# IMPROVING THE MAINTAINABILITY OF COMMERCIAL SOFTWARE

A THESIS IN
Information Technology

Presented to the Faculty of the Deakin University
in partial fulfillment of the requirements for the degree

MASTER OF INFORMATION TECHNOLOGY (PROFESSIONAL)

by
SIDDHANT GUPTA

M.I.T., Deakin University, Melbourne, Australia, 2024

Melbourne, Australia
2024

## ABSTRACT

Maintaining commercial software's quality and adaptability is crucial as systems grow increasingly complex. This thesis explores methods to enhance the maintainability of popular open-source software by focusing on reusability, which is often overlooked in traditional software development. The research investigates the effectiveness of architectural patterns and refactoring techniques, as well as the potential of machine learning and generative AI tools to improve code maintainability and reduce technical debt. Through an in-depth analysis of large codebases, this study aims to provide actionable recommendations for software developers and industry professionals, contributing to the ongoing efforts to make software systems more sustainable and cost-effective.

## ACKNOWLEDGMENTS

I would like to express my deepest gratitude to my supervisor, Professor Roopak Sinha, for his continuous support, guidance, and encouragement throughout this research. I am also grateful to Professor Manzur Murshed, my unit chair, for his insightful feedback and support. Additionally, I extend my thanks to the faculty at Deakin University, my fellow students, and my family and friends who have supported me in various ways during this journey.

CONTENTS

# CHAPTER 1

# INTRODUCTION

This thesis investigates techniques to improve commercial software's maintainability with an emphasis on reusability, a vital but sometimes disregarded component of conventional development. The project intends to minimise technical debt and enhance code quality through the use of generative AI and machine learning technologies, architectural patterns, and refactoring approaches. The study looks at well-known open-source projects and offers practical advice on how to improve the long-term viability, scalability, and sustainability of software.

## 1.1 Background and Motivation

Long-term adaptability and success of a system depend heavily on software maintainability, particularly in commercial environments where frequent and fast changes are expected. Software maintenance is harder as it gets more complicated, which raises expenses, causes technical debt, and lowers productivity. This thesis focusses on reusability, which is frequently disregarded in traditional development, in order to improve the maintainability of well-known open-source software. It looks at refactoring strategies, architectural patterns, and the application of generative AI and machine learning to enhance code maintainability with the goal of enhancing the software systems' scalability, sustainability, and affordability.

## 1.2 Problem Statement

Even while maintainability is acknowledged as a crucial feature of commercial software, the problem of reusability is frequently not sufficiently addressed by current approaches. This gap in the body of knowledge and industry standards emphasises the necessity of a thorough strategy to improve maintainability. In order to decrease technical debt and lengthen the lifespan of software systems, this project aims to investigate and use practical methods for enhancing reusability in software development.

## 1.3 Research Questions

- **Exploration**
  - What are the base criteria for measuring reusability in commercial software?
    - What factors influence the reusability of software components?
    - How is reusability currently measured in existing literature and industry practices?
- **Design**
  - What architectural patterns and refactoring techniques are most effective in enhancing software reusability?
    - Which architectural patterns are commonly used to promote reusability?
    - How do different refactoring techniques impact the reusability of software components?

- **Evaluation**
  - How can machine learning and generative AI tools be leveraged to improve reusability in software development?
    - What specific machine learning techniques can aid in identifying reusable code segments?
    - How can generative AI tools assist in writing more reusable and maintainable code?

## 1.4 Objectives and Scope

This research's major goal is to create and verify methods that increase commercial software's reusability and maintainability. The study will concentrate on examining well-known open-source software projects to find reusability issues and suggest fixes. The use of generative AI and machine learning technologies, architectural patterns, and refactoring strategies are all included in the scope.

## 1.5 Significance and Contributions

This study advances the area of software engineering by offering a thorough examination of the variables influencing reusability and by putting up creative ideas for improving maintainability. Academic researchers and experts in the sector will find value in the results, which provide actionable advice on enhancing the calibre of commercial software. This thesis seeks to decrease technical debt, minimise maintenance costs, and increase software system sustainability by filling in the gaps in existing approaches.

# CHAPTER 2

# REVIEW OF LITERATURE

## 2.1 Overview of Software Maintainability

The ease with which a software system may be altered to fix bugs, enhance performance, or adjust to a changing environment is known as software maintainability. It is a crucial component of software quality that has an impact on the systems' long-term viability and affordability. As to ISO/IEC 25010:2011, maintainability comprises many sub-characteristics such as testability, reusability, modularity, analyzability, and modifiability.

Software maintainability has historically been the subject of more complex methods that address the underlying architecture and design of software systems rather than just basic code readability and documentation. The focus of contemporary maintainability techniques is on the value of automation, refactoring, and architectural patterns in lowering technical debt and guaranteeing the long-term sustainability of software systems.

## 2.2 Reusability in Software Development

The ability to utilise software components in many applications or settings with little to no modification is known as reusability. Reusable components minimise the need for redundant code, make maintenance tasks easier, and encourage consistency across software systems, making them essential to achieve maintainability.

In a variety of settings, such as object-oriented programming, component-based development, and service-oriented architecture, reusability has been investigated. The Gang of Four established the idea of design patterns, which has had a big impact on how developers see reusability. Design patterns make it simpler to construct reusable and maintainable software components by offering tried-and-true answers to common design issues.

Reusability is important, but in reality it can be difficult to achieve. There are a number of things that can make software components less reusable, including close coupling, low modularity, and poor documentation. In addition, developers find it challenging to evaluate and enhance the reusability of their code due to the lack of common metrics for assessing reusability.

## 2.3 Architectural Patterns for Maintainability

Architectural problems in software systems play a crucial role in enhancing maintainability by promoting modularity, reducing complexity, and enabling better separation of concerns. Common architectural patterns that promote maintainability include:

- **Layered Architecture:** Divides the system into layers with distinct responsibilities, making it easier to isolate changes and manage dependencies.
- **Microservices Architecture:** Breaks down the system into loosely coupled services that can be developed, deployed, and maintained independently.
- **Model-View-Controller (MVC):** Separates the system into three componentsâmodel, view, and controllerâto facilitate modularity and ease of maintenance.

Each of these patterns offers specific benefits in terms of reusability and maintainability, but their effectiveness depends on the context in which they are applied. The choice of architectural pattern must consider the specific requirements of the software system, including its size, complexity, and intended lifespan.

## 2.4 Refactoring Techniques and Their Impact

Refactoring is the process of restructuring existing code without changing its external behavior. It aims to improve the internal structure of the code, making it more readable, maintainable, and reusable. Refactoring is a critical practice in software development, as it helps manage technical debt and prevents code decay. Common refactoring techniques include:

- **Extract Method:** Simplifies complex methods by breaking them down into smaller, more manageable methods.
- **Rename Variable:** Improves code readability by giving variables meaningful names that reflect their purpose.
- **Move Method:** Enhances modularity by relocating methods to the appropriate class or module.
- **Replace Magic Number with Symbolic Constant:** Increases code clarity by replacing hard-coded values with named constants.

Empirical studies have shown that refactoring can significantly improve the maintainability and reusability of software components. However, refactoring requires careful planning and execution to avoid introducing new defects or compromising the functionality of the system.

## 2.5 Role of Machine Learning and Generative AI in Software Development

Software development might undergo a revolution thanks to machine learning and generative AI tools, which can automate repetitive operations, find patterns in code, and produce high-quality code segments. There are several ways in which these technologies may be utilised to improve reusability and maintainability.

Code recommendation systems and anomaly detection are examples of machine learning approaches that can help developers find reusable code segments and provide suggestions for enhancements. By making intelligent recommendations and automating tedious operations, generative AI toolsâsuch as code generation models and automated refactoring toolsâcan assist developers in writing more reusable and maintainable code.

Even though software development is still in its infancy, artificial intelligence has a lot of potential advantages. Developers may lessen technical debt by incorporating machine learning and generative AI technologies into the development process.

# CHAPTER 3

# PROPOSED SOLUTION

This chapter describes the methods that have been suggested to address the research issues presented in Chapter 1 and enhance the maintainability of commercial software. The strategies concentrate on improving reusability by utilising generative AI and machine learning technologies, refactoring approaches, and architectural patterns. These ideas offer workable strategies for enhancing the longevity and quality of software systems while addressing the issues raised in the literature study.

## 3.1 Enhanced Documentation Generator with LLM Integration

### 3.1.1 Concept

The Enhanced Documentation Generator is a program made to automatically provide thorough documentation for every feature and part of a codebase. This program does a complete codebase scan, gathers data from code annotations and structure, and transforms the data into understandable, comprehensive documentation. Function names, descriptions, arguments, return types, exceptions, and use examples are all included in the documentation. This technology is unique in that it integrates with a large language model (LLM), which enables users developers in particular to query the documentation in a natural language to locate certain components, functions, or other pertinent data.

### 3.1.2 Use

To provide prepared documentation that developers can access quickly, the Enhanced Documentation Generator searches the codebase and pulls pertinent information from annotations and docstrings. The LLM integration, which enables more natural user interaction with the documents, is the main novelty, though. Users may ask LLM specific questions like ”How do I use the authentication function?” and ”Which components handle data validation?” without of having to go through the documentation by hand. After parsing the question and searching the documentation, the LLM will supply the pertinent data.

Depending on the needs of the project, a cron job is configured to run either daily or weekly to guarantee that the documentation is kept current. The Enhanced Documentation Generator is triggered by this cron task to rescan the codebase and update the documentation accordingly. By ensuring that the documentation matches the most recent changes made to the source, this automated updating system lowers the possibility of updated information and upholds strict requirements for code quality.

### 3.1.3 Implementation

1. **Code Scanning:**
    - Scan the entire codebase for functions, methods, and components.
    - Extract information from code annotations and docstrings.
2. **Documentation Generation:**

- Use a template to format the extracted information into readable documentation.
- Include sections for function descriptions, parameters, return types, exceptions, and usage examples.

3. **LLM Integration:**
    - Integrate the documentation with a large language model to enable natural language querying.
    - Train the LLM to understand the context of the codebase and documentation, allowing it to provide accurate and relevant responses to user queries.

4. **Cron Job Setup:**
    - Establish a cron job that runs daily or weekly to trigger the documentation generator.
    - Rescan the codebase and update the documentation automatically with each run.