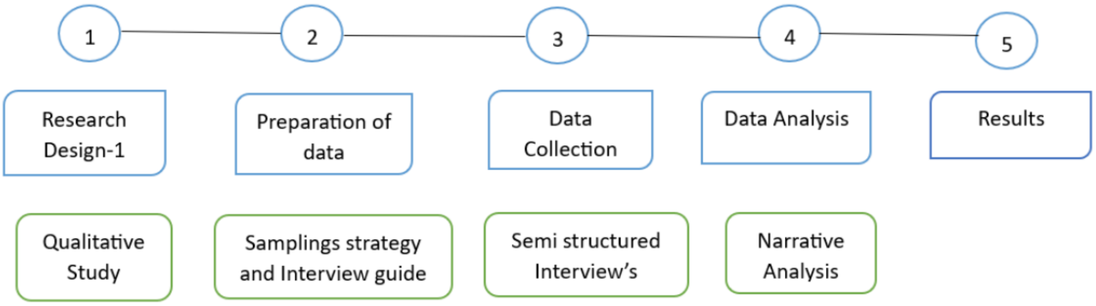


Figure 3.1: Research Method Process

3.1.1 Qualitative research method

Qualitative analysis consists of conducting an in-depth study with diverse data. Typically, in qualitative research, the analysis involves processes such as coding, categorizing, and interpreting data gathered through methods like interviews, observations, or document analysis. Qualitative analysis also helps us understand the real-world scenario since we are having direct communication with the Python developer's. In addition, it provides us with significant value when it comes to interpreting real-life situations. This consists of oral conversations, written documents, reports, document analysis, and observations, all of which are supported by research[12].

We have conducted initial planning for the method of sampling, ethical concerns, and the preparation of the interviews, with the introduction to code smells and refactoring all of which are important for data collection. In addition, we have performed an in-depth review of the existing literature to discover relevant works related to our research topic. Data has been collected through semi-structured interviews with participants along with the consent form.

3.1.2 Preparation for the Data Collection

The preparation for the data collection in this qualitative study is described in section 3.1.3. The interview procedure, the written content, the preliminary literature analysis, and the justification for the chosen samples are covered in the subsections. The interview guide served as the main interviewing document. The interview script, which included guidelines for the interviewee and the questions to ask was included in the interview guide.

Sampling

Sampling is the method we have selected to choose participant's for the interviews. In qualitative research, sampling is a commonly used method for the participants selection. The objective was to conduct a comprehensive investigation and exploration of the topic through interviews with industry professionals with expertise in Python as well as researchers with a range of experiences and viewpoints in software development and maintainability. The role of the individuals and their willingness to participate were the selection criteria.

The recruitment process involved two pilot studies. The pilot implementation plan was developed collaboratively with the people participated in the pilot study [10].

Pilot studies helped us to understand the feedback of our interview questions. We adapted the changes that were given as feedback from the pilot study participant. We felt more prepared and were able to ask more questions throughout the pilot study, which also helped us to clarify several aspects of the procedure.

We reached out to the potential interviewees through multiple links, personal connections, social media networks, and references from other participants. We successfully recruited ten participants who are working in different industries across Sweden where four participants work with Python in AI projects, rest six work with Python in software development ranging from 2 - 13 years of experience in the automotive industry. We approximately approached 20 individuals within our personal and social networks. Therefore, the participant's responses provided insights into various Python domains.

3.1.3 Data Collection

The techniques and procedures used for gathering data in the qualitative study will be addressed in this section.
The interviewing strategy, interviewing methodology, and the data arrangement and transcription for the next stage of the study are described in the following subsections.

We have followed a five-step process for conducting, analyzing, and reporting semi-structured interview data. The initial four steps focus on the preparation phase.

Step one involves determining the appropriate sampling strategy, including the selection criteria for participants.
The second step is gaining authorization to conduct interviews and scheduling the team meeting with the participants.
The third step is deciding which data collection techniques are best suited for the study objectives and the kind of data that is needed.
The fourth step involves creating the study tools, such as data gathering methods and interview protocols such as signing the consent form and sending the introduction and study purpose of the thesis. After this we conducted online interviews with ten participants individually and collected the audio recordings and transcripts from the interviews for our analysis [11].

Interviews Questions

We have selected a semi-structured interview approach for the interviews. One of the main advantages of semi-structured interviews is that they allow us for a focused interview while allowing the investigator to freely explore relevant concepts that may arise in the interview, which might help improve comprehension of the service under evaluation. This narrative review's objective is to provide an overview of the process and methodological aspects of conducting semi-structured interviews [11].

Additionally, at the start of the interview, we showed several pictorial illustrations of refactoring patterns. This particular illustration was retrieved from the following website [13] which allowed the developer's to identify the pattern's names and comprehend the visual associations with them. The following are further descriptions of the Illustration patterns [23].

Extract Method: This pattern involves the deconstruction of extensive, complicated methods or functionality or approaches. We are able to build new methods that can operate independently, thus improving the readability of the codebase. This reduces the amount of duplicated code. In the given figure 3.2 the block comprises of eight lines before being broken down into just three lines of code showing that a portion of the code has been extracted and placed into a new method.

Move Method: This pattern moves a method from one class to another where it suits better. The overall structure of the class to which the method is being moved is improved, and the code feels more natural and intuitive. In the given figure 3.2, the arrow illustrates the method is moving to another class.

Inline method: This pattern is used when a method is too short to enable the method's name to add enough importance to the program. In the given figure 3.2, the smaller method is integrated back into a larger method.

Rename Variable/Field: A known variable/field is renamed into a more informative one.

Encapsulate Collection: Involves encapsulating collection fields and creating methods to control how the fields are read and updated. In the given figure 3.2 accessible collection that is available through methods only.

Replace Conditional with Polymorphism: Where a conditional is being converted to a class hierarchy using inheritance or interfaces. In the given figure 3.2 a conditional statement is replaced with a class hierarchy using inheritance or interfaces.

Introduce Parameter Object: Pattern which consisted of many parameters being modified to accept a single parameter object. In the given figure 3.2 a method with multiple parameters being replaced to take a single parameter object.

Extract Class: This is the pattern used when a class becomes excessively large its segmented into smaller units. In the given figure 3.2 a large class is split into smaller classes, each handling specific responsibilities.
An interview guide that provided information on data collection and processing methods was presented to the participants before the start of each interview, ensur-

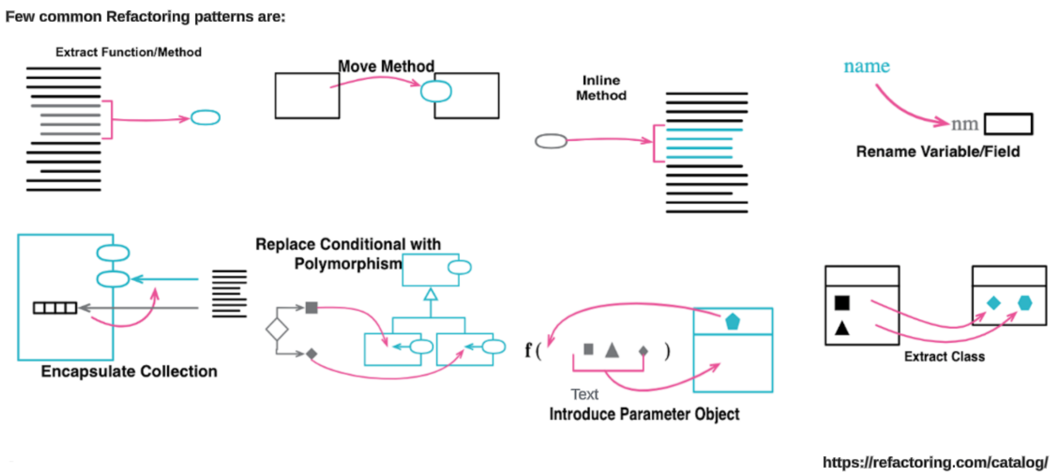


Figure 3.2: Figure shows different refactoring patterns.

Name	Role	Experience
Interviewee A	Senior Software Developer	13 years
Interviewee B	Senior Software Integration Developer	6 years
Interviewee C	Data Engineer	5 years
Interviewee D	Test Automation Engineer	3 years
Interviewee E	Python Developer	6 years
Interviewee F	Full Stack Python Developer	More than 3 years
Interviewee G	Software Developer	5 years
Interviewee H	Software Developer	Almost 7 years
Interviewee I	Full Stack Python Developer	5 years
Interviewee J	Python Developer	2 years

Table 3.1: The list of participants, their roles, and their experience.

ing privacy and anonymity. Teams and Zoom were used to conduct the interviews, and each session lasted roughly 45 minutes. The research was carried out by us, we divided interviews between ourselves. Each time an interview was conducted, the interviewer provided their professional background data, described the goals of the thesis, and got permission to record the interview.

3.1.4 Data Analysis

This section describes the process and steps that we followed for qualitative data analysis.
We conducted interviews with the above table 3.1 ten developer's from the industry.

Process for qualitative data analysis
We selected narrative analysis as our technique for qualitative data analysis because we were aware of its adaptability and popular usage in giving researchers an effective tool for examining and deciphering interview data. The narrative analysis produced detailed narratives of the informants' truths. It describes the process of looking for

patterns, trends, and insights in textual data produced by users or systems [12].

We utilized a data analysis tool to simplify the analysis process. The software has the benefit of enhancing work efficiency [12]. We chose MAXQDA as a tool to organize our research data. The MAXQDA software improved the analysis and interpretation of qualitative data. It enhanced the process of organizing, arranging, and analyzing the data. The total number of texts or other data helps in the handling of the resultant interpretations and assessments. This tool helped us to apply multiple strategies in parallel, like reading, reflecting, memo writing, coding, and segmenting. Data interpretation and evaluation involve the process of sorting data into codes and segments.

We have followed Yong Nie's [12] systematic method for conducting narrative analysis, which comprises different phases described below:

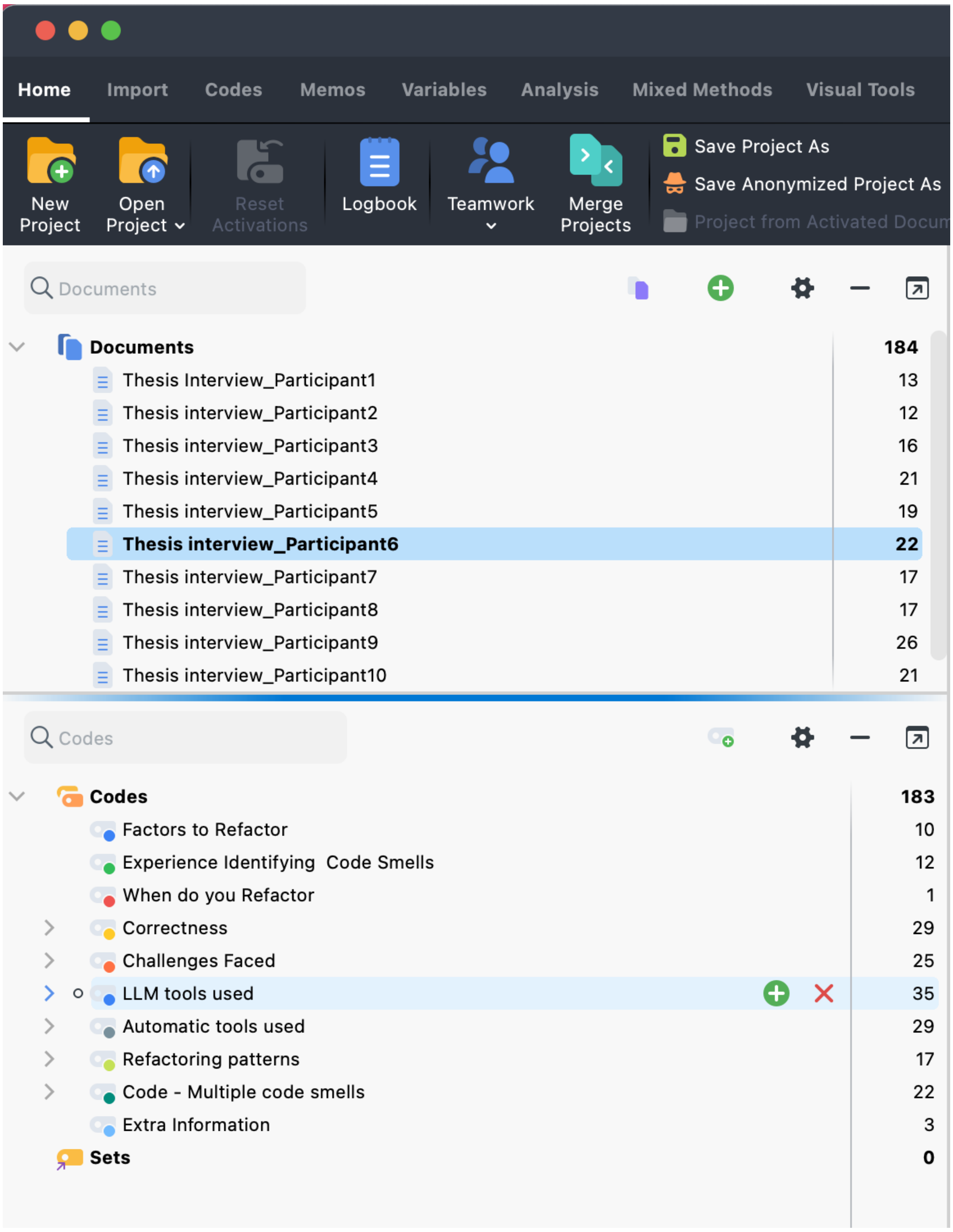
First Step: Uploading the transcript and making initial coding.
After conducting interviews with the participants, we utilized a feature that records and transcribes audio into text automatically. We successfully saved and uploaded these transcripts to Google Drive once the interviews concluded. From there, we imported the transcripts into MAXQDA software for further analysis.

Next, we broke down the interview questions into smaller initial codes. This process involves dissecting the main questions into more specific and manageable units of analysis. These codes help organize the data and make it easier to identify focused code, patterns, themes, and insights during the analysis phase. As shown in the figure 3.3

Second Step: Focus on coding and labeling data in the analysis software.

After getting an overview of our research data and initial coding we analyzed the data through focused coding, which is the second major phase in the coding process. Once we identify specific analytical directions using interview questions, we may continue with focused coding to combine and interpret bigger parts of the data. Our objective was to develop an understanding structure and categorization approach to capture the key aspects of the research.

This involves the process of comparing labels with other labels, and sometimes even comparing labels with the original codes or raw data. It additionally includes studying the natural relationships between these labels and combining closely related labels to create a new concept. We have included "In vivo" coding based on participants, which entails code, labeling data segments using the participants' own language or words during the coding process. This method helps us preserve the authenticity and diverse nature of the participants' views since it directly reflects their words and meanings. By incorporating in vivo codes, we want to accurately capture the richness of participants' perspectives and activities, increasing depth and allowing us to preserve the participants' meanings of their views and actions,



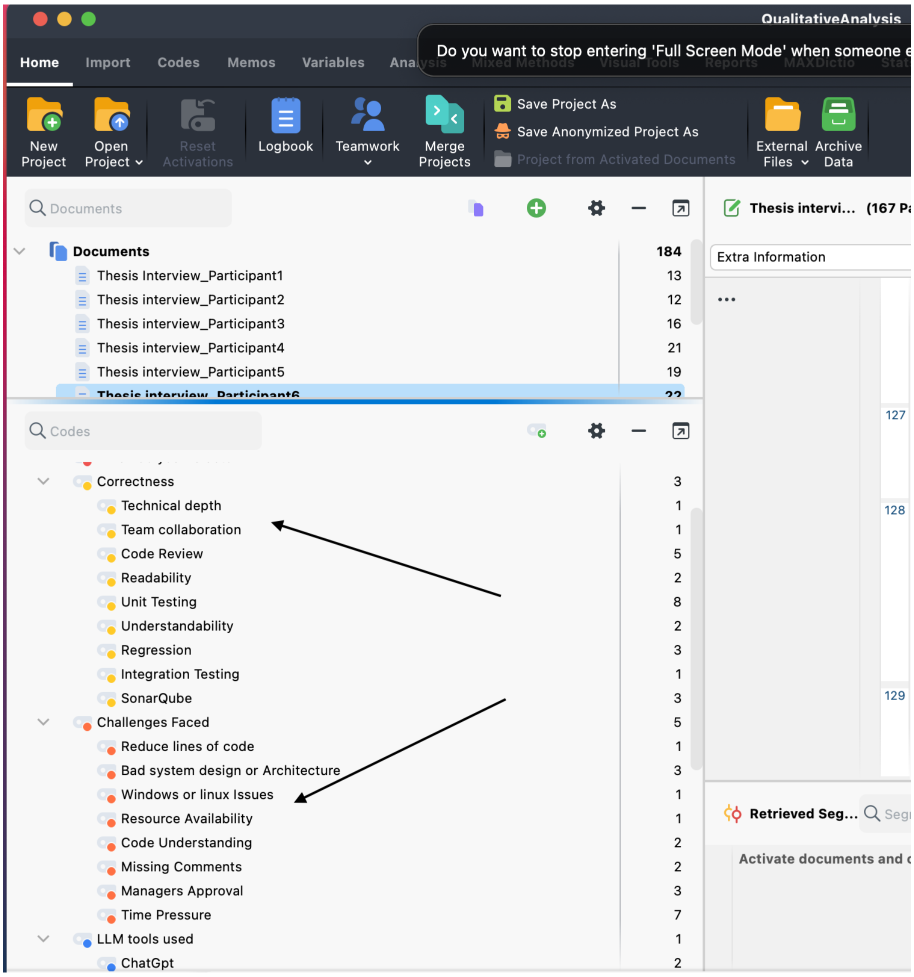
Documents	**184**
Thesis Interview_Participant1	13
Thesis interview_Participant2	12
Thesis interview_Participant3	16
Thesis interview_Participant4	21
Thesis interview_Participant5	19
Thesis interview_Participant6	**22**
Thesis interview_Participant7	17
Thesis interview_Participant8	17
Thesis interview_Participant9	26
Thesis interview_Participant10	21

Codes	**183**
Factors to Refactor	10
Experience Identifying Code Smells	12
When do you Refactor	1
Correctness	29
Challenges Faced	25
LLM tools used	35
Automatic tools used	29
Refactoring patterns	17
Code - Multiple code smells	22
Extra Information	3
Sets	**0**

Figure 3.3: Code segments

as well as the authenticity of the analysis.

After analyzing the interviews, we were able to identify nine different categories in the MAXQDA software. This helped us conduct a more detailed analysis of the data. Every category acts as a topic shield that helps organize the relevant segments of data. We carefully examined every transcript, organized categories into segments using a tree-like structure in the software as shown in Figure 3.4 [12]. By following a systematic approach, we can thoroughly examine the data. In addition, the software offers a user-friendly platform for exploring the links between different segments and the corresponding raw data, notes, and audio files collected while performing the research.



Documents	184
Thesis Interview_Participant1	13
Thesis interview_Participant2	12
Thesis interview_Participant3	16
Thesis interview_Participant4	21
Thesis interview_Participant5	19
Thesis interview_Participant6	22

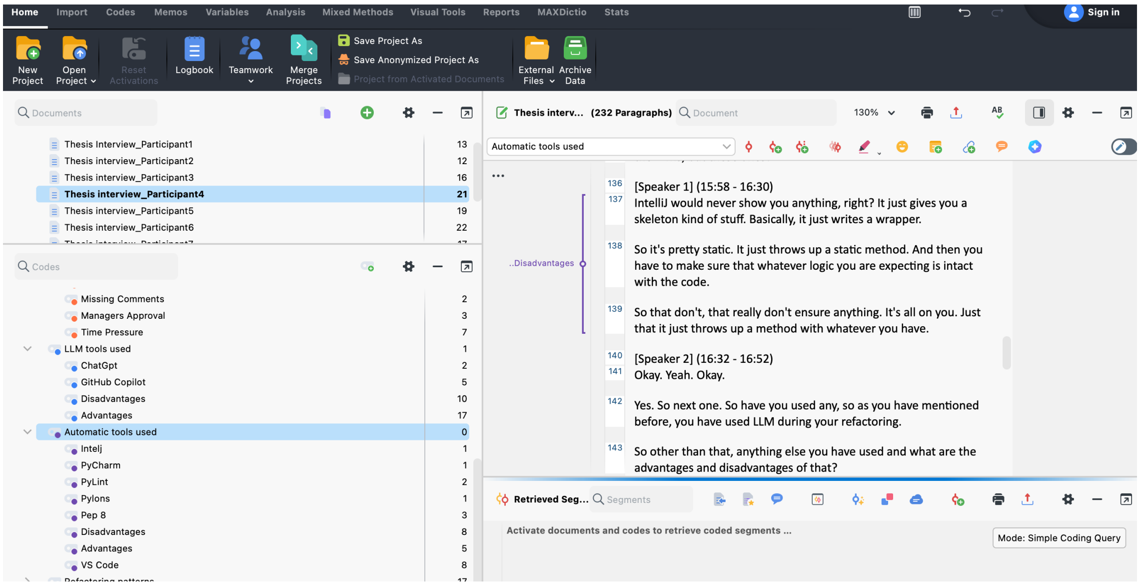
Codes	
Correctness	3
Technical depth	1
Team collaboration	1
Code Review	5
Readability	2
Unit Testing	8
Understandability	2
Regression	3
Integration Testing	1
SonarQube	3
Challenges Faced	5
Reduce lines of code	1
Bad system design or Architecture	3
Windows or linux Issues	1
Resource Availability	1
Code Understanding	2
Missing Comments	2
Managers Approval	3
Time Pressure	7
LLM tools used	1
ChatGpt	2

Figure 3.4: Category and code segments

Step Three: Initializing a label with the essential meaning of its text:

In this step, we started by initializing a label that represents the meaning of a particular code or segment. After labeling, we carefully examined each participant's transcripts to find the sections of text that express similar words. In this process, we look at identifying recurring words or phrases that are found in all the transcripts since they provide important concepts within the data. After identifying these texts, we carefully highlight the relevant sections with a specific label. The highlighted text is connected to each label, and if there is any repeated text, there are points associated with it.

Through this approach, we consistently highlighted the text with a label, therefor a well-organized structure was created. By this way of analyzing, we were able to identify and extract important points from the transcripts and therefore allowing us to understand various perspectives of the participants regarding our Research question as shown in Figure 3.5.



Thesis Interview_Participant1	13
Thesis interview_Participant2	12
Thesis interview_Participant3	16
Thesis interview_Participant4	**21**
Thesis interview_Participant5	19
Thesis interview_Participant6	22

Missing Comments	2
Managers Approval	3
Time Pressure	7
LLM tools used	1
ChatGpt	2
GitHub Copilot	5
Disadvantages	10
Advantages	17
Automatic tools used	0
Intelj	1
PyCharm	1
PyLint	2
Pylons	1
Pep 8	3
Disadvantages	8
Advantages	5
VS Code	8

136 [Speaker 1] (15:58 - 16:30)
137 IntelliJ would never show you anything, right? It just gives you a skeleton kind of stuff. Basically, it just writes a wrapper.

138 So it's pretty static. It just throws up a static method. And then you have to make sure that whatever logic you are expecting is intact with the code.

139 So that don't, that really don't ensure anything. It's all on you. Just that it just throws up a method with whatever you have.

140 [Speaker 2] (16:32 - 16:52)
141 Okay. Yeah. Okay.

142 Yes. So next one. So have you used any, so as you have mentioned before, you have used LLM during your refactoring.

143 So other than that, anything else you have used and what are the advantages and disadvantages of that?

Activate documents and codes to retrieve coded segments ...

Figure 3.5: Category and Label Highlighted

Step four: Conclusion

After thoroughly analyzing all 10 transcripts of the participants, and considering the points assigned to each code and category, we have chosen several key topics for further visualization. These include what refactoring patterns are prioritized, which code smells are considered, Which LLMs they work on, which automated refactoring tools do they use to refactor, what methods for evaluating the correctness of refactoring, and the challenges encountered in code refactoring.

A detailed discussion of our findings regarding these topics is provided in the result section of RQ1.

3.2 Methods for Research Question 2

RQ2: How correct and understandable are refactoring suggestions in addressing code smells?

RQ2.1: How do the understandability and correctness of code refactoring suggestions provided by LLMs compare with various LLMs?
RQ2.2: How do the understandability and correctness of code refactoring suggestions provided by LLMs compare with that of a human expert?

This section discusses the research methodology utilized in the RQ2. An overview of the research design for RQ2, RQ2.1 & RQ2.2 is provided in Figure 3.6. The individual steps are discussed in further detail in the following sections.

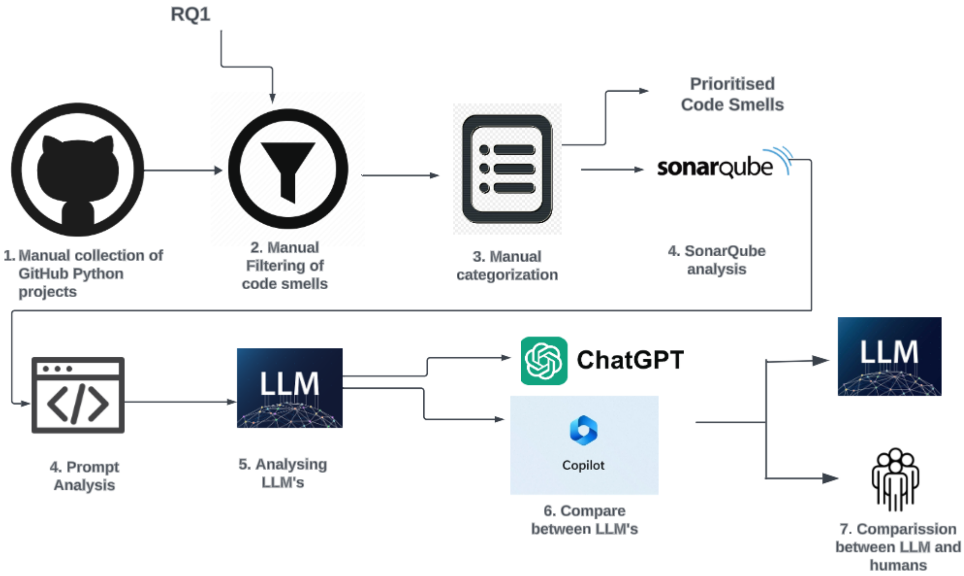


Figure 3.6: Overview of the experiment method applied in the thesis

3.2.1 Pre-Study for Prompt patterns and methods for RQ2.1

Data Collection This section discusses the data collection strategy used to find the sample of Python projects. In total, we were able to collect three Python projects.

Sampling Strategy This thesis is a study that was conducted using a sampling approach to collect Python projects with code smells. This means that these sample projects were explicitly selected according to some predefined criteria and not randomly. The data considered in this thesis involves Python projects. Essentially, the thesis is interested in Python projects that contain code smells in the code that is used to some extent to solve prioritzed code smells and provide it into readable projects. We have chosen Python projects from GitHub, and subsequently analyzed them using SonarQube, demonstrating the quality of these Python projects.

Additionally, the categorization and exclusion of samples was a manual process, checking if the code consists of both codes smells files and refactored files. In the categorization we divided them into the code files containing duplicated lines of code, long methods and complex conditional statements. These code smells were separated into different files based on each prioritzed code smells and further each code smells had separate training files and testing files. The Training files were used for Prompts optimization.

Inclusion Criteria We chose only Python files with code smells that consist of complex conditional statements, duplicated lines of code, and Long method. Files needs to be in a manageable size to ensure the task is moderate to test refactoring. Open source files are used for the research which are publicly available in the git repository projects.

Python developer's are recruited as Participants whose experience lies between 3 to 8 years in software development and who regularly perform refactoring in the industry. Prominent LLMs chosen for thesis include ChatGPT version 4.0 and GitHub Copilot for refactoring tasks. Python developer's are recruited as Experts whose has more than 10 years of experience in software development and who has a deep understanding of Python. Experts need to be able to evaluate code refactoring, particularly in understanding the code and validating the correctness of the refactored files.

Exclusion Criteria During the data collection, some exclusion criteria were defined to guarantee the quality of what would be considered relevant and representative Python projects asserted as being part of the analysis. In order to be able to perform any analysis, we exclude projects that do not use English in their codebase. Also, the Python code of projects had to necessarily contain code smells (projects with clean or well-structured code were excluded). Avoided other programming languages in preference for focusing completely on Python. In this way, these projects can be estimated as realistic and they contain the desired properties for investigating refactoring techniques and code quality issues.

Categorising of code smells In this thesis, the studies of source code projects of GitHub repositories created with Python language were analyzed using SonarQube, which is one of the most preferred static code analysis tools. As for the analysis, it is automated with locally performed tests to provide a specific setting for evaluating the code quality. Then, we distinguish between specific types of the detected code smells such as duplicated code, complex conditions, and long methods. These code segments are grouped according to certain categorizations and placed in different folders. After that, these segmented codes are used for training and testing by employing a Github Copilot, Chatgpt for the betterment of the code quality and for its improvement.

Prompt Engineering Techniques We have incorporated Code generation and execution techniques [14] where in we have provide a code along with the prompts to guide the ChatGPT and GitHub Copilot in refactoring tasks. This method helped the models to understand the context of the code provided better and generate more readable code. This method involved providing the models with sample codes with code smells that require refactoring. This context helps the LLMs to understand the current structure and logic of the code.

Prompt engineering techniques are an approach of developing and enhancing templates – messages provided to an LLMs to compel it to adhere to rules or standards or to perform prescribed tasks. This technique is a form of programming that enables the setting of the outputs and the manner through which an LLMs can engage with the programmers. Common problems that arise during interaction with LLMs are addressed in pattern form as in the prompt engineering techniques denoting reusable applications of these patterns corresponding to the software patterns. Since it is a largely recently coined field, meaning utilizing LLMs in natural language processing tools, the focus of prompt engineering is to enhance prompts for better performance. This field helps to improve LLMs performance on various applications and in different research areas. Specifically, prompt engineering entails matching LLMs needs with the design of scripts and questions that aid in the identification of relevant information, which in turn enhances the prompts' effectiveness [15][4].

We chose to apply the prompt patterns to LLMs like ChatGPT and copilot based on the results from RQ1. OpenAI created ChatGPT, an advanced programming model built on the GPT-4 architecture. It can provide responses that are human-like in a variety of situations. In regards to parity and channels of communication, prompts are the main way for the user to interface with LLMs. They help the model in the formulation or construction of responses in a way that fits into the user's expectations. An essential aspect of crafting prompts is anticipating how the model will interpret and respond to the given input. This involves considering factors such as the model's training data, biases, and other limitations that may influence its understanding and generation capabilities [20].

Techniques for Effective Prompt Engineering

1. Clear and specific instructions:
We have followed a clear and specific methodology outlined in the paper.
We used various prompt patterns with ChatGPT and Github-copilot, including basic questions, and we observed that these LLMs provided vague responses, based on the responses we rephrased the prompts by providing clearer and more specific instructions. This was our first prompt given for Chatgpt.

Example_Prompt_1: *"Remove duplicated lines of code"*
Additionally, we noticed the uncertainty in the prompt may result in responses that fail to satisfy the expectations of the developer's hence we have revised the prompt

Example_Prompt_Revised_1: *"From the Identified list of code smells remove duplicated lines of code"*
We wanted the LLMs to perform refactoring using a specific refactor pattern to remove specific code smells, The prompt was optimized using training data.

Optimized prompt 1: *"From the identified list of code smells remove duplicated lines of code using the extract method"*

The same procedure was followed for Github Copilot [21] **Example_Prompt_1**: *"Fix the problem in the selected code"*

Example_Prompt_Revised_1: *"Fix the duplicated lines of the code in the selected code"*

Optimized prompt 1: *"Fix the duplicated lines of code using the extract method in the selected code"*
We reached optimized prompt by iterative refined and testing the prompts by providing it to the LLMs multiple times (more than 5 times), ensuring that the responses from LLMs were consistent and relevant.

Optimized prompt with more specific Instruction: *"Fix the duplicated lines of code using extract method provide the method name or class name"*

We observed that while using GitHub Copilot, it is important to provide specific instructions, such as the method name or class name to provide an expected outcome.

2. Ensuring output consistency:

To keep LLMs output consistent through repeated prompting, one needs to force-feed similar or exact questions to the model. This practice helps to control the productivity rate and make certain that the model does not lose focus on the task by regularly revising the work it has done. By employing effective and well-written prompts from question formulation we were able to create a more coherent and concise type of response format from the LLMs, therefore avoiding inconsistency and increasing the credibility of the generated content. This technique is important in cases where it is desirable to have the model perform uniformly and give consistent results.

3.2.2 Methods for RQ2.2

Data Collection For answering RQ2.2 we choose one Python project from 3.2.1. Software engineering is characterized by controlled experimental studies, which constitute a research category in which researchers alter certain variables and monitor the change in the outcome of interest while keeping other factors constant. These types of experiments are frequently applied in the context of study participants,

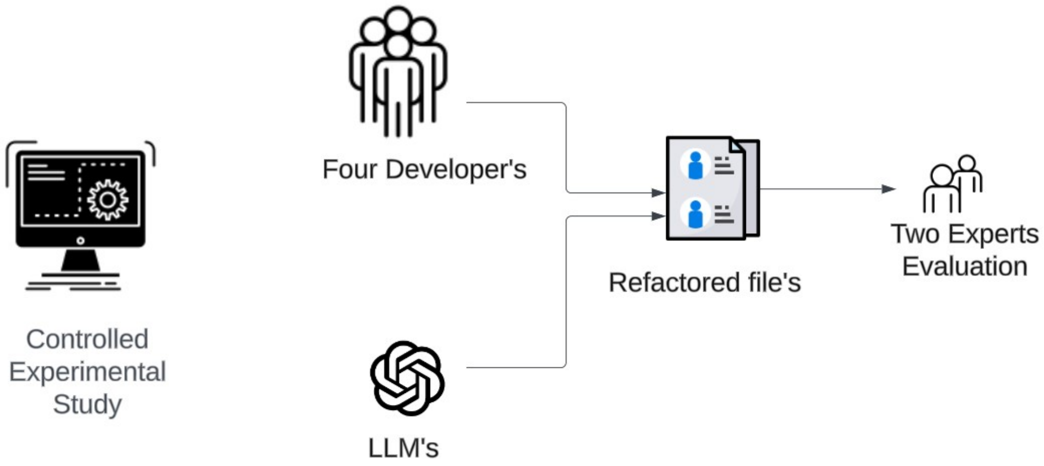


Figure 3.7: Comparison between humans and LLMs

where some treatment is applied to one group of subjects, and a different one is applied to another. Randomization is employed for the purposes of objectivity. The goal is to demonstrate how such elements influence the outcome measures, and more importantly, underline the quality and relevance of the said outcomes, especially when human factors are involved and both qualitative and quantitative measures are present [22][24].
Our experiment was conducted on an online setting, where an experiment is conducted and a limited part of the software development process is executed in a controlled manner.

Sampling We have decided to choose four participants and 2 LLMs for code refactoring. The objective was to conduct the investigation of refactored code smells with two industrial experts. This process also involved 1 pilot study. Pilot study helped us to understand the feedback of our questioners. We felt more prepared and were able to ask more questions throughout the pilot study, which also helped us to clarify several aspects of the procedure.

Code Smell Selection:
We selected two code smell files containing multiple code smells from one of the above-chosen Python projects from 3.2.1. These files consist of different types of code smells such as long method, large class, Duplicate lines of code, and nested conditional statements which represent the issues while refactoring the files. Each file was approximately 200 lines of code, ensuring that the task was substantial enough to test refactoring skills without being too lengthy and time-consuming for the developer's.

Experiment Principles:
The primary objective of the experiment is to evaluate the impact of the source of code refactoring (whether done by human developer's or LLMs) on the understandability and correctness of the refactored code. Initially, two code smell files with

an average of 200 lines were chosen and given to the developer's from the industry. These developer's were instructed to refactor the files by removing the code smells manually. For the same code smell files LLMs were applied. The outputs from both developer's and LLMs were then subjected to evaluation and comparison by two industry experts.

To evaluate the objective the experiment is designed with one factor with two treatments:
One Factor with Two Treatments we want to compare the two treatments that is Human refactored files and LLMs refactored files against each other based on the understandablility and correctness.
Treatment 1: Code refactoring was performed by 4 human developer's from the industry who had between 3 to 8 years of professional experience and were instructed to manually refactor the code without the use of automated tools or LLMs.
Treatment 2: Code refactoring performed by LLMs (ChatGPT and GitHub Copilot), we also utilized two popular LLMs-based coding assistants, ChatGPT and GitHub Copilot, to perform automated refactoring of the same code smell files. Both these treatments resulted in refactored files. These refactored files are then provided to two independent experts, each with more than 13 years of experience who evaluate the files for understandability and correctness.

Management of variables:
Dependent variables - the comprehensibility and correctness of refactored code as judged by experts.
Independent Variable by file: The developer's and LLM-specific refactored files that are extracted from 4 developer's. and 2 LLMs.

Experiment Process:
Step 1: Identification of Code Smell Files Two code-smell files, each including 200 lines of code, were selected from a Git repository of Python projects. These files exhibit code smells, such as duplicated lines of code, long methods, and complex conditional statements.

Step 2: Two Tasks were given to each of the developer's.
We selected four experienced software developer's from the industry, each with 3 to 8 years of experience, to manually refactor the code smell files. This was conducted online via email. We gave clear instructions to the developer's to remove the code smells without using any automated tools or LLMs.

Step 3: Automated Refactoring by LLMs.
For the same two code smell files, refactoring was also performed by two Large Language Models (LLMs): ChatGPT and GitHub Copilot. The LLMs were provided with prompts from RQ2.1 to refactor the code "From the given file removed duplicate lines of code" or "fix the problem". The LLMs processed the code and returned their refactored versions.

Participants	Filenames
Participant A - Developer (2 -3 years)	A_Refactored_task
Participant B - ChatGPT	B_Refactored_task
Participant C - Developer (7 years)	C_Refactored_task
Participant D - Developer (5 years)	D_Refactored_task
Participant E - Developer (7 years)	E_Refactored_task
Participant F - Copilot	F_Refactored_task

Table 3.2: The list of participants, their experience, and filenames.

Step 4: Data Collection of Refactored Files.
Four refactored files were obtained from the LLMs which were stored in separate folders with their respective names and 2 files each were stored from each developer's with the filenames as shown in 3.2 in a separate folders. As a result, we obtained 8 refactored files from the developer's and 4 refactored files from LLMs (ChatGPT and GitHub Copilot) for two tasks. Later these files were named in alphabetical order to hide their anonymity and provide it to the experts.

Step 5: Evaluation by Experts Two industry experts, each with over 13 years of experience, were provided with 12 sets of refactored files, 6 sets of refactored files for task 1 and 6 sets of refactored files for task 2 respectively along with the qualitative questionnaire, The evaluation was based on sets of qualitative questions to verify the understandability and correctness which are added in an appendix.

3.2.3 Data Analysis

This section describes the process and steps that we followed for qualitative data analysis. We had sent out 7 questions for task 1 and task 2 to the experts along with the 12 sets of refactored files.

Data Collection and Initial Coding:
After gathering the refactored files from both LLMs and human participants, we organized the data and sent them to two experts along with seven questions per task to evaluate the refactored files, the result was received in the MSWord format. This result was analyzed using MSWord. The files were uploaded to Google Drives and prepared for initial coding, where we created codes representing essential categories in the refactoring tasks, such as understandability of the code, breaking down of long methods, correctness of functionality, and reduction of code smells. For a clear and orderly process, the feedback provided by the experts was written down. This helped in aligning the information gained from the transcriptions and afforded us a more structured way of appreciating what each of the experts thought regarding different elements of refactoring.

We synthesized the codes into narrative summaries that encapsulated each expert's evaluation insights. This step involved a comparative analysis of the labels across human and LLMs refactoring outputs, assessing strengths and weaknesses in terms of given initial coding. Key findings were extracted and organized into visual summaries to highlight areas where LLMs closely matched human output and areas where they diverged significantly.

We articulated the codes into a narrative summary that consists of each expert's assessment. This phase compared LLMs and Human refactoring output labels regarding the strengths and weaknesses of given initial coding. Key results were outlined and presented in the result section 4.2.4 where visual summaries highlight areas where LLMs closely matched human output and areas where they diverged significantly.

4

Results

The findings gathered from the interviews are organized in this chapter and aligned with the RQ1 and RQ2. Section 4.1 delves into the outcomes related to the decision-making patterns that developer's use to choose practical refactoring solutions for different types of code smells in industries and section 4.2 delves into outcomes related to prompt patters and refactoring suggestions from both ChatGpt and GitHub Copilot.

4.1 Practitioner's perspectives on refactoring patterns - RQ1

We have explored the results concerning the refactoring patterns, prioritised code smells, and challenges faced by software developer's.

4.1.1 Code smells - RQ1.1

Code smells in software development are a very crucial aspect of maintaining the code quality hence readability and better maintainability. Based on the analyzed results of the interview, we have identified duplicated lines of code as a code smell that was chosen by 10 participants. Additionally, complex conditional statements were prioritized by 2 participants. Upon examining the code on GitHub, we observed that most of the complex conditional statements are present within long methods. Therefore, we have combined these two code smells. To narrow down the scope, we have excluded the shotgun surgery and unused functions.

Duplicated lines of code.

"Like Duplicate lines of codes as it is an easy one to spot on and an easy one to fix, just to Break out that base into a function." - Interviewee A

"When you're working in multiple projects with multiple team members, And when you have a larger code base, it's natural for one person to have worked on one piece of code, and probably somebody else could pick up the same piece of code or different code and write something with similar logic. So what happens is, if I have written something, probably to start a set of rows, and if there is a requirement for somebody else also to write it, he may also write it again. So with this, what happens is, that

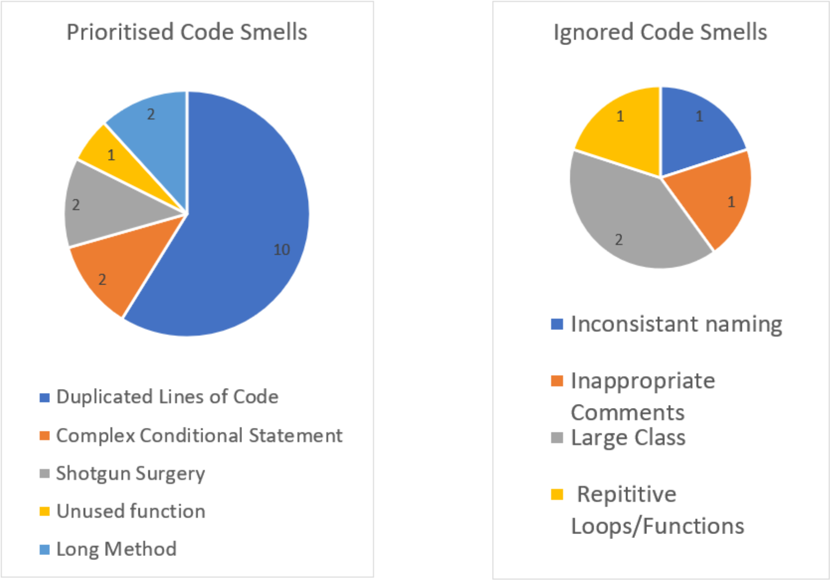


Figure 4.1: Prioritized code smells and Ignored code smells

is code duplication. So if there is a code repetition or duplication and if that could break, or if I see that it could be a potential disaster in production, then we prioritize it based on that." - Interviewee B

"We see that duplicated lines of code or repeated code try to affect the performance or the run times. So I guess we don't need such lines of code, So I would rank them, slightly higher compared to the other priority tasks. -Interviewee C

As mentioned by this interviewee we conclude that Duplicate lines of code are easy to identify and refactor, It's a common code smell in larger projects with multiple team members and can lead to problems in production. This code smell negatively impacts performance and run times and should be given higher priority during refactoring. By this we conclude that duplicated lines of code are important, and needs to be addressed. While we understand that Interviewee A is communicating on the ease of fixing the issue, whereas Interviewee B and Interviewee C highlight the broader problems. Figure 4.1 shows the summarized result of Prioritized and Ignored code smells chosen by the developers

Complex conditional statements and long methods.

"First, understand why you're using complex conditional statements or nested loops, what are you trying to achieve in the 1st place, It's very important to minimize the number of loops to avoid complication." - Interviewee - F.

"A long method with too many for loops and many if conditions inside it, we got an issue in that particular function and it was very difficult to understand the code." - Interviewee J.

As mentioned by this interviewee we conclude that refactoring complex conditional statements in a long method should be a top priority. This will improve code readability, and reduce the issues arising from complex code.

Ignored code smells.

Figure 4.1 shows the Ignored code smells.
"When you have a function that has no comments and it might be a pain to go in and refactor, but if it works and you're not doing anything with it, that's the easiest one I guess to just Ignore or leave." - Interviewee E.

"Large classes where we need time and thoughts to understand what is the essence of this class and what classes can it be divided into. So I think it better to fix low hanging fruits than large classes." - Interviewee F.

"If its renaming some attributes or functions, or if it is just replacing some magic numbers, then I don't think it is that high priority task. So you would rank them lower compared to if you have certain code which has nested." - Interviewee A.

In summary, we notice that when functions lacks comments, when there is a large class it can be hard for the developer's to refactor and a lot of time is spend on it whereas renaming attributes or replacing magic numbers often considered lower priority compared to more complex conditions. Overall, the aim is to prioritising code smells that provides a the most immediate and significant improvements while balancing time.

Below 4.2 shows the different automated refactoring tools and LLMs used by developer's within in Software development teams.

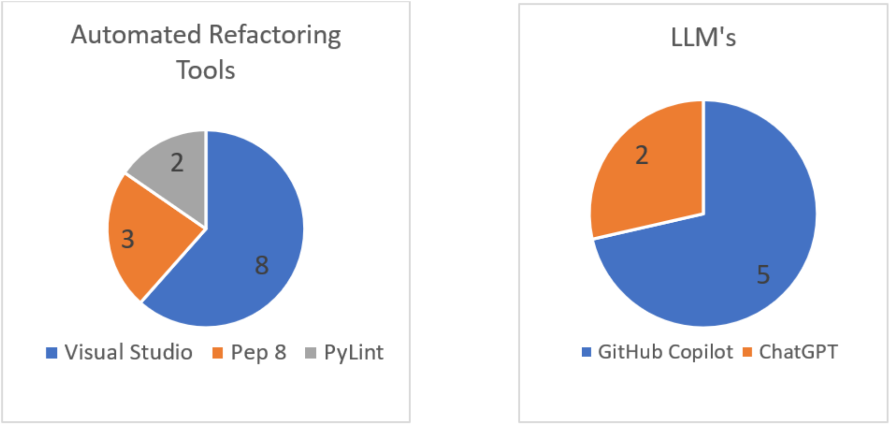


Figure 4.2: Automated refactoring tools and LLMs used by developer's

Refactoring Patterns.

"It's better to divide big functions into smaller ones for greater clarity. This makes code more readable. For instance, it can be done by applying an extract method.

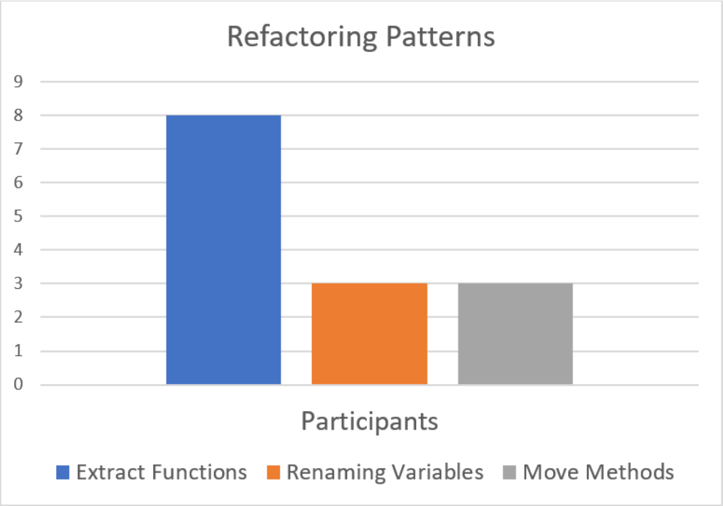


Figure 4.3: Refactoring Patterns used by participants

When you copy a code that is complex or has to be repeated several times, you take part in the process of creating extract method. This involves making smaller functions which can store these long pieces of code separately to be invoked when required." - Interviewee D.

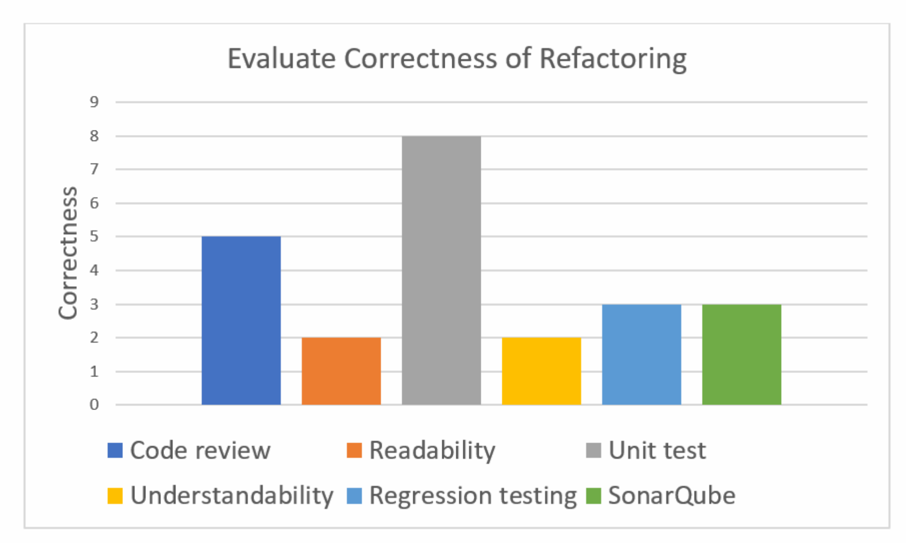


Figure 4.4: Evaluate the Correctness of Refactoring

"Commonly, we make use of patterns to simplify code such as the extract method. Specifically, where there are numerous for loops and if conditions within a single function, however, this can be resolved through the usage of the extract method as it enables breaking down complex areas into smaller units which are easier to read thus maintaining them all through." - Interviewee E.

In the opinion of both participants, the extract method simplifies long and hard code by breaking it into smaller parts that are easy to handle which generally leads to easier read as well as maintenance of the software.

Evaluate the correctness of Refactoring.

"There are multiple unit tests that we have written, to evaluate correctness. So like when you develop code, you have unit tests and some other regression tests that you have. Unit test also helps to improve code" - Interviewee B.

"The unit tests are written to make sure of the simplest way for correctness. Starting with unit tests first for all components separately as a rule allows us to check them one after another. To do this, complex tasks that involve creating deep learning code require that other parts of the code are switched off and only this segment is concentrated upon at a time. It will enable us to turn on and test them with a methodical approach as we go step by step thus ensuring comprehensive validation during the development process." - Interviewee D.

According to both the interviewers, there is a need to undertake several unit tests to determine if the codes are working correctly or not as well as raise the quality of the codes. This includes having one unit test for every single part thereby giving room for incremental validation. A comprehensive check of the code can be done when one enables and tests each part thoroughly.

4.1.2 Challenges faced for refactoring process - RQ1.2

Figure 4.5 shows the challenges faced by the developer's in the industries while performing refactoring tasks.

"Time requirement is probably the greatest challenge, as we have a lot of projects, we have a lot of tasks and we have no time to do refactoring. " - Interviewee I

"The hardest thing is to convince that your time spent refactoring code is worth it, since the nature of refactoring is to not change the behavior of the code, but just. You're doing something for the developer's and for the future of the project to make it better, but it doesn't really. If you're a product owner that, you're not developing for the project, you don't really see the value in it. As a developer, you need to be able to convince the product owners. So there is always like the time pressure and then. " - Interviewee A

"First challenge is if the code was written by someone else and when there are no proper comments and documentation of what the particular function or code does, but then it's time-consuming. Since we need to understand what the code does. It can be difficult sometimes since it's not human readable sometimes." - Interviewee F.

The primary challenges in refactoring code include the significant time requirement and time pressure since developer's are working on numerous projects and tasks, making it difficult to prioritize refactoring. Additionally, convincing higher management of the value of refactoring can be difficult since it doesn't have an impact on the code's behavior but aims to improve maintenance. Finally, refactoring can be challenging when there is poorly documented code written by others, and this can take a lot of time to analyze and refactor the code base.

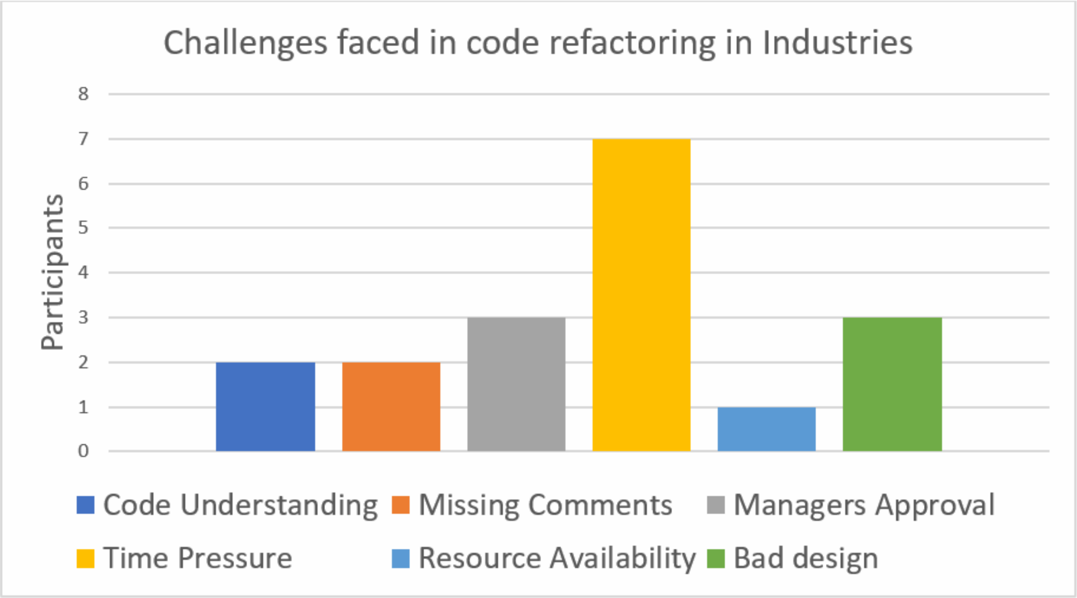


Figure 4.5: Challenges faced during code refactoring in Industries

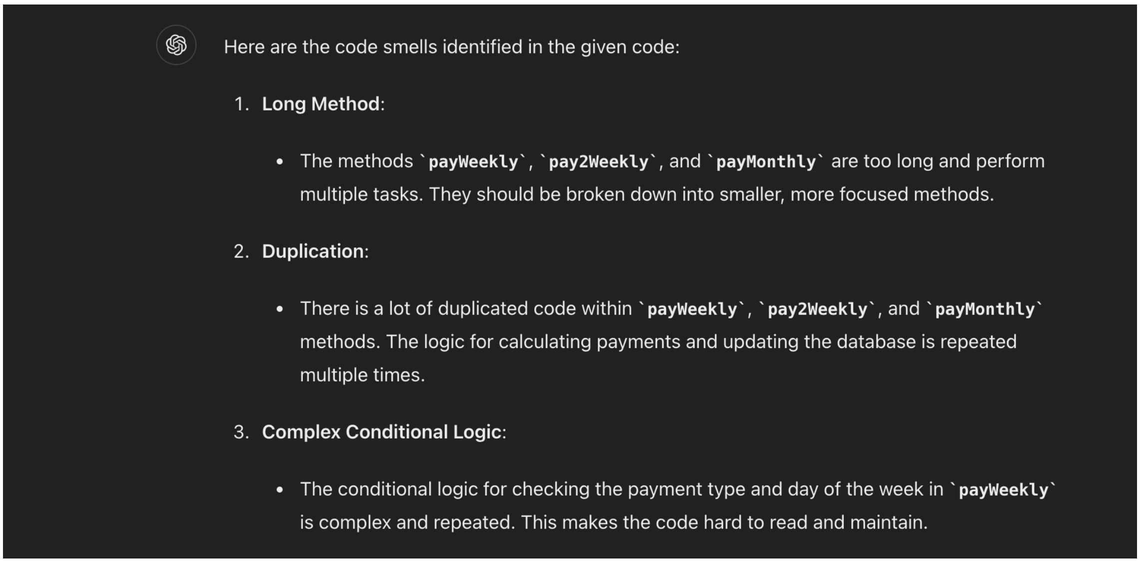
4.2 Understandability and Correctness of refactoring suggestions in addressing code smells - RQ2

We have explored the results concerning the effective prompt patterns incorporated to elicit refactoring suggestions from LLMs to solve code smells.
Upon analyzing the results in RQ1, we have discovered several prioritized code smells, including duplicated lines of code and long methods with complex conditional statements. We have opted to generate prompts and use a specific refactoring technique, such as extracting methods, to find a relevant refactoring suggestion for these code smells.

4.2.1 Observations on Prompt Patterns Results from ChatGPT

Our study of the optimized prompt patterns from ChatGPT showed us that versions 3.5 and 4.0 were quite different. The test file code which was refactored by ChatGPT 3.5 was not in line with the expected refactored file available in the GitHub repository. Contrarily, ChatGPT 4.0 gave a superior refactoring solution that was closer to the expected file. Yet, even though ChatGPT version 4.0 processes and improves the code in relation to the anticipated refactored file, both the versions had a common issue: refactored solutions had no comments. Therefore, there is a need for an additional prompt that would improve the readability and correctness of the code. Hence, we have chosen ChatGPT 4.0 to give the optimised prompts and captured the responses.

We first applied our optimized prompt from section 3.2.1 to the training data that we got from the GitHub repository. Next, for the purpose of detecting the code smells in the training data, SonarQube was used. Having analysed the training data, we then used the GitHub repository and the optimised prompt to revise this actual test data [19].
The figure 4.6 shows the response for the optimized prompt. *"From the above given code, identify and list the code smells."*



Here are the code smells identified in the given code:

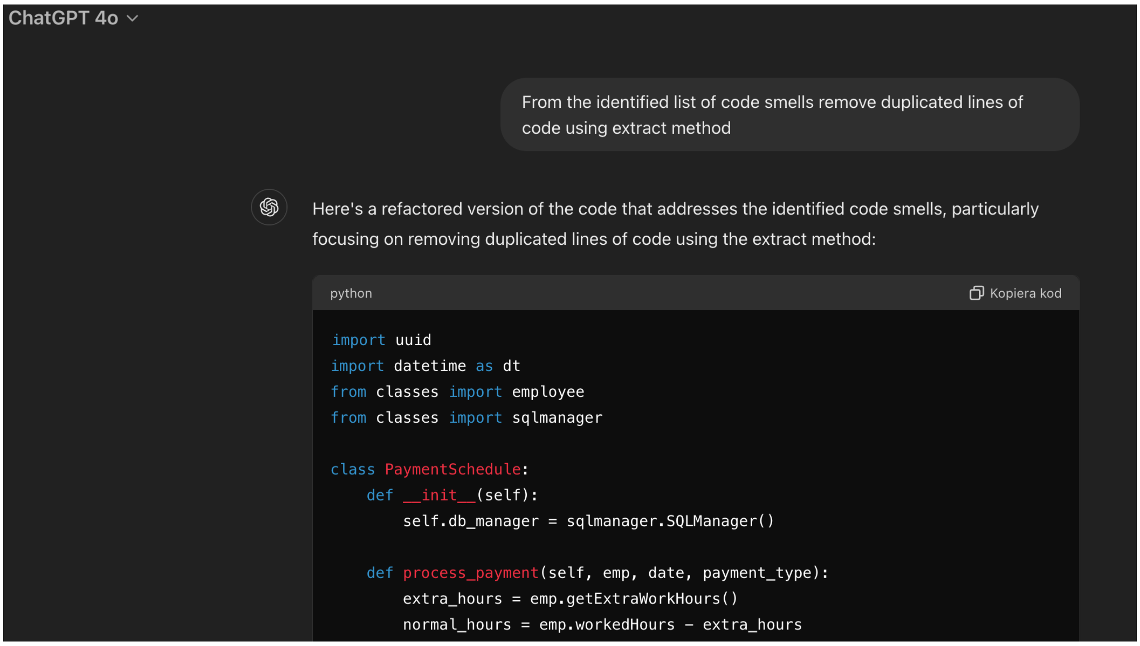
1. **Long Method**:
    - The methods `payWeekly`, `pay2Weekly`, and `payMonthly` are too long and perform multiple tasks. They should be broken down into smaller, more focused methods.
2. **Duplication**:
    - There is a lot of duplicated code within `payWeekly`, `pay2Weekly`, and `payMonthly` methods. The logic for calculating payments and updating the database is repeated multiple times.
3. **Complex Conditional Logic**:
    - The conditional logic for checking the payment type and day of the week in `payWeekly` is complex and repeated. This makes the code hard to read and maintain.

Figure 4.6: Identified list of code smells

The Figure 4.7 shows the responses for our second optimised prompt to remove duplicated lines of code.

The Figure 4.8 shows the comparison between excepted refactored file and Chatgpt refactored code for duplicated lines of code. In the expected refactoring file we notice that the developer extracts the print logic into a static method, thereby reducing redundancy in print statements which was present 7 times in the original code. Whereas in Chatgpt refactored code it produces an solution by combining payment calculation and printing it into a single method. Thereby removing duplication and improves readability and correctness.

The Figure 4.9 shows the comparison between excepted refactored file and ChatGPT refactored code for complex conditional statements. Both expected refactored file and chatgpt reduced the depth of nested conditionals. The expected refactored file

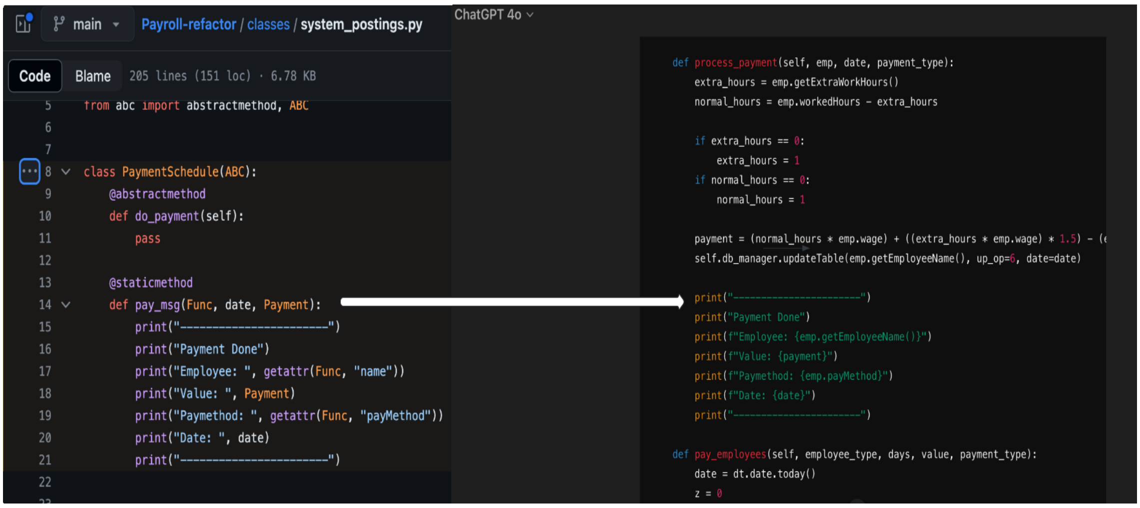


From the identified list of code smells remove duplicated lines of code using extract method

Here's a refactored version of the code that addresses the identified code smells, particularly focusing on removing duplicated lines of code using the extract method:

```python
import uuid
import datetime as dt
from classes import employee
from classes import sqlmanager

class PaymentSchedule:
    def __init__(self):
        self.db_manager = sqlmanager.SQLManager()

    def process_payment(self, emp, date, payment_type):
        extra_hours = emp.getExtraWorkHours()
        normal_hours = emp.workedHours - extra_hours
```

Figure 4.7: Optimized Prompt Pattern for Chatgpt 4.0

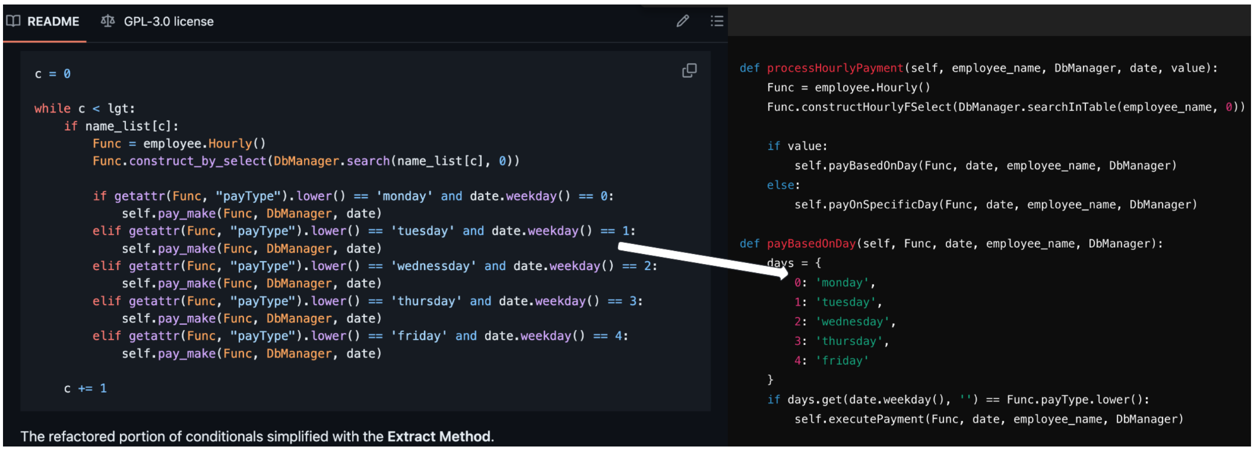


```python
from abc import abstractmethod, ABC


class PaymentSchedule(ABC):
    @abstractmethod
    def do_payment(self):
        pass

    @staticmethod
    def pay_msg(Func, date, Payment):
        print("----------------------")
        print("Payment Done")
        print("Employee: ", getattr(Func, "name"))
        print("Value: ", Payment)
        print("Paymethod: ", getattr(Func, "payMethod"))
        print("Date: ", date)
        print("----------------------")

```

```python
def process_payment(self, emp, date, payment_type):
    extra_hours = emp.getExtraWorkHours()
    normal_hours = emp.workedHours - extra_hours

    if extra_hours == 0:
        extra_hours = 1
    if normal_hours == 0:
        normal_hours = 1

    payment = (normal_hours * emp.wage) + ((extra_hours * emp.wage) * 1.5) - (
    self.db_manager.updateTable(emp.getEmployeeName(), up_op=6, date=date)

    print("----------------------")
    print("Payment Done")
    print(f"Employee: {emp.getEmployeeName()}")
    print(f"Value: {payment}")
    print(f"Paymethod: {emp.payMethod}")
    print(f"Date: {date}")
    print("----------------------")

def pay_employees(self, employee_type, days, value, payment_type):
    date = dt.date.today()
    z = 0
```

Figure 4.8: Comparing the responses between Git and ChatGPT 4.0 - 1

uses if-elif statements to check the day of the week. Whereas, ChatGPT uses a dictionary and a single method call to simplify the logic.



```
c = 0

while c < lgt:
    if name_list[c]:
        Func = employee.Hourly()
        Func.construct_by_select(DbManager.search(name_list[c], 0))

        if getattr(Func, "payType").lower() == 'monday' and date.weekday() == 0:
            self.pay_make(Func, DbManager, date)
        elif getattr(Func, "payType").lower() == 'tuesday' and date.weekday() == 1:
            self.pay_make(Func, DbManager, date)
        elif getattr(Func, "payType").lower() == 'wednessday' and date.weekday() == 2:
            self.pay_make(Func, DbManager, date)
        elif getattr(Func, "payType").lower() == 'thursday' and date.weekday() == 3:
            self.pay_make(Func, DbManager, date)
        elif getattr(Func, "payType").lower() == 'friday' and date.weekday() == 4:
            self.pay_make(Func, DbManager, date)

    c += 1
```

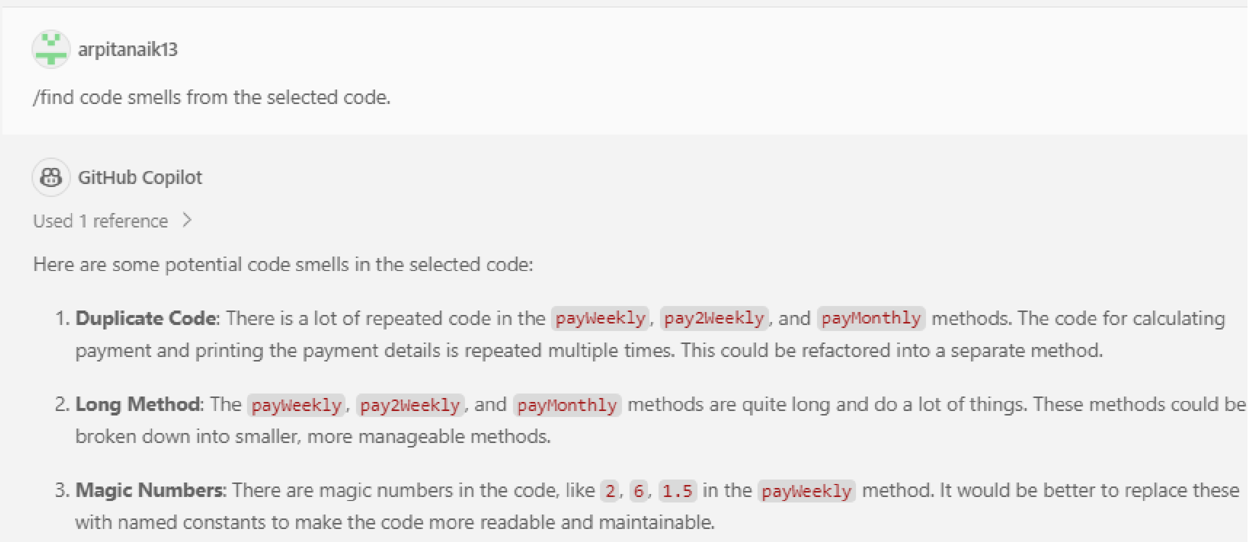
The refactored portion of conditionals simplified with the **Extract Method**.

```
def processHourlyPayment(self, employee_name, DbManager, date, value):
    Func = employee.Hourly()
    Func.constructHourlyFSelect(DbManager.searchInTable(employee_name, 0))

    if value:
        self.payBasedOnDay(Func, date, employee_name, DbManager)
    else:
        self.payOnSpecificDay(Func, date, employee_name, DbManager)

def payBasedOnDay(self, Func, date, employee_name, DbManager):
    days = {
        0: 'monday',
        1: 'tuesday',
        2: 'wednesday',
        3: 'thursday',
        4: 'friday'
    }
    if days.get(date.weekday(), '') == Func.payType.lower():
        self.executePayment(Func, date, employee_name, DbManager)
```

Figure 4.9: Comparing the responses between Git and ChatGPT 4.0 - 2

4.2.2 Observations on Optimised Prompt Patterns Results from GitHub Copilot

For the GitHub Copilot we have provided different prompts as it is crucial for obtaining diverse, comprehensive, and high-quality solutions.
Figure 4.10 shows the prompt provided and the response list of code smells identified in the code by Github Copilot.



arpitanaik13

/find code smells from the selected code.

GitHub Copilot

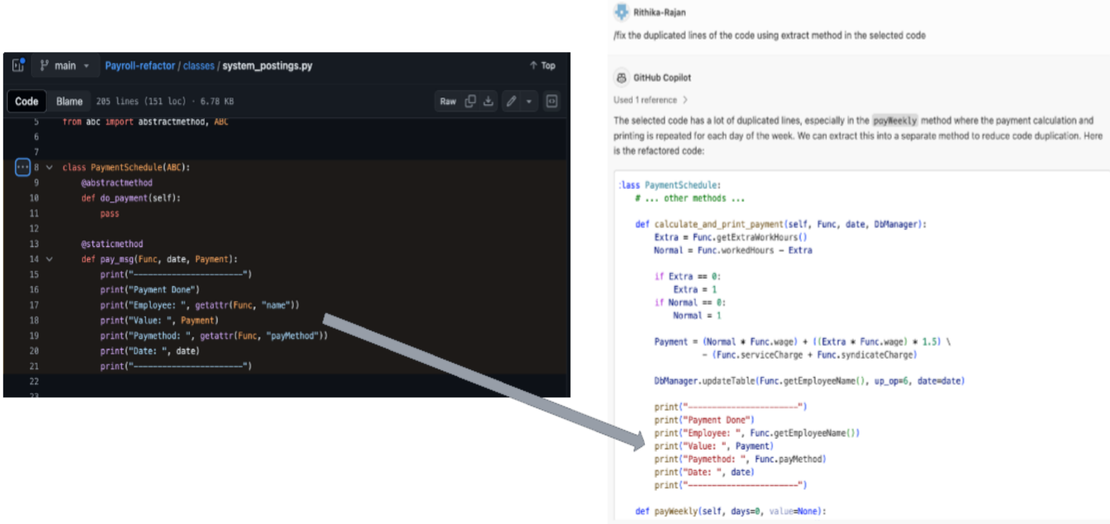
Used 1 reference

Here are some potential code smells in the selected code:

1. **Duplicate Code**: There is a lot of repeated code in the `payWeekly`, `pay2Weekly`, and `payMonthly` methods. The code for calculating payment and printing the payment details is repeated multiple times. This could be refactored into a separate method.
2. **Long Method**: The `payWeekly`, `pay2Weekly`, and `payMonthly` methods are quite long and do a lot of things. These methods could be broken down into smaller, more manageable methods.
3. **Magic Numbers**: There are magic numbers in the code, like `2`, `6`, `1.5` in the `payWeekly` method. It would be better to replace these with named constants to make the code more readable and maintainable.

Figure 4.10: Identified list of code smells in GitHub Copilot

The Figure 4.11 shows the comparison between excepted refactored file and Github copilot refactored code for duplicated lines of code. In the expected refactoring file we notice that the developer extracts the print logic into a static method, thereby reducing redundancy in print statements which was present 7 times in the original code. Whereas, in Github copilot refactored code it mentions that the selected code has a lot of duplicated lines, specifically mentioning the method payWeekly, where calculate and print payment method extracts as a separate method to reduce code duplication



```
from abc import abstractmethod, ABC


class PaymentSchedule(ABC):
    @abstractmethod
    def do_payment(self):
        pass

    @staticmethod
    def pay_msg(Func, date, Payment):
        print("----------------------")
        print("Payment Done")
        print("Employee: ", getattr(Func, "name"))
        print("Value: ", Payment)
        print("Paymethod: ", getattr(Func, "payMethod"))
        print("Date: ", date)
        print("----------------------")
```

Rithika-Rajan

/fix the duplicated lines of the code using extract method in the selected code

GitHub Copilot

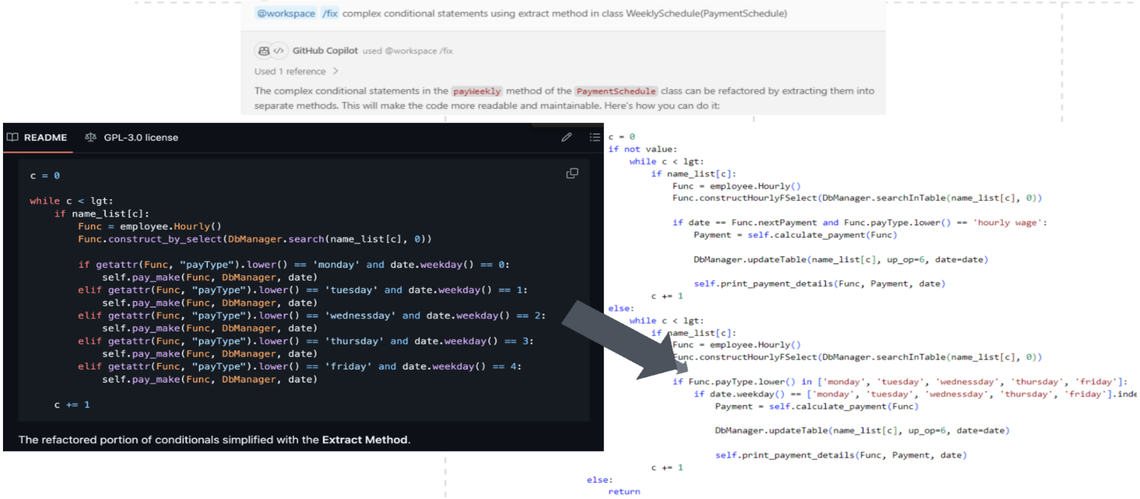
Used 1 reference

The selected code has a lot of duplicated lines, especially in the `payWeekly` method where the payment calculation and printing is repeated for each day of the week. We can extract this into a separate method to reduce code duplication. Here is the refactored code:

```
:lass PaymentSchedule:
    # ... other methods ...

    def calculate_and_print_payment(self, Func, date, DbManager):
        Extra = Func.getExtraWorkHours()
        Normal = Func.workedHours - Extra

        if Extra == 0:
            Extra = 1
        if Normal == 0:
            Normal = 1

        Payment = (Normal * Func.wage) + ((Extra * Func.wage) * 1.5) \
                  - (Func.serviceCharge + Func.syndicateCharge)

        DbManager.updateTable(Func.getEmployeeName(), up_op=6, date=date)

        print("----------------------")
        print("Payment Done")
        print("Employee: ", Func.getEmployeeName())
        print("Value: ", Payment)
        print("Paymethod: ", Func.payMethod)
        print("Date: ", date)
        print("----------------------")

    def payWeekly(self, days=0, value=None):
```

Figure 4.11: Comparing duplicated lines of code in GitHub Copilot

The Figure 4.12 shows the comparison between excepted refactored file and Github copilot refactored code for complex conditional statements. Both expected refactored file and Github copilot consolidates the checks into a single block, using helper methods for calculations and printing. Each method is found in an organized code structure, making it easier to understand.



@workspace /fix complex conditional statements using extract method in class WeeklySchedule(PaymentSchedule)

GitHub Copilot used @workspace /fix

Used 1 reference

The complex conditional statements in the `payWeekly` method of the `PaymentSchedule` class can be refactored by extracting them into separate methods. This will make the code more readable and maintainable. Here's how you can do it:

```
c = 0

while c < lgt:
    if name_list[c]:
        Func = employee.Hourly()
        Func.construct_by_select(DbManager.search(name_list[c], 0))

        if getattr(Func, "payType").lower() == 'monday' and date.weekday() == 0:
            self.pay_make(Func, DbManager, date)
        elif getattr(Func, "payType").lower() == 'tuesday' and date.weekday() == 1:
            self.pay_make(Func, DbManager, date)
        elif getattr(Func, "payType").lower() == 'wednessday' and date.weekday() == 2:
            self.pay_make(Func, DbManager, date)
        elif getattr(Func, "payType").lower() == 'thursday' and date.weekday() == 3:
            self.pay_make(Func, DbManager, date)
        elif getattr(Func, "payType").lower() == 'friday' and date.weekday() == 4:
            self.pay_make(Func, DbManager, date)

    c += 1
```

The refactored portion of conditionals simplified with the **Extract Method**.

```
    c = 0
    if not value:
        while c < lgt:
            if name_list[c]:
                Func = employee.Hourly()
                Func.constructHourlyFSelect(DbManager.searchInTable(name_list[c], 0))

                if date == Func.nextPayment and Func.payType.lower() == 'hourly wage':
                    Payment = self.calculate_payment(Func)

                    DbManager.updateTable(name_list[c], up_op=6, date=date)

                    self.print_payment_details(Func, Payment, date)
            c += 1
    else:
        while c < lgt:
            if name_list[c]:
                Func = employee.Hourly()
                Func.constructHourlyFSelect(DbManager.searchInTable(name_list[c], 0))

                if Func.payType.lower() in ['monday', 'tuesday', 'wednessday', 'thursday', 'friday']:
                    if date.weekday() == ['monday', 'tuesday', 'wednessday', 'thursday', 'friday'].inde
                        Payment = self.calculate_payment(Func)

                        DbManager.updateTable(name_list[c], up_op=6, date=date)

                        self.print_payment_details(Func, Payment, date)
            c += 1
else:
    return
```

Figure 4.12: Comparing complex conditional statements with Copilot

4.2.3 Understandability and Correctness of code refactoring suggestions compare between different LLMs - RQ2.1

Compared based Readability and Understandability:
ChatGPT: According to the result, the refactoring solutions offered by ChatGPT 4.0 were more precis when compared with those of ChatGPT 3.5 against developer's refactored code from section 4.2.1. However, improved code for the two versions did not include documentation or comments, which made it difficult to read.
GitHub Copilot: GitHub Copilot recognised repeated lines of code and complex conditional statements and refactored them efficiently, resulting in a better outcome for reducing duplication and improving comprehension.

Compared based on Prompts:
ChatGPT: Results were better when the prompts were optimized. For instance, the prompt "Remove duplicated lines of code using extract method" was more specific than a generic prompt.
GitHub Copilot: We found that Github Copilot needed more detailed instructions, like the names of methods or classes, to be able to give exact results. Also, it came to our attention that We came to understand that good practices in prompt optimization enhances suggestion quality significantly. The table 4.1 shows the summarised results of comparison between LLMs indicating from 1 to 5 scale, where 5 is the highest and 1 being the lowest score in each category.

Category	ChatGPT 3.5	ChatGPT 4.0	GitHub Copilot
Refactored code quality	2	4	4.5
Documentation/Comments	2	2	3.5
Duplicated lines	3	4.5	4.5
Complex conditional statements	3.5	4	4.5
Readability and understandability	2.5	3	3
Prompt optimization techniques	3	4.5	4.5
Performance in addressing code smells	2.5	4.5	4.5
Comparison with expected refactoring	3	4.5	4.5

Table 4.1: Summarised results of comparison between LLMs.

4.2.4 Understandability and Correctness of code refactoring suggestions provided by LLMs compared to those given by human experts - RQ2.2

Below shows the results from comparison of two human experts from two different tasks based on understandability and correctness. The below categories are divided based on the interview questions given for the experts:

Task 1- File1 with code smells

1. Understandability of Methods and Functions
According to the experts "File C was most easy to understand, as they also noticed the usage of environment variables and the fact that they were at the beginning of the file and upper-case of naming convention was constant. The code was given in Good structure only one issue was that it was lacking type hints in SQLManager".

2. Variable and Method Naming
According to the experts "File F, C & E had a good variable and method naming(semantics and PEP8 compliance)" Other files had the method names wrong or not correct such as persoschedule/personschedule instead of PersonalSchedule therefore naming convention was not followed. File B also followed a proper method and variable naming except for few variable naming such as execute'_'transation - this should have been prefixed _ to show it's private method.

3. Breaking Down Methods into Smaller, Manageable Sections - To check which file have refactored long methods
According to experts File B and C had Good breakdown of long methods to separate section as transaction logic from intended action, they also had better definitions of methods whereas File E don't have methods at all. File A methods are not optimised.

4. Minimization of Complex Nested Structures
According to the experts File C had minimized complex nested structure the code was clean, It had optimised loop and implementation, separating transaction logic as in File B, but File B took a little more effort to read whereas File F has no methods defined.

5. Reduction of Duplicated Code
According to the experts File B and File F has reduced lines of duplications.

6. Correctness of Functionality
According to the experts File B, C & F has the correct functionality. Although none of them are complete they have missing search function etc - but these are the closest ones.

7. Feedbacks on other files
Feedbacks regarding all the files According to the experts DB authentication was often hard-coded, Entity fields should be defined at class-level and not overridden in _init_, No files have implemented search_in_table method except for in File A , type hints were missing or optional but preferable for code accessed from outside IMHO. They noticed that File A, C, D & E had many duplicate lines of code, Overall File C has the best implementation of methods and File F has the best class definitions.

Observations from task1 results
Based on the results from task 1 we conclude that developer's refactored File C

was understandable, code was well-structured, and had the best method implementation. ChatGPT refactored file B demonstrated good method breakdown and separation of logic, improving code clarity as similar as developer refactored File C. Both ChatGPT and Github Copilot refactored files were able to successfully minimize duplicated lines of code. Both ChatGPT and Github Copilot refactored files along with a developer refactored file had the correct functionality but lacked some features. Copilot refactored File had better class definitions.

Task 2- File2 with code smells

1. Understandability of Methods and Functions

According to the experts File E was easier to understand, Usage of environment variables and good coding structure.

2. Variable and Method Naming

According to the experts File E had a good variable and method naming.

3. Breaking Down Methods into Smaller, Manageable Sections

According to experts File B and E had better breakdown of long method to subsection.

4. Minimization of Complex Nested Structures

According to the experts File B and E, they notice the use of update_operation_mapping method to minimize nested structure.

5. Reduction of Duplicated Code

According to the experts File E reduced duplicated lines of code they notice that even though there were incomplete functions from update_operation_mapping method, the pattern is nice.

6. Correctness of Functionality

According to the experts File B was better compared to other files as other files were really hard to read out what the original functionality intends to do and few functions are not implemented by file D and E .

7. Feedbacks on other files

According to experts File A, File C, File D and F has too many loops in methods hampering readability and therefore increasing maintenance. The method is not atomic in nature. Experts also notice that a conditional statement was confusing if up_op in range(0, 8): which also introduces a bug causing up_op 8 & 9 or above numbers would never be executed.

Observations from task2 results

Based on the results from task 2, we conclude that developer-refactored File E was the easiest to understand, with good code structure, correct variable and method naming and use of environment variables. ChatGPT-refactored File B was able to refactor the long method into subsections and minimized complex nested conditions, improving code readability, similar to developer-refactored File E. ChatGPT refactored file B had functionalities similar to the original functionality intend to perform. However, GitHub Copilot refactored file F and other developer files A, C and D had multiple loops in the method therefore making it hard to understand in task 2. The given table 4.2 shows the summary of the results.

Category	Task 1 (File 1)	Task 2 (File 2)
Understanability of methods and functions	File C	File E
Variable and method naming	File F,C,E	File E
Breaking down long methods	File B,C	File B,E
Minimization of complex nested structure	File C	File B,E
Reduction of duplicated code	File B,F	File F
Correctness of functionality	File C,B,F	File B

Table 4.2: Summarised results between two tasks.

5

Discussion

5.1 Research Questions Summary

RQ1: What refactoring patterns do Developer's use to address various types of code smells?
Findings from the RQ1 indicate that developer's focus more on the code smells that have an adverse effect on the readability, understandability and general structure of the code. The most prioritized code smells were found to be duplicated lines of code, complex conditional statements and long methods. During the process of refactoring the code, developer's commonly used the Extract Method, renaming fields and moving functions to make the code modular and easier to handle. Such patterns help in simplifying complex methods into smaller relative easier to understand methods. Refactoring of the code helps achieve this in that it enables a developer to alter the structure of the code in a bid to ease maintenance and extensions of the code without altering its functionalities.

The developer's found difficulties when working with large classes and repetitive loops. For example, there were specific code smells such as large class and repetitive loops methods that made it hard to enhance and refactor the code due to the lack of knowledge of the other methods in the code. Overall reducing complexity was the common goal of the developer's, yet it was also the continuously standing issue because of lack of time which made refactoring processes slow. In addition, we observe that they have been using tools such as ChatGPT and GitHub Copilot for coding purposes during the software development phase [4].

RQ2: How correct are refactoring suggestions in addressing code smells?
The results show that LLMs were helpful for addressing code smells such as duplicated code, long methods, and nested conditions. These tools were used by employing prompt patterns to refactor the given code using Extract Method, this improved both understandability and correctness of the code. We observe that ChatGPT was productive in removing conditional statements , and GitHub Copilot had improved versions of class definitions. When compared to ChatGPT, Copilot chooses small portion of the code from the whole code base. We also observed that even though LLMs offered suggestions, their refactoring files were not always fully complete when compared with developer's. developer's files showed a better understanding of the overall code structure, particularly when it came to a bit more complex refactoring. We also noticed that LLMs sometimes missed some functionality or failed to address

bugs introduced by incorrect conditions.

Overall, by this comparison, we understood that even though LLMs are capable of addressing the simple code smells, they lack an in-depth understanding of the code structure; therefore, it is necessary for the developer's to validate the refactored code to ensure the functionality is maintained.

The study conducts by researches [4] notes that to utilize LLMs such as ChatGPT in software development effectivety, it is necessary to prepare a collection of prompts and how to apply them throughout the software life cycle. However, there is still a big gap because LLMs, including ChatGPT, give incorrect content. While prompt patterns can have some problems, there is a need for more improvement in quality control measures in prompt repositories to make LLM outputs reliable and beneficial in software engineering settings.

The same study [4] also noticed that When there are reported code snippets that need to be refactored, ChatGPT fails in understanding such fragments or gets confused when only a part of code and not complete code is provided, particularly if the code is too large. Even its corrections did not misunderstood the purpose of an excerpt and would slightly alter the logic. Hence, the more complex the code input that is provided, the higher the chances that the code output will not work properly. By underlining issues and patterns associated with these conversations on refactoring, researchers and developer's can identify areas in which ChatGPT can be enhanced. This knowledge can be useful in designing the model in such a manner that it meets the requirements of developer's more effectively.

5.2 Future works

More Research on Prompt Engineering for LLMs: Future research could explore more improved prompts in particular to refactoring to refine or get an improved LLM outputs. We also noticed that both ChatGPT and GitHub Copilot require different sets of prompts. This research helps us understand that when we provide clear, simple and detailed prompts, the output from the improved and, thereby increasing the performance of LLM.

Research to widen the scope to other Programming Languages: Our master thesis focused only on Python. In Future this research can extend to many other programming languages like C++, C-Sharp, Javascript and many more. This could require examining whether LLMs in these programming languages work similarly when refactoring is performed.

Real-Time Collaboration Between LLMs and Developers: More research should be conducted in understanding the communication between Developer's and LLM how developer's and LLMs can work hand in hand in real-time scenarios, how we can integrate integrating LLMs to daily coding and how it practised in current software development Research could focus on developing different communication patterns that allow Developer's to use LLM suggestions during active coding, also understanding if the suggestions are valid or not.

Improving LLM Training for Refactoring Tasks: The Future work includes more accurate instructions on more specialized data for LLMs which focuses on code refactoring patterns; improvement on tasks such as reducing duplicated lines of code or improving method breakdown. Future work should develop the various data of real-world codebases that help to enhance adaptability across projects.

5.3 Threats to validity

In this thesis, there are several factors that might compromise the results validity, which include internal, external and construct validity issues.

5.3.1 Internal validity

The thesis primarily focuses on the investigation of LLM's which makes the study different from the studies analyzing general software systems. However, it also has some internal validity threats due to participants' backgrounds and experience which is diverse as well as the possibility of the bias of the researches. Our thesis considers ChatGPT, which may not be used in many industries for security reasons. The diversity of participants may be beneficial in answering RQ1 however, different levels of experience and expertise in the Python domain may raise internal validity issues. When developer's were refactoring the code manually, they were also given prompts for LLMs such as ChatGpt and GitHub Copilot. Biases may arise in the results obtained due to differences in the level of instruction clarity between human developer's and LLMs as we provide proper prompts. In cases where developer's who refactor manually readily deviate, the findings may be inconsistent with the introduction of LLMs in the comparison. If the raters of the refactored code carry out their assessment with some bias, it will affect the comparison of LLM and human-based refactoring.

5.3.2 External validity

We could not generalize our findings since we have limited our research only on Python projects. Our research participants are working within Sweden. The Python code repositories with code smells which was discovered on 'GitHub,' is an open source, which can be used to train the models, which we were aware of, it would increase the external validity of this research.
Since a very few number of developer's and experts were involved in the refactoring process of our research, the results might cannot be generalized to a larger population of software engineers. The study aims on particular types of code smells, which may not represent all refactoring cases encountered in the industry.

5.3.3 Construct validity

Construct validity has to do with whether the study is measuring what it is intended to measure. This, however, is affected by several potentials. One of the issues can be the assessment of the success of refactoring. If this were to be solely grounded on the expert opinion without looking at objective measures such as how well the code performs or how maintainable it is, then it may lead to biases . What is more, the indicators of understandability or correctness assessment can be defined as well, where evaluations from experts who have experience for 13 years has validated this assessment. Another rather misleading factor might be the operational requirements of the LLMs provided, in which case how productive the outputs from the refactored system will depend on how precise the LLMs were interpreted. The inherent development of other LLMs, such as ChatGPT and Github Copilot, causes them a drawback in completing a refining task, thereby calling entirely different attention to the human versus LLM refreshing code comparison as made earlier.

6

Conclusion

Our thesis examines the correctness of LLMs like ChatGPT and GitHub Copilot for removing the code smells and provides refactoring suggestions for making the code quality better in Python projects. In this study, it was discovered that LLMs were capable of recognizing and proposing refactoring suggestions for some of the less complex code smells, such as complex conditional statements and duplicated lines of code. However, more often they fail to understand when compared with developer's, who would typically perform better when it comes to complex refactoring for complex code structure. This limitation implies a need for developer validation in order to maintain the functionality after refactoring. The research further notice that enhancement of prompt engineering can yield better results from LLMs, pointing to promising areas of future work and utilization.

This thesis work provides insights for both researchers and practitioners working in the context of software development and code refactoring. To researchers, it provides information about the use of Large Language Models (LLMs) such as ChatGPT and GitHub Copilot for code refactoring in Python, which is less addressed in the literature compared to other programming languages like Java. While the current study inspects some research in prompt engineering and makes a path for future research, such as the improvement of prompt engineering methods, extending the refactoring tools to other programming languages, and real-time working interaction of LLMs with the developer's. On the practitioners side, the LLMs have offered solutions to assist software engineers on code refactoring, which include examples like repetition in lines of codes, shortening lengthy function codes, and enhancing overall code's understandablity and correctness. While it does inform professionals about the advantages and disadvantages of applying LLMs to refactoring projects, especially on a Python project, it also proposes ways in which better code can be achieved without excessive LLM participation, especially validation of the changes made by the LLM.

To summarize, although LLMs serve as a useful addition to efforts directed towards refactoring a codebase, they still have no checkers for more complex code smells. Including LLMs in daily development practices, as well as development in prompt engineering has great hope in the future of software development, particularly for minimizing technical debt and improving code maintainability.

Bibliography

[1] L. Guilherme , F. Petrilloc, M. Pimenta and Y. G. Guéhéneuc, (2020) “Code smells and refactoring: A tertiary systematic review of challenges and observations,” The Journal of Systems and Software, p. 36.

[2] V. Lenarduzzi, F. Lomio, H. Huttunen and D. Taibi. (2018) “Are SonarQube Rules Inducing Bugs?,” p. 11.

[3] V. Alizadeh„ M. A. Ouali„ M. Kessentini and M. Chater, 2019 “RefBot: Intelligent Software Refactoring Bot,” p. 12.

[4] E. A. AlOmar, A. Venkatakrishnan, M. W. Mkaouer, C. D. . Newman and A. Ouni. 2024 “How to Refactor this Code? An Exploratory Study on Developer-ChatGPT Refactoring Conversations,” p. 6.

[5] J. White, S. Hays, Q. Fu, Spencer-Smith and D. C. Schmidt, “ChatGPT Prompt Patterns for Improving Code Quality, Refactoring, Requirements Elicitation, and Software Design,” 2023.

[6] A. Almogahed and M. Omar, “Refactoring Techniques for Improving Software Quality: Practitioners’ Perspectives,” Journal of Information and Communication Technology, 2021.

[7] E. A. AlOmar„ M. W. Mkaouer, and A. Ouni„ “Behind the Intent of Extract Method Refactoring,” A Systematic Literature Review, p. 26.

[8] M. Alharbi and M. Alshayeb, “A Comparative Study of Automated Refactoring Tools,” 2024.

[9] H. Atwi, “PYREF: Refactoring Detection in Python Projects,” p. 6.

[10] M. kasunic. (2004) "Conducting Effective Pilot study," Carnegie Mellon Software Engineering Institute, p. 40, 2004.

[11] O. A. Adeoye-olatunde and N. L. Olenik. (2021) "Research and Schonarly methods: Semi- structured Interviwes," Journal of American college of clinical pharmacy, p. 10

[12] A. N. I. and Y. Nie. (2017) "Combining narrative analysis, ground theory and qualitative data analysis software to develop a case study research," Journal of managements research, p. 19.

[13] https://refactoring.com/catalog/ - Accessed refactoring patterns - Figure 3.2

[14] Pranab Sahoo1 , Ayush Kumar Singh , Sriparna Saha , Vinija Jain , Samrat Mondal1 and Aman Chadha (2024) "A Systematic Survey of Prompt Engineering in Large Language Models: Techniques and Applications," p. 9

[15] Louie Giray1 (2023) "Prompt Engineering with ChatGPT: A Guide for Academic Writers"Annals of Biomedical Engineering (2023), p. 5

[16] Yongchao Zhou, Andrei Ioan Muresanu, Ziwen Han, Keiran Paster, Silviu Pitis, Harris Chan, Jimmy Ba1(2023) "LARGE LANGUAGE MODELS ARE

HUMAN-LEVEL PROMPT ENGINEERS" , conference paper at ICLR 2023, p. 43

[17] David OBrien, Rabe Abdalkareem, Sumon Biswas, Emad Shihab, Sayem Mohammad Imtiaz, Hridesh Rajan (2024) "Are Prompt Engineering and TODO Comments Friends or Foes? An Evaluation on GitHub Copilot", p.14

[18] Nalin Wadhwa, Jui Pradhan, Atharv Sonwane, Surya Prakash Sahu, Nagarajan Natarajan, Atharv Sonwane, Aditya Kanade, Suresh Parthasarathy, Sriram Rajamani,(2023) "Frustrated with Code Quality Issues? LLMs can Help!", p.19

[19] https://github.com/JT4v4res/Payroll-System/tree/main

[20] Sabit Ekin , (2023) "Prompt Engineering For ChatGPT: A Quick Guide To Techniques, Tips, And Best Practices"

[21] Beiqi Zhang, Peng Liang, Qiong Feng, Nanjing, Yujia Fu, Zengyang Li ,(2024)"Copilot Refinement: Addressing Code Smells in Copilot-Generated Python Code" p.5

[22] Claes Wohlin, Martin Höst, Björn Regnell, Per Runeson, Magnus C. Ohlsson, Anders Wesslén Ericsson, "Experimentation in Software Engineering Second Edition"

[23] Martin Fowler, contribution by Kent Beck, "Refactoring Improving the design of Existing Code"

[24] Amy J. Ko, Thomas D. LaToza, Margaret M. Burnett, (2013) "A practical guide to controlled experiments of software engineering tools with human participants"

[25] Talita Vieira Ribeiro and Guilherme Horta Travassos, (2015) "On the Alignment of Source Code Quality Perspectives through Experimentation: An Industrial Case"

A

Appendix 1

Interview Questions:

1. Professional introduction of the developers, their roles are and responsibilities in their day to day work. How many years of experience do you having in python developing.
2. Can you describe your experience with identifying and addressing code smells in software development?
3. When there are multiple code smells in a code, how do you prioritise which ones are important to address first?
4. Could you think of any recent example of a code smell you encountered and how you decided on the most suitable refactoring solution?
5. What do you consider when selecting a refactoring pattern for a particular code smell?
6. How do you ensure that the chosen refactoring solution works with the overall design goals and architecture of the software ?
7. Which are the tools which you commonly use for refactoring? What are the Advantages and disadvantages of these refactoring tools ?
8. Have you used any LLM during your refactoring process, Can you mention which of them? Advantages and disadvantages of them.
9. In your opinion, How do you evaluate the correctness of the refactoring?
10. Finally to wrap up this interview, are there any challenges that you faced when dealing with code smells and refactoring and how did you overcome them?

Interview questions for experts for Task 1 & Task 2
1. From which refactored file was the function and method easy to understand?
A to F file names (choose one or more)

2. Which refactored file had followed the variable/method names which are easier to understand?
A to F file names (choose one or more)
Motivation: We want to check if the developers and LLM provided names are readable and understandable.

3. Which refactored file had methods broken down into manageable and reusable smaller sections?
A to F file names (choose one or more)

Motivation: We want to check which file have refactored long methods.

4. Which refactored file minimized complex nested structures (e.g., loops or conditionals) that were difficult to read?
A to F file names (choose one or more)

5. Which refactored file minimized duplicated lines of code?
A to F file names (choose one or more)

6. Which refactored file correctly implements the intended functionality based on the original code? Choose any 2
Motivation: To check the refactored file can correctly implement the functionality.

7. Issues or problems found with other refactored files?