Independent University **Bangladesh (IUB)**

IUB Academic Repository

Graduate Thesis 2026

2026-08-24

Automated software requirement and design understanding using deep learning-based UML diagram recognition and LLM-based design pattern detection

Bhuiyan, Zarif Wasif

Independent University, Bangladesh (IUB)

https://ar.iub.edu.bd/handle/11348/1597

Downloaded from IUB Academic Repository



Automated Software Requirement and Design Understanding Using Deep Learning-Based UML Diagram Recognition and LLM-Based Design Pattern Detection

By

Zarif Wasif Bhuiyan
Student ID: 2531262

Md. Ataur Rahman
Student ID: 2511936

Summer 2026

Supervisor

Dr. Md. Tarek Habib

Associate Professor
Department of Computer Science and Engineering
Independent University, Bangladesh

24 August 2026

Dissertation submitted in partial fulfillment of the requirements for the degree of

Master of Science in Software Engineering

Department of Computer Science and Engineering

Independent University, Bangladesh

Attestation

We, Zarif Wasif Bhuiyan and Md. Ataur Rahman, hereby certify that the thesis entitled *Automated Software Requirement and Design Understanding Using Deep Learning-Based UML Diagram Recognition and LLM-Based Design Pattern Recognition* is an original work carried out by us. This thesis has been prepared as part of the academic requirements for the degree of Master of Science in Software Engineering at Independent University, Bangladesh.

We confirm that the research presented in this thesis is entirely our own and does not contain any plagiarized material. All sources of information, concepts, and ideas obtained from other works have been properly acknowledged and cited. The methodologies, analyses, results, and conclusions reported in this thesis are based on our own research, development, and implementation efforts.

We further declare that this thesis, either in its entirety or in part, has not previously been submitted to any other university or institution for the award of any academic degree, diploma, or qualification.

Zarif Wasif Bhuiyan	Md. Ataur Rahman
Date: 24 August 2026	Date: 24 August 2026

Acknowledgement

We would like to extend our sincere appreciation to our supervisor, **Dr. Md. Tarek Habib**, Associate Professor, Department of Computer Science and Engineering, Independent University, Bangladesh, for his exceptional guidance, thoughtful advice, consistent support, and encouragement throughout the course of this research. His academic expertise, valuable insights, and constructive feedback were instrumental in shaping this work and bringing the thesis to successful completion.

We are also thankful to the Department of Computer Science and Engineering, Independent University, Bangladesh, for providing a supportive academic environment along with the resources and facilities necessary for carrying out this research.

Our sincere gratitude also goes to all the respected faculty members whose teaching, guidance, and encouragement have contributed greatly to our academic development. We are equally grateful to our friends and colleagues for their cooperation, helpful discussions, and continued support during the research process.

Lastly, we would like to express our heartfelt appreciation to our families for their unwavering love, patience, understanding, and constant encouragement. Their support has been a continuous source of strength and inspiration throughout our academic journey.

Zarif Wasif Bhuiyan

Md. Ataur Rahman

Letter of Transmittal

Date: 24 August 2026

To
The Head
Department of Computer Science and Engineering
Independent University, Bangladesh

Subject: Submission of Thesis Report

Respected Sir,

We are pleased to submit our thesis report entitled *"Automated Software Requirement and Design Understanding Using Deep Learning-Based UML Diagram Recognition and LLM-Based Design Pattern Recognition"* as a partial fulfillment of the requirements for the degree of Master of Science in Software Engineering.

This thesis presents our research work on automated software understanding through deep learning-based UML diagram recognition and large language model-based design pattern recognition. The proposed framework integrates software requirement understanding and design knowledge extraction to support intelligent software engineering practices.

We have completed this research following proper academic standards and sincerely hope that this thesis report fulfills the requirements of the Department of Computer Science and Engineering.

We would like to express our sincere gratitude for providing us with the opportunity to conduct this research.

Sincerely,

Zarif Wasif Bhuiyan
Student ID: 2531262

Md. Ataur Rahman
Student ID: 2511936

Evaluation Committee

Supervisor
Name: Dr. Md. Tarek Habib, PhD **Signature: _____**
External Examiner
Name: Md. Sadekur Rahman **Signature: _____**
Office Use
Name: Dr. Saadia Binte Alam **Signature: _____** Program Coordinator
Name: Dr. AKM Mahbubur Rahman **Signature: _____** Head of the Department
Name: Prof. M Arshad Momen **Signature: _____** Director, Graduate Studies, Research and Industry Relations
Name: Md. Shahajada Masud Anowarul Haque **Signature: _____** Librarian

Abstract

Software understanding plays a critical role in software engineering activities such as requirement analysis, design comprehension, maintenance, and evolution. However, manually analyzing software artifacts, including UML diagrams and source code, is time-consuming, error-prone, and requires significant expertise. Recent advances in artificial intelligence, particularly deep learning and large language models (LLMs), provide new opportunities for automating software understanding tasks.

This thesis presents an artificial intelligence-based framework for automated software requirement and design understanding by integrating deep learning-based UML diagram recognition and LLM-based design pattern recognition. The first contribution focuses on automated UML diagram classification from images using multiple deep learning architectures. A balanced dataset containing 6,666 UML diagram images from six different categories was utilized to evaluate the performance of Inception-V3, Xception, MobileNet-V3, Custom CNN, and YOLO models. The experimental results show that YOLO achieved the highest recognition accuracy of approximately 99%, demonstrating the effectiveness of object detection-based approaches for UML diagram analysis.

The second contribution investigates automated software design pattern recognition using large language model-based code representations. Five pre-trained language models, including GraphCodeBERT, CodeBERT, CodeT5, RoBERTa, and TinyLlama, were evaluated for generating source code embeddings. The generated embeddings were classified using a k-Nearest Neighbor (k-NN) classifier to identify different software design patterns. Among the evaluated approaches, GraphCodeBERT with k-NN achieved the best performance with an F1-score of 0.969.

The findings demonstrate that artificial intelligence techniques can effectively analyze both visual software artifacts and source code representations. By combining UML diagram understanding and design pattern recognition, this research provides a unified framework for improving automated software analysis, reverse engineering, and intelligent software engineering applications.

The contributions of this thesis support the development of future AI-assisted software engineering systems by reducing manual analysis effort and improving software comprehension through automated understanding of requirements and design information.

Contents

Attestation i

Acknowledgement ii

Letter of Transmittal iii

Evaluation Committee iv

Abstract v

1 Introduction 1

1.1 Background of the Project 1
1.2 Problem Statement 3
1.3 Research Objectives 3
1.4 Research Contributions 4
1.5 Outcome of the Project 4
1.5.1 Theoretical Outcomes 4
1.5.2 Practical Outcomes 5
1.6 Organization of the Thesis 5

2 Literature Review 6

2.1 Introduction 6
2.2 Artificial Intelligence in Software Engineering 7
2.2.1 Role of AI in Software Development Lifecycle 7
2.2.2 Deep Learning Applications in Software Engineering 7
2.3 Large-Scale AI Models and Software Understanding 8
2.4 Software Requirement Understanding 8
2.5 Unified Modeling Language (UML) in Software Engineering 9
2.6 Summary 9
2.7 Deep Learning-Based UML Diagram Recognition 10
2.7.1 UML Dataset Development 10
2.7.2 CNN-Based UML Classification Approaches 11

2.7.3 Semantic Element Recognition from UML Images 11
2.7.4 Deep Learning-Based UML Extraction Approaches 11
2.7.5 Vision-Based UML Understanding Using Modern AI Models 12
2.8 Software Design Pattern Detection 12
2.8.1 Traditional Design Pattern Detection Approaches 13
2.8.2 Machine Learning-Based Design Pattern Recognition 13
2.8.3 Feature-Based and Hybrid Design Pattern Detection 13
2.8.4 Graph-Based Software Understanding 14
2.9 Large Language Model-Based Software Understanding 14
2.9.1 Transformer-Based Code Representation Models 15
2.9.2 Foundation Models for Software Engineering 15
2.9.3 Code Foundation Models 16
2.9.4 LLM-Based Software Architecture Understanding 16
2.10 Explainable Artificial Intelligence in Software Engineering 17
2.11 Research Gap 17
2.12 Summary 18

3 Research Methodology **19**
3.1 Introduction 19
3.2 Overall Research Framework 19
3.3 Research Workflow 20
3.4 UML Diagram Recognition Methodology 21
3.4.1 Dataset Description 21
3.4.2 Image Preprocessing 22
3.4.3 Deep Learning Model Development 22
3.4.4 Model Training and Optimization 23
3.4.5 Evaluation Metrics 24
3.5 LLM-Based Design Pattern Recognition Methodology 24
3.5.1 Dataset Description 25
3.5.2 Source Code Preprocessing 25
3.5.3 Code Representation Using Language Models 25
3.5.4 Embedding Extraction 26
3.5.5 k-Nearest Neighbor Classification 26
3.5.6 Experimental Evaluation 27
3.5.7 Evaluation Metrics 27
3.6 Experimental Environment 28
3.7 Summary 28

4 Deep Learning-Based UML Diagram Recognition **29**

4.1 Introduction . . . 29
4.2 Proposed UML Recognition Framework . . . 29
4.3 Image Collection . . . 31
4.4 Image Preprocessing . . . 31
4.5 Deep Learning Model Evaluation . . . 32
4.5.1 Inception-V3 . . . 32
4.5.2 Xception . . . 33
4.5.3 MobileNet-V3 . . . 34
4.5.4 YOLO Network . . . 35
4.5.5 Custom Layered CNN . . . 37
4.6 Model Evaluation and UML Recognition Decision . . . 38
4.7 Dataset Description . . . 38
4.8 UML Diagram Samples . . . 39
4.9 Dataset Splitting Strategy . . . 40
4.10 Input Image Configuration . . . 40
4.11 Deep Learning Model Architecture and Training Configuration . . . 41
4.11.1 Transfer Learning-Based Models . . . 41
4.11.2 Custom CNN Architecture . . . 42
4.11.3 YOLO-Based Recognition Model . . . 42
4.12 Training Configuration . . . 42
4.13 Model Evaluation Strategy . . . 43
4.14 Experimental Results and Performance Analysis . . . 43
4.14.1 Performance Comparison of Deep Learning Models . . . 44
4.14.2 Analysis of Model Performance . . . 46
4.15 Confusion Matrix Analysis . . . 46
4.16 Error Analysis . . . 47
4.17 UML Recognition Application Development . . . 47
4.18 Summary . . . 48

5 LLM-Based Design Pattern Recognition 49
5.1 Introduction . . . 49
5.2 Proposed Design Pattern Recognition Framework . . . 49
5.3 Dataset Description . . . 50
5.4 Source Code Preprocessing . . . 51
5.5 Code Representation and Embedding Generation . . . 51
5.5.1 Language Models for Code Embedding . . . 52
5.6 Embedding Extraction Process . . . 56
5.7 Embedding Visualization Analysis . . . 57
5.8 Design Pattern Classification Using k-NN . . . 58

5.8.1 k-Nearest Neighbor Classification Process . . . 58
5.8.2 Performance Evaluation of Different Language Models . . . 58
5.9 Classification Analysis . . . 59
5.10 Explainability Analysis . . . 59
5.11 Summary . . . 60

6 Results and Discussion **61**
6.1 Introduction . . . 61
6.2 Discussion of UML Diagram Recognition Results . . . 62
6.2.1 Performance Comparison of UML Recognition Models . . . 62
6.2.2 Comparison with Existing UML Recognition Studies . . . 63
6.3 Discussion of LLM-Based Design Pattern Recognition Results . . . 63
6.3.1 Performance Analysis of LLM-Based Recognition . . . 64
6.4 Integrated Analysis of Software Understanding . . . 65
6.5 Practical Implications . . . 65
6.6 Limitations and Future Improvements . . . 66
6.7 Summary . . . 66

7 Conclusion and Future Work **67**
7.1 Introduction . . . 67
7.2 Summary of Research Contributions . . . 67
7.3 Key Findings . . . 68
7.4 Research Significance . . . 68
7.5 Limitations . . . 69
7.6 Future Work . . . 69
7.7 Final Remarks . . . 69

References **71**

A Additional Experimental Details **80**
A.1 UML Recognition Experimental Configuration . . . 80
A.2 UML Dataset Categories . . . 80
A.3 Design Pattern Recognition Configuration . . . 81
A.4 Design Pattern Categories . . . 81

B Publication Statements **82**
B.1 Publication 1 . . . 82
B.2 Publication 2 . . . 83

List of Figures

3.1 Overall framework of automated software requirement and design understanding using deep learning and large language models. 20
3.2 Research workflow combining UML diagram recognition and LLM-based design pattern detection. 21

4.1 Conceptual framework of the proposed UML diagram recognition method. . . . 30
4.2 Inception-V3 architecture used for UML diagram classification. 32
4.3 Xception architecture used for UML diagram classification. 34
4.4 MobileNet-V3 architecture used for UML diagram classification. 35
4.5 YOLO architecture flowchart used for UML diagram classification. 36
4.6 Custom layered CNN architecture flowchart used for UML diagram classification. 37
4.7 Representative samples of different UML diagram categories. 39
4.8 Confusion matrix analysis of deep learning models for UML diagram recognition. 47
4.9 Web-based UML diagram recognition application. 48

5.1 Framework of LLM-based design pattern recognition using code embeddings and machine learning classification. 50
5.2 GraphCodeBERT architecture for design pattern recognition. 52
5.3 CodeBERT architecture for design pattern recognition. 54
5.4 CodeT5 pipeline for design pattern recognition. 55
5.5 t-SNE visualization of code embeddings generated by different language models for design pattern recognition. 57
5.6 SHAP-based explainability analysis of design pattern recognition using source code features. 60

6.1 Integrated framework combining UML diagram recognition and LLM-based design pattern recognition. 61

List of Tables

4.1 Distribution of UML Diagram Dataset 39
4.2 Dataset Splitting Strategy 40
4.3 Deep Learning Training Configuration 43
4.4 Performance comparison of deep learning models for UML diagram recognition 44

5.1 Design Pattern Dataset Categories 51
5.2 Performance Comparison of Language Model Embeddings with k-NN 59

6.1 Performance Summary of UML Diagram Recognition Models 62
6.2 Comparison with Existing UML Recognition Studies 63
6.3 Performance Summary of LLM-Based Design Pattern Recognition 64
6.4 Integrated Comparison of Research Contributions 65

A.1 UML Recognition Configuration 80
A.2 Design Pattern Recognition Configuration 81

Chapter 1

Introduction

1.1 Background of the Project

Software engineering has become increasingly complex due to the rapid growth of software systems, increasing user requirements, and continuous evolution of software architectures. Modern software development requires effective understanding of software requirements, system structures, and implementation details. Successful software engineering practices depend on accurate analysis, proper documentation, and effective communication among developers, architects, and stakeholders. Recent advances in artificial intelligence (AI) have introduced new opportunities for automating complex analysis tasks in software engineering and other domains [1, 2].

Deep learning has become one of the most influential approaches in artificial intelligence due to its ability to automatically learn meaningful feature representations from large-scale data. Deep neural networks have achieved remarkable performance in computer vision, natural language processing, and pattern recognition tasks. The success of convolutional neural networks (CNNs) in image classification has demonstrated the effectiveness of automatic feature learning for complex visual problems [3, 4, 11].

Several advanced deep learning architectures have further improved visual recognition performance. Xception introduced depthwise separable convolution for efficient feature extraction, MobileNetV3 focused on lightweight and efficient architecture design, and YOLO enabled real-time object detection through a unified detection framework [5, 6, 7]. Additionally, effective data augmentation strategies and efficient model scaling techniques have improved the generalization capability of deep learning models [8, 9]. Earlier architectures such as VGG also played an important role in establishing deep feature extraction methods for image recognition [10].

In software engineering, understanding software requirements and system designs is an essential activity for successful software development. Unified Modeling Language (UML) has become a widely accepted standard for representing software systems visually. UML diagrams provide structured representations of software components, classes, activities, deployment en-

vironments, and interactions among system elements. These visual representations help developers and stakeholders communicate system behavior and design decisions effectively [17, 19].

However, as software systems continue to increase in size and complexity, manual analysis of UML diagrams becomes time-consuming and requires significant domain expertise. Different UML diagrams may contain variations in layout, visual style, complexity, and representation format, making automated recognition a challenging research problem. Previous studies have explored deep learning-based UML classification and semantic element recognition, demonstrating the potential of AI techniques for automated UML understanding [18, 20].

Recent research has focused on developing reliable UML datasets, extracting semantic information from diagram images, and generating UML models using advanced artificial intelligence techniques. These studies highlight the importance of automated UML analysis for improving software documentation, reverse engineering, and intelligent software development [21, 22, 23].

Furthermore, multimodal learning approaches have introduced new possibilities for understanding UML diagrams from different visual sources, including hand-drawn sketches and diagram images. These approaches demonstrate the potential of combining vision-based models with advanced language-based methods for software modeling tasks [24, 25].

Apart from software requirement understanding, software design understanding is another important challenge in software engineering. Software design patterns provide reusable solutions to commonly occurring software design problems and help improve software maintainability, scalability, and reliability. Automatically recognizing design patterns from existing source code can assist developers in understanding software structures and supporting software maintenance activities.

Traditional design pattern recognition methods have mainly relied on manually defined rules, software metrics, graph matching approaches, and handcrafted features. Although these approaches achieved considerable success, they often face difficulties when patterns are implemented using different programming styles or modified structures. Recent studies have investigated machine learning-based approaches for improving automated design pattern recognition [26, 27].

The advancement of computer vision and artificial intelligence has also been supported by large-scale benchmark datasets, object detection frameworks, region-based detection methods, and dense feature learning techniques. These developments have contributed significantly to improving automated image analysis systems [12, 68, 69, 67, 13].

Therefore, this research proposes an integrated artificial intelligence-based software understanding framework that combines deep learning-based UML diagram recognition and large language model-based design pattern recognition. The framework integrates computer vision-based analysis of visual software artifacts with code representation learning techniques to support automated software requirement and design understanding.

1.2 Problem Statement

Although software engineering tools have significantly improved software development processes, understanding existing software artifacts remains a challenging task. Developers often need to manually analyze UML diagrams, source code, and software documentation to understand system requirements and architectural decisions. This manual process requires expert knowledge, consumes significant time, and may introduce interpretation errors.

Existing UML recognition approaches mainly focus on specific diagram types or limited classification problems. Although deep learning techniques have shown promising results, comprehensive recognition of multiple UML diagram categories within a unified framework remains a challenging research problem.

Similarly, existing design pattern recognition approaches depend heavily on predefined rules and manually engineered features. These approaches may not generalize effectively when developers use different implementation styles or modify traditional pattern structures.

Furthermore, most existing approaches separately address software requirement analysis and software design analysis. A unified AI-based framework capable of understanding both visual software models and source code-level design knowledge remains limited.

The major problems addressed in this research are:

- Manual UML diagram analysis requires significant time and expert knowledge.
- Existing UML recognition approaches have limited capability for automated multi-category software model understanding.
- Traditional design pattern recognition approaches lack flexibility for different programming implementations.
- Existing approaches do not effectively integrate requirement-level and implementation-level software analysis.
- There is a need for an AI-based framework that can automatically analyze both UML diagrams and source code.

1.3 Research Objectives

The main objective of this research is to develop an artificial intelligence based framework for automated software understanding by combining visual software artifact analysis and source code intelligence.

The specific objectives of this research are:

- To develop a deep learning-based approach for automatic recognition of different UML diagram categories.

- To evaluate multiple deep learning architectures for UML diagram image classification and recognition.
- To investigate large language model-based techniques for automated software design pattern recognition from source code.
- To analyze the effectiveness of intelligent code representation methods for understanding source code structures.
- To develop an integrated framework connecting software requirements and software design analysis.

1.4 Research Contributions

The major contributions of this research are summarized as follows:

- A deep learning-based UML diagram recognition framework is proposed for automatically identifying different UML diagram categories.
- Multiple deep learning architectures are evaluated to determine their effectiveness for visual software artifact understanding.
- A large language model-based software design pattern recognition approach is introduced using code embeddings and machine learning classification for automated source code analysis.
- An integrated software understanding perspective is presented by connecting visual design representations with implementation-level knowledge.
- This research contributes to the development of AI-assisted software engineering tools for reducing manual analysis effort.

1.5 Outcome of the Project

The theoretical and practical outcomes of this project are described below.

1.5.1 Theoretical Outcomes

The major theoretical outcomes of this project are:

- Development of an integrated AI-based software understanding framework.
- Investigation of deep learning techniques for UML diagram recognition.

- Analysis of automated approaches for software design understanding.
- Contribution to AI-assisted software engineering research.
- Providing insights into automated extraction of software knowledge.

1.5.2 Practical Outcomes

The practical outcomes of this project include:

- An automated UML diagram recognition system.
- An intelligent software design analysis approach.
- Reduction of manual effort in software documentation and analysis.
- Support for software maintenance and reverse engineering activities.
- A foundation for future AI-based software engineering applications.

1.6 Organization of the Thesis

The remainder of this thesis is organized as follows.

Chapter 2 presents the literature review related to UML diagram recognition, software design pattern recognition, code representation learning, and AI applications in software engineering.

Chapter 3 describes the research methodology, datasets, experimental setup, model selection, and evaluation procedures.

Chapter 4 presents the proposed deep learning-based UML diagram recognition framework and experimental results.

Chapter 5 presents the large language model-based design pattern recognition framework, code embedding generation approaches, and classification analysis.

Chapter 6 presents the integrated discussion, practical implications, limitations, and future research directions.

Finally, Chapter 7 concludes the thesis by summarizing the major findings and future extensions of this research.

Chapter 2

Literature Review

2.1 Introduction

Software engineering has experienced significant transformation due to the rapid advancement of artificial intelligence (AI), machine learning (ML), and deep learning technologies. The increasing complexity of modern software systems has created new challenges in understanding software requirements, design structures, implementation details, and maintenance processes. Traditional software engineering activities heavily depend on manual analysis and expert knowledge, which becomes difficult when software systems become large, complex, and continuously evolving.

Recent advances in artificial intelligence have introduced intelligent approaches for automatically analyzing and understanding software artifacts. AI-based techniques enable systems to learn meaningful patterns from large amounts of data and support automated decision-making processes. Deep learning models have demonstrated strong capabilities in learning complex feature representations from different types of data, including images, text, and source code [2, 1, 14, 15].

This thesis focuses on automated software requirement and design understanding through two complementary research directions. The first direction explores deep learning-based UML diagram recognition, where software design artifacts are analyzed as visual information. The second direction investigates large language model-based design pattern recognition, where source code representations are analyzed to identify reusable software structures.

This chapter presents a comprehensive review of existing research related to artificial intelligence in software engineering, UML diagram recognition, software design pattern detection, and large language model-based code understanding. The limitations of existing approaches are also analyzed to establish the motivation and research gap addressed in this thesis.

2.2 Artificial Intelligence in Software Engineering

Artificial intelligence has become an important research area in software engineering because of its ability to automate complex development and analysis activities. AI techniques have been applied throughout different phases of the software development lifecycle, including requirements engineering, software design, implementation, testing, and maintenance.

Modern software systems generate large volumes of information through various software artifacts such as requirement documents, UML diagrams, source code, test cases, and architectural descriptions. Extracting useful knowledge from these artifacts manually requires significant time and expertise. Therefore, intelligent approaches capable of automatically learning patterns from software-related data have gained increasing research attention [59, 63, 64, 65, 66].

Machine learning techniques allow computational systems to learn patterns from historical examples without relying on explicitly defined rules. Deep learning extends this capability by automatically learning hierarchical feature representations from complex data. These characteristics make AI techniques suitable for software engineering tasks involving classification, prediction, pattern recognition, and knowledge extraction [2, 61].

2.2.1 Role of AI in Software Development Lifecycle

The software development lifecycle consists of several interconnected phases, including requirement analysis, system design, implementation, testing, and maintenance. Each phase produces valuable software artifacts that contain important knowledge about the system.

Requirement engineering focuses on identifying user expectations and defining system functionality. Software design transforms these requirements into architectural structures, while implementation converts design decisions into executable source code. During maintenance, developers analyze existing systems to identify improvements, modifications, and potential issues.

Traditional software engineering practices depend largely on human-driven analysis of these artifacts. However, the increasing size and complexity of software systems make manual analysis time-consuming and challenging. AI-based approaches provide opportunities to automate artifact analysis and support developers in understanding complex software systems [60, 62].

Machine learning approaches can identify hidden patterns from software data, while deep learning models can automatically extract meaningful representations from complex inputs. These capabilities have encouraged the application of AI techniques in different software engineering activities.

2.2.2 Deep Learning Applications in Software Engineering

Deep learning has demonstrated remarkable success in various domains because of its ability to automatically learn effective feature representations from large-scale datasets. Convolutional

neural networks (CNNs), in particular, have been widely applied for image classification, object detection, and visual feature extraction [3, 11].

Recent deep learning architectures have significantly improved the efficiency and accuracy of automated analysis systems. Vision Transformer (ViT) introduced transformer-based architectures into computer vision by treating image patches as sequential tokens and applying attention mechanisms for visual understanding [14].

Similarly, surveys on convolutional neural network architectures have highlighted the continuous development of deep learning models and their applications in different computer vision tasks [15].

These advancements have encouraged researchers to apply deep learning techniques for analyzing software artifacts represented in visual formats, including UML diagrams. Since UML diagrams contain structural and visual patterns, deep learning approaches provide a promising solution for automated software model understanding.

2.3 Large-Scale AI Models and Software Understanding

Recent advances in artificial intelligence have resulted in powerful models capable of learning complex patterns from large-scale datasets. These models have significantly influenced software engineering research by enabling automated code analysis, program understanding, software documentation generation, and knowledge extraction.

Transformer-based architectures have become a foundation for many modern AI systems because of their ability to capture contextual relationships and long-range dependencies. Unlike traditional sequential models, transformer architectures utilize attention mechanisms to identify important relationships within input data [1].

Large language models (LLMs) have introduced new opportunities for software engineering automation by providing semantic understanding of source code. These models are trained on large collections of programming languages and natural language data, allowing them to learn programming structures, dependencies, and contextual relationships [41, 42].

Recent studies have demonstrated the effectiveness of large language models for different software engineering tasks, including code generation, program comprehension, debugging, documentation, and software analysis. The ability of LLMs to understand complex code structures makes them promising for automated software design understanding [47, 48].

2.4 Software Requirement Understanding

Software requirement understanding is one of the fundamental activities in software engineering. Requirements define the expected behavior, functionalities, constraints, and objectives of a software system. Accurate requirement analysis is essential because errors introduced dur-

ing this phase may negatively affect later stages of software development.

Software requirements are represented through different forms, including textual descriptions, models, and graphical representations. Among these representations, Unified Modeling Language (UML) diagrams are widely used because they provide standardized visual communication between developers, architects, and stakeholders.

Automated requirement understanding aims to reduce manual analysis effort and improve consistency between software documentation and actual system implementation. Artificial intelligence techniques provide opportunities to analyze software artifacts automatically and extract meaningful knowledge from them [59].

The increasing complexity of software systems has created the need for intelligent approaches capable of understanding both high-level software models and implementation-level information. Therefore, automated analysis of UML diagrams and source code has become an important research direction in AI-assisted software engineering.

2.5 Unified Modeling Language (UML) in Software Engineering

Unified Modeling Language (UML) is a standardized modeling language used for visualizing, specifying, constructing, and documenting software systems. UML provides different diagram types that represent structural and behavioral aspects of software applications [16].

Structural UML diagrams describe the static organization of software systems. Examples include class diagrams, component diagrams, and deployment diagrams. These diagrams represent elements such as classes, components, relationships, and system deployment structures [70].

Behavioral UML diagrams describe the dynamic behavior and interaction of software systems. Activity diagrams represent system workflows, while sequence diagrams describe interactions among different system components over time [71].

UML diagrams contain valuable software knowledge, including system structure, component relationships, and design decisions. Therefore, automated UML understanding has become an important research area for supporting software maintenance, reverse engineering, documentation analysis, and intelligent software development [72].

2.6 Summary

This section discussed the importance of artificial intelligence and advanced AI models in modern software engineering. Deep learning approaches provide effective solutions for visual software artifact analysis, while large language models enable semantic understanding of source code.

The following sections review existing research on deep learning-based UML diagram recognition, including dataset development, image classification, semantic extraction, and modern AI-based diagram understanding approaches.

2.7 Deep Learning-Based UML Diagram Recognition

The application of deep learning in computer vision has created new opportunities for automated UML diagram understanding. Since UML diagrams are visual software artifacts, they can be analyzed using image classification, feature extraction, and object detection techniques [73, 74, 75].

Traditional UML analysis methods mainly depend on manual inspection or rule-based approaches. However, these approaches become difficult to apply for large-scale software systems because UML diagrams may contain variations in layout, notation style, complexity, and visual representation.

Deep learning approaches provide an alternative solution by automatically learning meaningful visual patterns from UML images. These models can extract hierarchical features from diagram structures and classify different UML categories without requiring manually designed features [2, 3].

2.7.1 UML Dataset Development

Dataset availability is one of the most important factors for developing reliable deep learning-based UML recognition systems. A well-structured dataset enables effective model training and provides a standard platform for performance evaluation.

In this study, a dataset for UML diagram classification to support software engineering education and automated UML analysis research. Their work provided a structured collection of UML diagrams and demonstrated the potential of machine learning approaches for software modeling tasks [19].

However, developing UML datasets remains challenging because diagrams created by different developers may vary in notation style, visual arrangement, resolution, and complexity. Models trained on limited diagram styles may face difficulty when applied to unseen software projects.

The importance of reliable ground-truth datasets for UML recognition. Their study focused on creating and validating UML datasets using deep learning techniques to improve the reliability and evaluation process of automated UML analysis systems [21].

2.7.2 CNN-Based UML Classification Approaches

Convolutional Neural Networks (CNNs) have been widely applied in image classification tasks because of their ability to automatically extract hierarchical visual features from input images [3, 10, 11].

In this study, a CNN-based approach for automatic classification of UML class diagrams. Their study demonstrated that CNN models can effectively learn visual characteristics from UML images and classify diagram categories without relying on manually engineered features [17, 76].

The success of CNN-based UML recognition demonstrates that software diagrams can be treated as visual objects containing meaningful structural information. However, focusing mainly on class diagrams limits the ability of these approaches to provide complete software understanding.

Multiclass UML diagram classification using deep learning techniques. Their research explored the capability of neural network models to recognize multiple UML diagram categories from image data [18].

Although multiclass recognition improves the flexibility of automated UML analysis, existing approaches still face challenges related to dataset diversity, diagram complexity, and model generalization capability.

2.7.3 Semantic Element Recognition from UML Images

Beyond diagram-level classification, extracting semantic information from UML images is another important research direction. Semantic recognition aims to identify meaningful elements such as classes, attributes, relationships, components, and other software modeling structures.

In this study, an approach for automatically recognizing semantic elements from UML class diagram images. Their research demonstrated that combining visual feature extraction with semantic analysis can improve automated UML understanding [20].

Semantic extraction provides deeper understanding compared with simple classification because it enables systems to identify internal structures and relationships within UML diagrams. However, accurate semantic recognition remains challenging due to variations in UML notation, diagram complexity, and visual representation.

2.7.4 Deep Learning-Based UML Extraction Approaches

Recent studies have investigated more advanced approaches for extracting UML information using deep learning techniques.

This study conducted a comparative study on deep learning-based UML class diagram extraction methods. Their research analyzed different approaches for automatically extracting UML

information and highlighted challenges related to diagram variations and structural complexity [22].

These studies demonstrate that deep learning techniques can effectively support automated UML analysis. However, many existing approaches focus on limited UML categories, especially class diagrams. A complete software understanding framework requires the capability to recognize multiple UML diagram types and connect visual information with software design knowledge.

2.7.5 Vision-Based UML Understanding Using Modern AI Models

Recent developments in artificial intelligence have introduced multimodal models capable of understanding both visual and textual information. These approaches provide new possibilities for analyzing software diagrams beyond traditional CNN-based classification methods.

Conrardy and Cabot investigated image-to-UML generation using large language models. Their study demonstrated the potential of multimodal AI systems for interpreting UML diagram images and generating structured software models [23].

UML model generation from diagram images using multimodal large language models. Their research showed that advanced AI models can understand visual software representations and transform them into structured software artifacts [24].

Buchmann and Fraas investigated AI-based recognition of sketched class diagrams, demonstrating that intelligent models can analyze manually created software diagrams [25].

Although multimodal AI approaches provide promising results, several challenges remain, including limited datasets, computational requirements, model reliability, and explainability. These limitations motivate further research into robust and generalized AI-based UML recognition frameworks.

2.8 Software Design Pattern Detection

Software design patterns represent reusable solutions to commonly occurring software design problems. They provide standardized approaches for improving software maintainability, flexibility, scalability, and architectural quality [28].

Automatically detecting design patterns from existing source code is valuable for software comprehension, reverse engineering, documentation generation, and software maintenance. However, identifying design patterns remains challenging because developers may implement the same pattern using different coding styles, programming structures, and design variations.

2.8.1 Traditional Design Pattern Detection Approaches

Early design pattern detection approaches mainly relied on predefined rules, software metrics, and structural matching techniques [29, 30].

These approaches attempted to identify design patterns by comparing source code structures with known pattern definitions. Although effective for some well-defined implementations, rule-based techniques often struggle with complex software systems and pattern variations.

A design pattern mining approach enhanced by machine learning techniques. Their research demonstrated that software metrics combined with machine learning methods could improve automated pattern identification compared with purely rule-based approaches [29].

Design pattern detection using software metrics and machine learning techniques. Their study showed that software characteristics such as inheritance relationships and coupling information could provide useful information for identifying design patterns [30].

Although traditional approaches improved automated detection, they mainly depended on manually selected features and predefined structural rules. Therefore, their effectiveness was limited when developers implemented patterns using different programming styles [31, 32].

2.8.2 Machine Learning-Based Design Pattern Recognition

Machine learning approaches improved design pattern recognition by allowing models to learn pattern characteristics from software examples instead of depending only on handcrafted rules.

Machine learning-based design pattern recognition and investigated the possibility of automatically identifying design patterns from software systems using learning algorithms [31].

In this study, a comprehensive survey of design pattern detection approaches. Their study analyzed traditional and intelligent approaches and highlighted the transition from rule-based methods toward machine learning-based solutions [32].

In this study, a design pattern detection method based on idiomatic implementation matching in Java. Their approach focused on recognizing implementation characteristics associated with specific software design patterns [33].

Oberhauser investigated automatic software design pattern recognition across multiple programming languages using machine learning approaches. The study demonstrated the potential of learned representations for improving automated pattern recognition [34].

2.8.3 Feature-Based and Hybrid Design Pattern Detection

Feature engineering has remained an important component of many design pattern detection approaches. Researchers have explored software features such as class relationships, method interactions, inheritance structures, and architectural characteristics.

Kouli and Rasoolzadegan proposed a feature-based method for detecting design patterns in

source code. Their work emphasized the importance of selecting effective software features for improving detection performance [35].

In this study, a comparative study of design pattern detection tools and analyzed the effectiveness of different automated approaches. Their study highlighted the challenges of achieving reliable pattern detection across different software systems [36].

This study investigated supervised machine learning-based detection of Singleton design pattern variants. Their research demonstrated that machine learning models can identify variations of traditional software patterns [37].

Oberhauser and Moser introduced a hybrid approach combining Bayesian networks, machine learning, graph embeddings, and micropattern rules. Their findings showed that combining multiple software representations can improve design pattern detection capability [38].

2.8.4 Graph-Based Software Understanding

Source code naturally contains structural relationships among classes, methods, dependencies, and software components. Graph-based approaches represent software systems as graphs and analyze relationships among different elements.

This study investigated machine learning approaches for predicting architectural design patterns. Their work demonstrated the potential of learning-based methods for understanding software architecture [39].

In this study, an approach for recovering and optimizing software architecture based on source code and directory hierarchies. Their research highlighted the importance of extracting architectural knowledge from software implementations [40].

Although graph-based methods provide structural understanding, they often require complex representation techniques and may not fully capture semantic information contained in source code.

2.9 Large Language Model-Based Software Understanding

Recent advances in transformer architectures and large language models have significantly influenced software engineering research. Unlike traditional machine learning approaches, LLMs can learn contextual information from large collections of source code and natural language data. These capabilities have created new opportunities for automated software analysis, code understanding, and intelligent software development.

Transformer-based models have become the foundation of many modern AI systems because of their ability to capture contextual relationships and long-range dependencies through attention mechanisms [1].

BERT introduced bidirectional transformer-based representation learning and demonstrated the effectiveness of contextual embeddings for natural language understanding. The model

learns information from both previous and following tokens, enabling better semantic representation [41].

The success of transformer architectures in natural language processing encouraged researchers to develop specialized models for programming languages. These models are trained on large-scale source code repositories and are capable of learning programming syntax, semantics, and structural relationships.

2.9.1 Transformer-Based Code Representation Models

Understanding source code requires capturing both textual information and structural relationships among programming elements. Traditional machine learning methods usually depend on manually designed software metrics and features, which limits their ability to generalize across different programming styles.

Transformer-based code representation models overcome these limitations by learning distributed representations directly from source code. These representations capture contextual information about variables, functions, classes, and dependencies.

CodeBERT extended transformer-based representation learning to programming languages by jointly learning from natural language and source code. The model was trained using large-scale code repositories and demonstrated strong performance in various code understanding tasks, including code search and classification [42].

GraphCodeBERT further improved code representation learning by incorporating program data-flow information. Unlike traditional sequence-based models, GraphCodeBERT considers structural relationships between variables and expressions, allowing better understanding of program semantics [43].

CodeT5 introduced identifier-aware encoder-decoder architecture for source code understanding and generation. The model uses programming language-specific characteristics to improve tasks such as code generation, summarization, and translation [44].

This Study proposed unified pre-training approaches for program understanding and generation. Their research demonstrated that large-scale pre-trained models can effectively learn general programming knowledge and support multiple software engineering tasks [45].

2.9.2 Foundation Models for Software Engineering

Large-scale foundation models have introduced a significant change in artificial intelligence research. These models are trained on massive datasets and can be adapted to perform multiple downstream tasks without requiring extensive task-specific training.

This study demonstrated the effectiveness of large language models through few-shot learning. Their research showed that sufficiently large models can perform various tasks by learning from limited examples [46].

In software engineering, foundation models provide opportunities for automated programming assistance, software documentation generation, debugging, testing, and code analysis.

This study presented a comprehensive survey of large language models for software engineering. Their study discussed current applications, challenges, and future opportunities of LLMs in different software engineering activities [47].

The application of large language models in software engineering and highlighted their potential for improving development productivity, automation, and software quality [48].

The DevGPT study analyzed real-world developer interactions with ChatGPT and provided insights into how developers use conversational AI systems for software-related tasks [49].

2.9.3 Code Foundation Models

Recent research has introduced specialized foundation models designed specifically for programming tasks. These models are trained using large-scale code datasets and provide powerful capabilities for software understanding.

Code Llama is a code-specialized foundation model developed from the LLaMA architecture. It was optimized for programming-related tasks such as code generation, completion, and reasoning [50].

LLaMA demonstrated that efficient open foundation models can achieve strong language understanding performance while using fewer computational resources compared with extremely large proprietary models [51].

Mistral 7B introduced an efficient large language model architecture that achieved competitive performance with a relatively smaller parameter size. This demonstrates that efficient models can provide practical solutions for resource-limited environments [52].

The opportunities and risks associated with foundation models, including their capabilities, limitations, reliability issues, and responsible deployment challenges [53].

2.9.4 LLM-Based Software Architecture Understanding

Software architecture understanding is an important aspect of software maintenance and evolution. Traditional architecture recovery techniques mainly depend on static analysis, software metrics, and manually defined rules.

Recent studies indicate that large language models can provide semantic understanding of software systems by analyzing source code, documentation, and architectural descriptions.

This study conducted a systematic literature review on the relationship between software architecture and large language models. Their study highlighted the potential of LLMs for automated architecture analysis, recovery, and decision support [54, 77, 78].

Although LLMs demonstrate strong capabilities in general code understanding, their application for specific software design pattern recognition remains limited. Identifying reusable design structures from source code requires combining semantic representations with effective

classification approaches [79, 80].

2.10 Explainable Artificial Intelligence in Software Engineering

As AI-based approaches become increasingly integrated into software engineering tasks, understanding model decisions has become an important requirement. Explainable Artificial Intelligence (XAI) techniques aim to provide interpretable explanations of model predictions.

Lundberg and Lee proposed SHAP (SHapley Additive exPlanations), a unified approach for interpreting machine learning model predictions. SHAP provides feature contribution analysis by estimating the importance of individual features [55].

This Study introduced LIME, a model-agnostic explanation technique that explains predictions by learning local interpretable models around individual instances [56].

This Study provided a comprehensive review of explainable artificial intelligence concepts, including taxonomies, challenges, and opportunities for responsible AI systems [57, 58].

In software engineering applications, explainability is important because developers need confidence in automated recommendations and predictions. Therefore, XAI techniques can improve trust and transparency in AI-assisted software analysis systems [82, 83].

2.11 Research Gap

Based on the reviewed literature, several research gaps have been identified.

First, existing UML recognition approaches mainly focus on specific UML categories or limited image classification problems. Although deep learning models achieve promising performance, comprehensive software requirement understanding remains challenging.

Second, many UML recognition studies mainly concentrate on class diagrams, while other important UML categories such as activity, sequence, component, and deployment diagrams receive comparatively less attention.

Third, traditional design pattern detection approaches mainly depend on handcrafted rules, software metrics, and structural matching. These approaches may not generalize effectively when developers implement patterns using different programming styles.

Fourth, machine learning-based design pattern recognition approaches improve automation but often depend on manually selected features. Such approaches may fail to capture deeper semantic relationships present in modern software systems.

Fifth, although transformer-based code models and large language models have demonstrated strong code understanding capabilities, their application for automated design pattern recognition remains an emerging research direction.

Sixth, UML recognition and source code design pattern detection are generally investigated

as separate problems. A unified artificial intelligence framework connecting visual software models and source code understanding is still limited.

Therefore, this thesis proposes an integrated AI-based software understanding framework combining deep learning-based UML diagram recognition and large language model-based design pattern recognition. The proposed approach aims to bridge the gap between software requirement understanding and software design knowledge extraction.

2.12 Summary

This chapter presented a comprehensive review of artificial intelligence applications in software engineering, UML diagram recognition, software design pattern detection, and large language model-based software understanding.

Deep learning approaches have demonstrated strong capabilities for analyzing visual software artifacts by automatically extracting meaningful features from UML diagrams. However, existing approaches are often limited by dataset size, diagram diversity, and restricted UML categories.

Similarly, traditional and machine learning-based design pattern detection methods provide automated solutions for identifying reusable software structures. However, these approaches often struggle with implementation variations and limited semantic understanding.

Recent transformer-based code models and large language models provide powerful representations for source code analysis. Models such as CodeBERT, GraphCodeBERT, and CodeT5 demonstrate the potential of AI-driven software understanding.

Despite these advancements, limited research has explored a unified framework that combines visual software model recognition with source code-level design pattern understanding. This research gap motivates the development of the proposed artificial intelligence-based software understanding framework presented in this thesis.

Chapter 3

Research Methodology

3.1 Introduction

This chapter presents the research methodology adopted in this thesis for automated software requirement and design understanding. The proposed research combines two major artificial intelligence-based approaches: deep learning-based UML diagram recognition and large language model (LLM)-based design pattern detection.

The methodology is designed to analyze two important software engineering artifacts: visual UML diagrams and source code. UML diagrams represent software requirements and design structures visually, whereas source code contains implementation-level design knowledge. By analyzing both artifacts, the proposed framework aims to provide a comprehensive understanding of software systems.

The research methodology consists of several phases, including data collection, preprocessing, feature extraction, model development, training, classification, and performance evaluation. The complete workflow integrates computer vision techniques for UML recognition and code representation learning techniques for design pattern detection.

3.2 Overall Research Framework

The overall framework of this research is illustrated in Figure 3.1.

The proposed framework consists of two parallel research pipelines. The first pipeline focuses on UML diagram recognition using deep learning models. UML diagram images are collected, preprocessed, and used to train different deep learning architectures for automated classification.

The second pipeline focuses on design pattern recognition from source code. Java programs containing different design patterns are processed to generate semantic code representations using pre-trained language models. The generated embeddings are then classified using machine learning techniques.

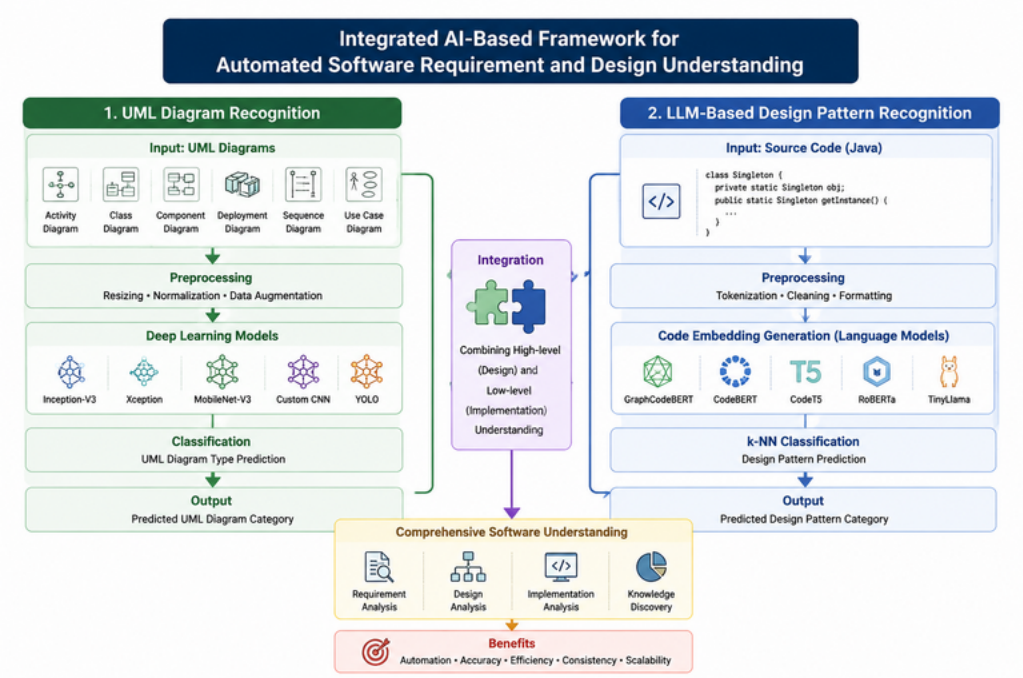


Figure 3.1: Overall framework of automated software requirement and design understanding using deep learning and large language models.

Finally, outputs from both pipelines are analyzed together to support automated software requirement and design understanding.

3.3 Research Workflow

The complete research workflow is presented in Figure 3.2.

The workflow contains the following major stages:

1. Data collection and preparation
2. Data preprocessing
3. Feature extraction and model development
4. Model training and optimization
5. Performance evaluation
6. Software understanding and analysis

Each research component follows a systematic experimental process to ensure reliable evaluation and comparison among different approaches.

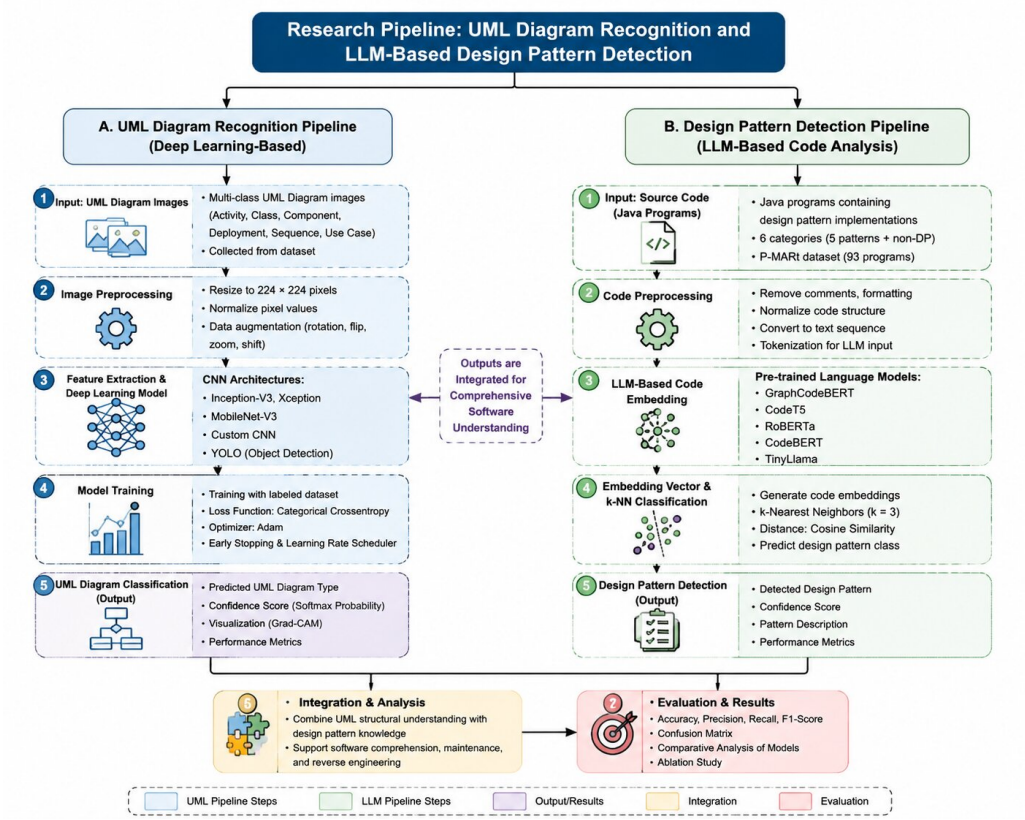


Figure 3.2: Research workflow combining UML diagram recognition and LLM-based design pattern detection.

3.4 UML Diagram Recognition Methodology

The first research component of this thesis focuses on automated recognition of UML diagrams using deep learning-based computer vision techniques. UML diagrams are visual software artifacts that contain important information about system requirements and design structures. The objective of this component is to develop a robust classification framework capable of identifying different UML diagram categories automatically.

The proposed methodology consists of five major steps: dataset preparation, image preprocessing, deep learning model development, model training, and performance evaluation.

3.4.1 Dataset Description

A dedicated UML diagram dataset was used for developing and evaluating the recognition framework. The dataset contains six different UML diagram categories:

- Activity Diagram
- Class Diagram
- Component Diagram

- Deployment Diagram
- Sequence Diagram
- Use Case Diagram

The dataset contains a total of 6,666 UML diagram images, with 1,111 images collected for each UML category. The balanced distribution of classes helps reduce classification bias and enables fair performance comparison among different deep learning models.

The dataset was divided into three subsets:

- Training set: 80% of the dataset
- Validation set: 10% of the dataset
- Testing set: 10% of the dataset

The training dataset was used for model learning, the validation dataset was used for hyperparameter optimization, and the testing dataset was used for final performance evaluation.

3.4.2 Image Preprocessing

Image preprocessing is an essential step in deep learning-based image classification because input images must maintain consistent characteristics before being provided to neural networks.

All UML diagram images were resized to 224×224 pixels to maintain uniform input dimensions across different models. Image normalization was performed to improve training stability and accelerate convergence.

The preprocessing pipeline included:

1. Image resizing
2. Pixel value normalization
3. Data organization according to UML categories
4. Dataset splitting for training, validation, and testing

These preprocessing operations ensure that the models receive standardized inputs and can effectively learn discriminative visual features from UML diagrams.

3.4.3 Deep Learning Model Development

Several deep learning architectures were investigated to evaluate their effectiveness for UML diagram recognition. Both transfer learning-based models and a custom convolutional neural network architecture were considered.

Inception-V3

Inception-V3 is a deep convolutional neural network architecture that uses multiple convolutional operations with different filter sizes to extract complex visual features. The model has demonstrated strong performance in various image classification tasks and was selected as one of the baseline transfer learning models.

Xception

Xception is an advanced CNN architecture based on depthwise separable convolution operations. It reduces computational complexity while maintaining strong feature extraction capability. The model was evaluated to analyze its effectiveness for UML image classification.

MobileNet-V3

MobileNet-V3 is a lightweight deep learning architecture designed for efficient deployment. It combines depthwise separable convolution with attention mechanisms, making it suitable for resource-efficient intelligent applications.

Custom CNN

A custom convolutional neural network architecture was developed specifically for UML diagram recognition. The proposed CNN consists of multiple convolutional layers with increasing feature extraction capacity.

The architecture was designed to automatically learn UML-specific visual features such as shapes, relationships, and structural patterns present in different diagram categories.

YOLO-Based Recognition

A YOLO-based object recognition approach was also investigated for UML recognition. Unlike traditional classification models, YOLO provides real-time detection capabilities by identifying objects and their locations within images.

The objective of including YOLO was to evaluate whether object detection approaches could provide effective solutions for automated UML understanding.

3.4.4 Model Training and Optimization

All deep learning models were trained using the prepared UML dataset. The training process involved optimization of model parameters through iterative learning.

The main training procedures included:

- Forward propagation for feature extraction

- Loss calculation using prediction errors
- Backpropagation for parameter optimization
- Validation-based performance monitoring

Model performance was monitored during training to reduce overfitting and ensure effective generalization on unseen UML diagram images.

3.4.5 Evaluation Metrics

The performance of UML recognition models was evaluated using standard classification metrics:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{3.1}$$

$$Precision = \frac{TP}{TP + FP} \tag{3.2}$$

$$Recall = \frac{TP}{TP + FN} \tag{3.3}$$

$$F1 - score = 2 \times \frac{Precision \times Recall}{Precision + Recall} \tag{3.4}$$

where TP, TN, FP, and FN represent true positive, true negative, false positive, and false negative predictions, respectively.

Confusion matrices and class-wise performance analysis were also used to understand model behavior across different UML categories.

3.5 LLM-Based Design Pattern Recognition Methodology

The second research component of this thesis focuses on automated software design pattern recognition using large language model (LLM)-based code representations and machine learning classification. Design patterns provide reusable solutions for common software design problems; however, identifying them manually from source code is challenging, especially in large-scale software systems.

The proposed methodology utilizes pre-trained language models to extract semantic representations from source code. The generated code embeddings are then classified using a k-Nearest Neighbor (k-NN) classifier to identify different design pattern categories.

3.5.1 Dataset Description

The design pattern recognition experiment was conducted using the P-MARt dataset, which contains Java programs representing different software design patterns.

The dataset consists of 93 Java source code samples distributed into six different categories:

- Abstract Factory
- Builder
- Factory Method
- Prototype
- Singleton
- Non-Design Pattern

The inclusion of non-design pattern samples allows the model to distinguish between pattern-based and normal software implementations. This makes the classification task more realistic for practical software analysis scenarios.

3.5.2 Source Code Preprocessing

Before generating code representations, the source code samples were preprocessed to ensure consistency and remove unnecessary information.

The preprocessing steps include:

1. Reading Java source files from the dataset.
2. Removing unnecessary formatting variations.
3. Preparing source code representation for language model input.
4. Generating structured input sequences for embedding extraction.

The preprocessing stage ensures that the language models focus on meaningful software structures and semantic information rather than irrelevant formatting differences.

3.5.3 Code Representation Using Language Models

Large language models are capable of learning contextual information from source code. In this research, multiple pre-trained models were investigated to generate meaningful code embeddings.

The following models were evaluated:

- GraphCodeBERT
- CodeBERT
- CodeT5
- RoBERTa
- TinyLlama

These models generate high-dimensional vector representations of source code, where similar software structures are represented by similar embedding patterns.

GraphCodeBERT incorporates structural information from source code and captures relationships between programming elements. CodeBERT combines natural language and programming language representations, while CodeT5 provides an encoder-decoder-based approach for code understanding.

3.5.4 Embedding Extraction

The source code samples were passed through the selected language models to generate embedding vectors. These embeddings represent semantic and structural characteristics of the software implementation.

The extracted embeddings transform source code into numerical feature spaces that can be processed by traditional machine learning classifiers.

The general embedding generation process can be represented as:

$$E = f(C) \tag{3.5}$$

where,

- C represents the input source code.
- f represents the pre-trained language model.
- E represents the generated code embedding vector.

3.5.5 k-Nearest Neighbor Classification

After generating code embeddings, a k-Nearest Neighbor classifier was applied for design pattern classification.

The k-NN algorithm classifies a new sample based on the similarity between its embedding representation and existing labeled samples. The distance between embedding vectors was calculated to identify the closest training examples.

The classification process can be represented as:

$$d(x, y) = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2} \tag{3.6}$$

where $d(x, y)$ represents the distance between two embedding vectors.

The class of the new sample is determined based on the majority class among its nearest neighbors.

3.5.6 Experimental Evaluation

The performance of different language model embeddings was evaluated by comparing their classification results using the k-NN classifier.

The experiments focused on identifying the most effective combination of language model embedding and classification approach for software design pattern recognition.

The evaluation process included comparison among:

- Different code representation models.
- Different embedding characteristics.
- Classification performance across design pattern categories.

3.5.7 Evaluation Metrics

The design pattern recognition performance was evaluated using standard classification metrics[87-97], including accuracy, precision, recall, and F1-score.

$$Accuracy = \frac{Correct\ Predictions}{Total\ Predictions} \tag{3.7}$$

$$Precision = \frac{TP}{TP + FP} \tag{3.8}$$

$$Recall = \frac{TP}{TP + FN} \tag{3.9}$$

$$F1 = 2 \times \frac{Precision \times Recall}{Precision + Recall} \tag{3.10}$$

These metrics provide comprehensive evaluation of classification performance, especially for multiple design pattern categories with different implementation characteristics.

3.6 Experimental Environment

The experiments for both UML recognition and design pattern recognition were conducted using modern machine learning and deep learning frameworks.

The software environment included:

- Python programming language.
- Deep learning frameworks for model development.
- Transformer-based libraries for language model processing.
- Machine learning libraries for classification and evaluation.

The hardware environment consisted of computational resources capable of training deep learning models and processing large-scale neural network operations.

3.7 Summary

This chapter presented the complete research methodology adopted in this thesis. The proposed framework combines computer vision-based UML diagram recognition and LLM-based design pattern recognition to achieve automated software requirement and design understanding.

The UML recognition component applies deep learning models for analyzing visual software artifacts, while the design pattern recognition component uses code embeddings generated from large language models and machine learning classification.

The following chapters present the experimental results, analysis, and discussion of the proposed approaches.

Chapter 4

Deep Learning-Based UML Diagram Recognition

4.1 Introduction

This chapter presents the first major contribution of this thesis, which focuses on automated UML diagram recognition using deep learning techniques. UML diagrams are essential software engineering artifacts that represent system requirements, structures, and behaviors. However, manually analyzing large numbers of UML diagrams requires significant effort and domain knowledge.

To address this challenge, a deep learning-based framework was developed for automatic recognition of different UML diagram categories. The proposed approach treats UML diagrams as visual data and applies computer vision-based deep learning models to extract meaningful features and perform classification.

The overall workflow of the proposed UML recognition approach is illustrated in Figure 4.1.

4.2 Proposed UML Recognition Framework

The proposed UML diagram recognition framework provides an automated approach for identifying different UML diagram categories using deep learning techniques. The overall conceptual workflow of the proposed framework is illustrated in Figure 4.1.

The framework is designed to analyze UML diagrams as visual software artifacts and extract meaningful patterns for automatic classification. The complete workflow consists of several sequential stages, including UML image collection, preprocessing, dataset preparation, deep learning model training, performance evaluation, and final UML diagram recognition.

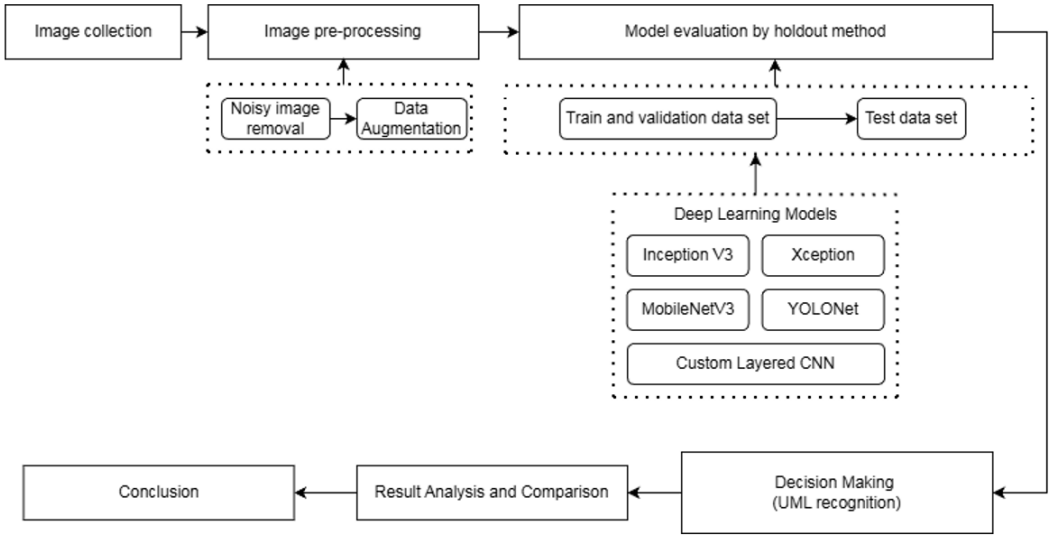


Figure 4.1: Conceptual framework of the proposed UML diagram recognition method.

As shown in Figure 4.1, the proposed framework begins with the collection of UML diagram images from the prepared dataset. The collected images represent different UML categories and provide the visual information required for training deep learning models.

Since UML diagrams may contain variations in layout, resolution, notation style, and visual complexity, preprocessing is performed before model training. The preprocessing stage ensures image consistency by applying necessary transformations and preparing the images into a suitable format for deep learning architectures.

After preprocessing, the dataset is divided into training, validation, and testing subsets. The training set is used for learning visual features from UML diagrams, while the validation set is used for monitoring model performance and adjusting training parameters. The testing set is kept separate for final evaluation of the developed models.

In the next stage, the processed UML images are provided to multiple deep learning architectures. These models automatically extract hierarchical features from diagram images and learn the visual characteristics associated with different UML categories. The extracted features allow the models to distinguish between different diagram types based on their structural and visual patterns.

After completing the training process, the developed models are evaluated using standard classification metrics, including accuracy, precision, recall, and F1-score. A comparative analysis is performed to identify the most effective architecture for UML diagram recognition.

Finally, the selected model is used for automated UML diagram classification. The predicted output provides the recognized UML category, reducing the need for manual inspection and supporting efficient software requirement understanding.

The proposed framework provides a complete pipeline for transforming raw UML diagram images into meaningful classification results. By combining computer vision techniques with deep learning models, the framework demonstrates the potential of artificial intelligence for

automated software engineering artifact analysis.

4.3 Image Collection

The UML diagram dataset contains multiple categories of UML diagrams representing different software modeling perspectives. The dataset includes six major UML diagram types:

- Activity Diagram
- Class Diagram
- Component Diagram
- Deployment Diagram
- Sequence Diagram
- Use Case Diagram

A balanced dataset distribution was maintained to ensure that each UML category contributes equally during model training and evaluation.

4.4 Image Preprocessing

Image preprocessing is performed to prepare UML diagram images for deep learning models. Since images collected from different sources may contain variations in size, quality, and visual characteristics, preprocessing helps create standardized input data.

The preprocessing stage includes:

- Noise removal
- Data augmentation
- Image resizing
- Dataset normalization

Data augmentation techniques are applied to improve model generalization by creating variations of training samples. This reduces overfitting and helps the models learn more robust visual representations.

4.5 Deep Learning Model Evaluation

To evaluate the effectiveness of different architectures for UML recognition, multiple deep learning models were investigated. The selected models include both transfer learning-based architectures and a custom CNN model.

The evaluated models are:

4.5.1 Inception-V3

Inception-V3 is a deep convolutional neural network architecture designed to efficiently extract complex visual features from images. The architecture improves traditional CNN models by introducing Inception modules, where multiple convolution operations with different filter sizes are performed in parallel. These parallel operations enable the network to capture features at different spatial levels while reducing computational complexity.

The Inception module combines 1×1, 3×3, and factorized convolutions to learn both low-level and high-level visual patterns. The 1×1 convolution operation is used for dimensionality reduction and feature transformation, while larger convolution filters capture spatial information from images. This design allows Inception-V3 to learn detailed structural patterns from complex visual inputs.

In this research, Inception-V3 was applied as a transfer learning-based model for UML diagram recognition. The overall architecture of the modified Inception-V3 model used for UML classification is illustrated in Figure 4.2.

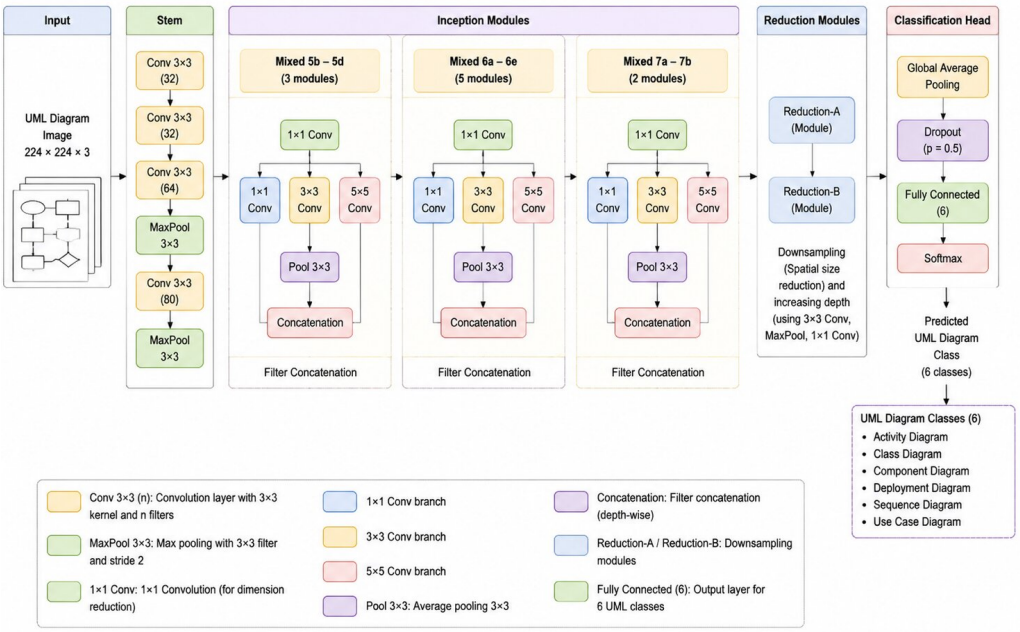


Figure 4.2: Inception-V3 architecture used for UML diagram classification.

The original ImageNet-pretrained Inception-V3 model was adapted to analyze UML diagram

images by replacing its final classification layer with a new output layer corresponding to the six UML diagram categories used in this study. The convolutional layers of the pretrained network were utilized as feature extractors to learn important visual characteristics from UML diagrams, including diagram structures, graphical relationships, component arrangements, and visual patterns.

Before training, all UML diagram images were resized to 224×224 pixels to match the required input format of the model. The extracted feature representations were then passed through the modified classification layer to predict the corresponding UML category.

During the training phase, the model parameters were optimized using the training dataset, while the validation dataset was used to monitor model performance and reduce overfitting. The trained model was finally evaluated using the testing dataset.

The performance of Inception-V3 was measured using standard classification metrics, including accuracy, precision, recall, and F1-score. The obtained results were compared with other deep learning architectures to analyze the effectiveness of Inception-V3 for automated UML diagram recognition.

The use of Inception-V3 in this study demonstrates the capability of transfer learning-based CNN architectures to understand UML diagrams as visual software artifacts and automatically classify different UML diagram categories.

4.5.2 Xception

Xception is a deep convolutional neural network architecture that utilizes depthwise separable convolution operations to improve feature extraction efficiency while reducing computational complexity. Unlike traditional CNN architectures, Xception separates spatial feature extraction and channel-wise feature transformation, allowing the model to learn more efficient visual representations.

The architecture consists of three main flow stages: Entry Flow, Middle Flow, and Exit Flow. The Entry Flow performs initial feature extraction and spatial dimension reduction, while the Middle Flow applies repeated depthwise separable convolution blocks to learn complex feature representations. The Exit Flow generates high-level features that are used for final classification.

In this research, Xception was applied as a transfer learning-based model for UML diagram recognition. The overall architecture of the modified Xception model used for UML classification is illustrated in Figure 4.3.

The original ImageNet-pretrained Xception model was adapted by replacing the final classification layer with a customized output layer containing six classes corresponding to the UML diagram categories used in this study. The pretrained convolutional layers were used to extract important visual features from UML images, including structural arrangements, graphical relationships, and diagram-specific patterns.

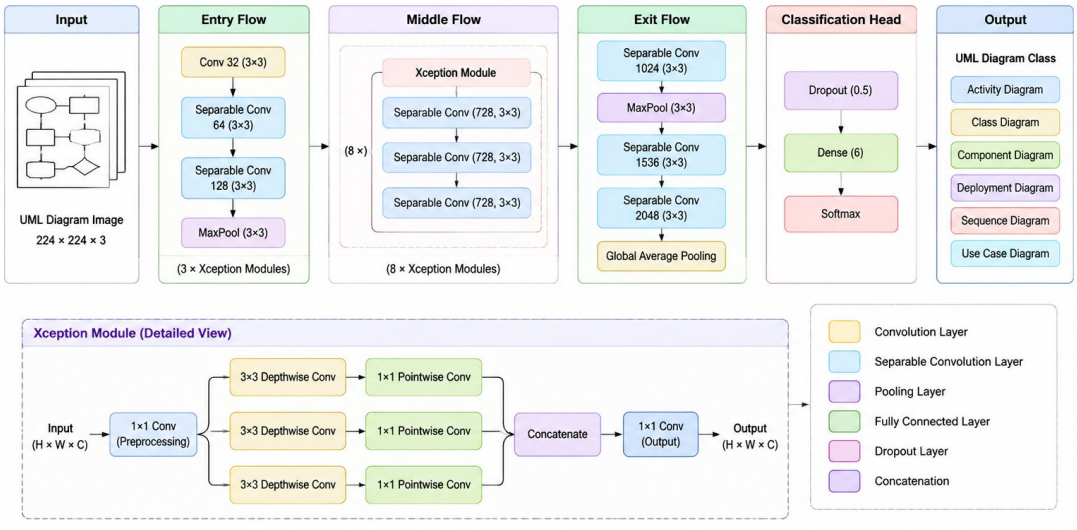


Figure 4.3: Xception architecture used for UML diagram classification.

All UML diagram images were resized to 224×224 pixels before being provided to the model. During training, the extracted feature representations were passed through the modified classification head to predict the corresponding UML category.

The performance of Xception was evaluated using accuracy, precision, recall, and F1-score. The obtained results were compared with other CNN-based models to analyze the effectiveness of depthwise separable convolution for automated UML diagram recognition.

The use of Xception in this study demonstrates the capability of efficient CNN architectures to learn complex visual representations from UML diagrams while reducing computational cost.

4.5.3 MobileNet-V3

MobileNet-V3 is a lightweight convolutional neural network architecture designed for efficient image classification with reduced computational cost. It combines depthwise separable convolutions, inverted residual blocks, and Squeeze-and-Excitation (SE) modules to achieve a balance between accuracy and computational efficiency.

The architecture improves mobile-oriented deep learning by using optimized network blocks and the Hard-Swish activation function. Depthwise separable convolutions reduce the number of parameters and operations, while SE modules enhance important feature channels by adaptively recalibrating feature representations.

In this research, MobileNet-V3 was applied as a transfer learning-based model for UML diagram recognition. The modified MobileNet-V3 architecture used for UML classification is illustrated in Figure 4.4.



Figure 4.4: MobileNet-V3 architecture used for UML diagram classification.

The original ImageNet-pretrained MobileNet-V3 model was adapted by replacing the final classification layer with a customized output layer containing six UML diagram categories used in this study. The pretrained feature extraction layers were utilized to learn important visual characteristics from UML diagrams, including structural patterns, component arrangements, and diagram relationships.

All UML diagram images were resized to 224×224 pixels before being provided to the network. During training, the extracted feature representations were passed through the modified classification head to predict the corresponding UML diagram category.

The performance of MobileNet-V3 was evaluated using accuracy, precision, recall, and F1-score. The results were compared with other deep learning architectures to analyze the effectiveness of lightweight CNN models for automated UML diagram recognition.

The application of MobileNet-V3 in this study demonstrates the potential of efficient deep learning architectures for software artifact analysis, especially in scenarios where computational resources are limited.

4.5.4 YOLO Network

YOLO (You Only Look Once) is an object detection-based deep learning architecture designed for fast and efficient visual recognition. Unlike traditional image classification approaches that assign a single class label to an entire image, YOLO analyzes image regions and simultaneously identifies important visual features and class information.

The YOLO architecture used in this research is illustrated in Figure 4.5. The framework follows a single-stage detection approach where feature extraction, feature aggregation, and classification are performed within a unified network.

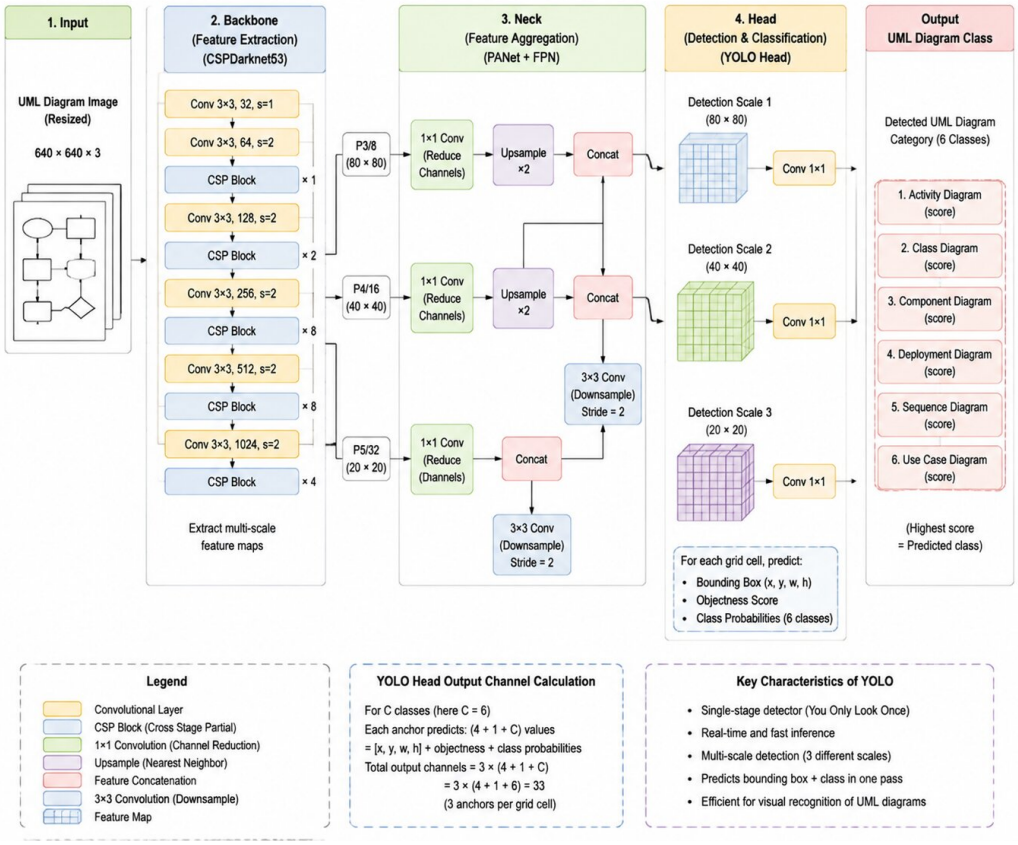


Figure 4.5: YOLO architecture flowchart used for UML diagram classification.

As shown in Figure 4.5, the proposed YOLO-based UML recognition framework consists of three major components: backbone, neck, and detection head. The backbone network extracts hierarchical visual features from UML diagram images, while the neck module combines features from different scales to improve recognition capability.

The detection head generates the final predictions by analyzing the extracted features and identifying the corresponding UML diagram category. In this study, the YOLO model was adapted to recognize six UML diagram classes: Activity Diagram, Class Diagram, Component Diagram, Deployment Diagram, Sequence Diagram, and Use Case Diagram.

The input UML diagram images were processed and provided to the YOLO network for feature extraction and classification. The model learns important visual characteristics such as diagram structures, object arrangements, connections, and spatial relationships between different UML elements.

The trained YOLO model was evaluated using standard classification metrics, including accuracy, precision, recall, and F1-score. The obtained performance was compared with other deep learning architectures to analyze the effectiveness of object detection-based learning for

automated UML diagram recognition.

The application of YOLO in this research demonstrates the capability of single-stage detection models to efficiently analyze UML diagrams and provide fast automated recognition of software engineering visual artifacts.

4.5.5 Custom Layered CNN

A custom layered CNN architecture was developed specifically for UML diagram recognition. Unlike pretrained transfer learning models, the proposed CNN was designed to learn UML-specific visual features directly from the dataset. The model focuses on extracting meaningful characteristics from diagram structures, shapes, connections, and visual relationships.

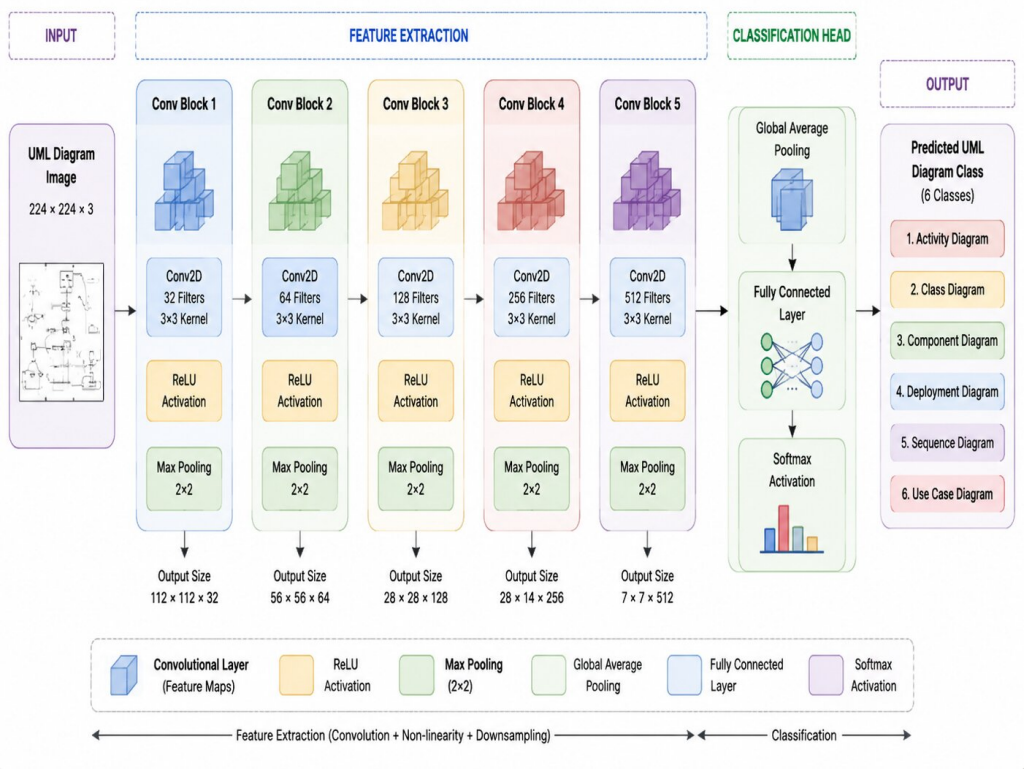


Figure 4.6: Custom layered CNN architecture flowchart used for UML diagram classification.

The overall architecture of the proposed custom layered CNN model is illustrated in Figure 4.6. The architecture consists of multiple convolutional layers followed by activation functions, pooling operations, and classification layers to perform automated UML diagram recognition.

As shown in Figure 4.6, the input UML diagram images are first passed through a sequence of convolutional layers. These layers progressively extract low-level and high-level visual features from the input images. The number of filters increases gradually in deeper layers, allowing the network to learn more complex representations of UML diagram patterns.

The convolutional feature extraction process is followed by pooling operations to reduce spa-

tial dimensions while preserving important information. The extracted feature maps are then passed through fully connected layers and a softmax classification layer to predict the corresponding UML diagram category.

In this study, the custom CNN model was trained to classify six UML diagram categories: Activity Diagram, Class Diagram, Component Diagram, Deployment Diagram, Sequence Diagram, and Use Case Diagram. The model performance was evaluated using accuracy, precision, recall, and F1-score and compared with other deep learning architectures.

The development of this task-specific CNN architecture enables the model to learn UML-specific visual characteristics and provides an effective approach for automated software artifact recognition.

4.6 Model Evaluation and UML Recognition Decision

After training, the performance of all models was compared using evaluation metrics including accuracy, precision, recall, and F1-score.

The comparative analysis identifies the most effective model for automated UML diagram recognition. Based on experimental results, the selected model is used for final UML recognition decision making.

The complete process from image collection to final UML recognition provides an automated approach for understanding software design artifacts and reduces the dependency on manual diagram analysis.

4.7 Dataset Description

A high-quality and balanced dataset is essential for developing a reliable deep learning-based UML recognition system. In this research, a dedicated UML diagram image dataset was prepared containing multiple categories of UML diagrams commonly used in software engineering.

The dataset consists of six UML diagram classes representing different structural and behavioral aspects of software systems. The selected UML categories are Activity Diagram, Class Diagram, Component Diagram, Deployment Diagram, Sequence Diagram, and Use Case Diagram.

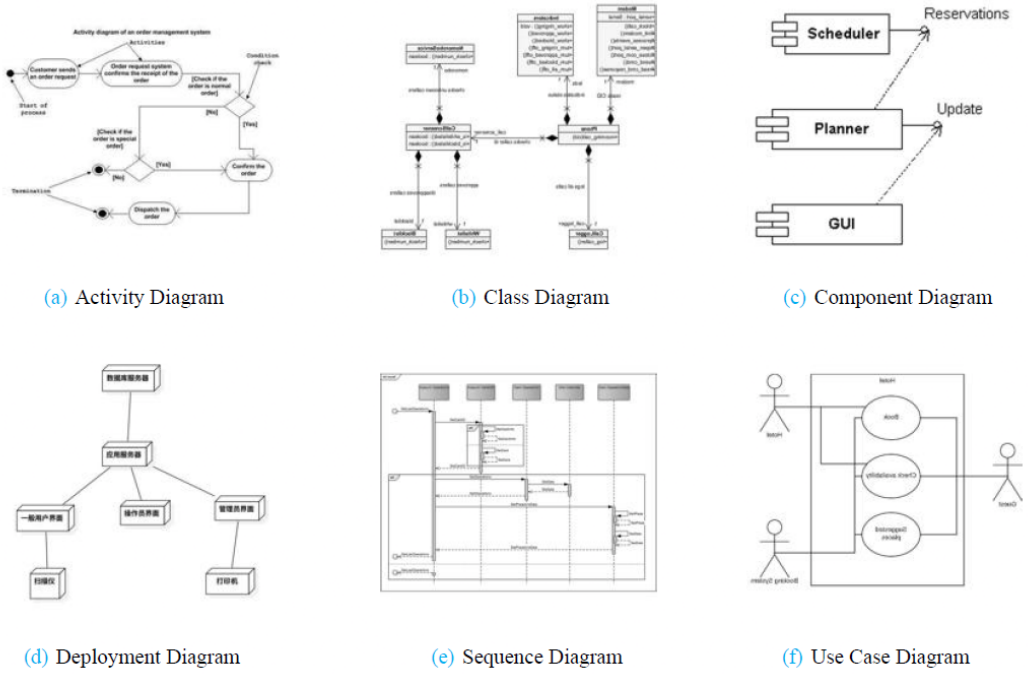
Table 4.1: Distribution of UML Diagram Dataset

UML Diagram Category	Number of Images
Activity Diagram	1111
Class Diagram	1111
Component Diagram	1111
Deployment Diagram	1111
Sequence Diagram	1111
Use Case Diagram	1111
Total	**6666**

The dataset contains a total of 6,666 UML diagram images, with 1,111 images for each category. The balanced distribution ensures that the learning models receive equal representation from all UML classes and prevents bias towards any particular category.

4.8 UML Diagram Samples

Figure 4.7 presents representative examples from the six UML diagram categories included in the dataset.



(a) Activity Diagram (b) Class Diagram (c) Component Diagram

(d) Deployment Diagram (e) Sequence Diagram (f) Use Case Diagram

Figure 4.7: Representative samples of different UML diagram categories.

The figure demonstrates the visual diversity among different UML diagram types. Each

category contains unique structural characteristics and graphical patterns. For example, class diagrams mainly represent classes and their relationships, while sequence diagrams focus on interactions between system components over time.

This visual variation creates a challenging classification problem because different UML diagrams may contain similar graphical elements but represent different software concepts.

4.9 Dataset Splitting Strategy

To evaluate the generalization capability of the proposed deep learning models, the dataset was divided into training, validation, and testing subsets.

The dataset splitting strategy followed an 80:10:10 ratio:

Table 4.2: Dataset Splitting Strategy

Dataset	**Percentage**	**Number of Images**
Training	80%	5332
Validation	10%	667
Testing	10%	667
Total	**100%**	**6666**

The training set was used for learning model parameters, while the validation set was used for monitoring training performance and optimizing model configurations. The independent testing set was reserved for final evaluation to measure the actual recognition capability of the trained models.

4.10 Input Image Configuration

Before being provided to the deep learning models, all UML diagram images were standardized into a fixed input format.

The following preprocessing configuration was applied:

- Image size: 224×224 pixels
- Input format: RGB image
- Dataset normalization applied before training
- Data augmentation applied during training

The standardized image representation allows different deep learning architectures to receive consistent input and enables fair comparison among the evaluated models.

4.11 Deep Learning Model Architecture and Training Configuration

To evaluate the effectiveness of different deep learning architectures for UML diagram recognition, multiple convolutional neural network-based models were implemented and compared. The selected models include both transfer learning approaches and a custom-designed CNN architecture.

The objective of evaluating multiple architectures was to identify the model that provides the most effective feature extraction capability for UML diagram images.

4.11.1 Transfer Learning-Based Models

Transfer learning allows previously trained deep learning models to be adapted for new classification tasks. Instead of training a complete network from scratch, pre-trained feature extraction capabilities are utilized and fine-tuned for UML diagram recognition.

Inception-V3 Architecture

Inception-V3 is a deep convolutional neural network architecture designed for efficient image classification. It uses multiple convolution operations with different filter sizes to capture information at different levels of abstraction.

For UML recognition, the pre-trained Inception-V3 model was adapted by replacing the original classification layer with a new classifier containing six output classes corresponding to the UML diagram categories.

The model extracts hierarchical visual features such as shapes, edges, connections, and structural patterns from UML diagrams.

Xception Architecture

Xception is an advanced CNN architecture based on depthwise separable convolution. It separates spatial and channel-wise feature extraction, reducing computational complexity while maintaining strong representation capability.

The architecture was evaluated to determine its effectiveness in recognizing different UML diagram patterns with high visual similarity.

MobileNet-V3 Architecture

MobileNet-V3 is a lightweight neural network architecture designed for efficient deployment on resource-constrained environments. It combines depthwise separable convolution, squeeze-and-excitation mechanisms, and optimization techniques to achieve a balance between accuracy and efficiency.

The model was included in this research because lightweight architectures are important for practical software engineering applications where fast inference is required.

4.11.2 Custom CNN Architecture

In addition to transfer learning models, a custom convolutional neural network was developed specifically for UML diagram recognition.

The custom CNN architecture consists of multiple convolutional layers followed by feature extraction and classification layers. The network gradually increases feature learning capacity by extracting low-level and high-level visual characteristics from UML diagrams.

The architecture is designed to learn UML-specific patterns, including:

- Diagram shapes and symbols
- Relationship connections
- Structural layouts
- Visual characteristics among UML categories

Unlike pre-trained models that learn from general image datasets, the custom CNN learns directly from UML-specific visual information.

4.11.3 YOLO-Based Recognition Model

A YOLO-based approach was also investigated for UML diagram recognition. YOLO (You Only Look Once) is a real-time object detection architecture that performs recognition by analyzing image regions.

The motivation behind using YOLO was to evaluate whether object detection approaches could provide effective solutions for software diagram understanding.

The YOLO model provides fast prediction capability and can potentially support real-time UML recognition applications.

4.12 Training Configuration

All deep learning models were trained using the prepared UML dataset under the same experimental conditions to ensure a fair comparison.

The general training configuration is presented in Table 4.3.

Table 4.3: Deep Learning Training Configuration

Parameter	Configuration
Input Image Size	224×224
Number of Classes	6
Dataset Split	80% Training, 10% Validation, 10% Testing
Color Format	RGB
Evaluation Method	Holdout Validation
Performance Metrics	Accuracy, Precision, Recall, F1-score

4.13 Model Evaluation Strategy

The trained models were evaluated using the independent test dataset. Performance comparison was performed based on multiple evaluation metrics to identify the most effective architecture.

The evaluation process included:

- Overall classification accuracy comparison.
- Class-wise performance analysis.
- Confusion matrix evaluation.
- Error analysis among UML categories.

The best-performing model was selected based on its ability to accurately recognize different UML diagram categories while maintaining reliable generalization performance on unseen images.

4.14 Experimental Results and Performance Analysis

This section presents the experimental results of the evaluated deep learning models for UML diagram recognition. The performance of each model was analyzed using the test dataset containing unseen UML diagram images.

The objective of this evaluation was to compare different deep learning architectures and identify the most effective model for automated UML diagram classification.

4.14.1 Performance Comparison of Deep Learning Models

The performance of the evaluated deep learning models was compared to analyze their effectiveness for automated UML diagram recognition. Five different architectures, including MobileNet-V3, Xception, Inception-V3, Custom CNN, and YOLO, were evaluated using four standard classification metrics: accuracy, precision, recall, and F1-score.

Table 4.4: Performance comparison of deep learning models for UML diagram recognition

Model	Class Name	Precision	Recall	F1-score	Accuracy
MobileNet-V3	Activity Diagram	0.97	0.97	0.97	0.96
	Class Diagram	0.95	0.94	0.95	0.96
	Component Diagram	0.94	0.93	0.94	0.96
	Deployment Diagram	0.95	0.95	0.95	0.96
	Sequence Diagram	0.97	0.98	0.98	0.96
	Use Case Diagram	1.00	1.00	1.00	0.96
Xception	Activity Diagram	0.39	0.53	0.45	0.49
	Class Diagram	0.49	0.51	0.50	0.49
	Component Diagram	0.39	0.27	0.32	0.49
	Deployment Diagram	0.49	0.35	0.41	0.49
	Sequence Diagram	0.51	0.55	0.53	0.49
	Use Case Diagram	0.67	0.77	0.71	0.49
Inception-V3	Activity Diagram	0.50	0.40	0.44	0.50
	Class Diagram	0.46	0.62	0.53	0.50
	Component Diagram	0.31	0.28	0.30	0.50
	Deployment Diagram	0.45	0.54	0.49	0.50
	Sequence Diagram	0.53	0.54	0.54	0.50
	Use Case Diagram	0.82	0.62	0.71	0.50

Model	Class Name	Precision	Recall	F1-score	Accuracy
Custom CNN	Activity Diagram	0.92	0.86	0.89	0.76
	Class Diagram	0.75	0.56	0.64	0.76
	Component Diagram	0.54	0.74	0.62	0.76
	Deployment Diagram	0.75	0.73	0.74	0.76
	Sequence Diagram	0.80	0.78	0.79	0.76
	Use Case Diagram	0.87	0.86	0.87	0.76
YOLO	Activity Diagram	0.99	1.00	1.00	0.99
	Class Diagram	0.98	0.99	0.99	0.99
	Component Diagram	0.98	1.00	0.99	0.99
	Deployment Diagram	1.00	0.96	0.98	0.99
	Sequence Diagram	1.00	1.00	1.00	0.99
	Use Case Diagram	1.00	0.99	0.99	0.99

The detailed class-wise performance comparison of all evaluated models across the six UML diagram categories is presented in Table 4.4. The evaluated UML classes include Activity Diagram, Class Diagram, Component Diagram, Deployment Diagram, Sequence Diagram, and Use Case Diagram. This comparison provides insights into the capability of different deep learning architectures for learning visual patterns and structural characteristics from UML diagrams.

The experimental results show significant variations among the evaluated architectures. MobileNet-V3 achieved strong performance with an accuracy of 96%, demonstrating the effectiveness of lightweight CNN architectures for UML diagram recognition. The model achieved excellent results for most UML categories, particularly for the Use Case Diagram category where it obtained an F1-score of 1.00.

Xception and Inception-V3 achieved comparatively lower performance with accuracies of 49% and 50%, respectively. Although these architectures are highly effective for general image classification tasks, their learned feature representations were less suitable for capturing the unique structural characteristics of UML diagrams.

The Custom CNN model achieved an accuracy of 76%. The improved performance compared with Xception and Inception-V3 indicates that a task-specific CNN architecture can effectively learn UML-related visual features. The model was able to capture important diagram

structures, shapes, and relationships through customized feature extraction layers.

Among all evaluated models, YOLO achieved the highest performance with an accuracy of 99%. As presented in Table 4.4, YOLO obtained near-perfect precision, recall, and F1-score values across all UML categories. This superior performance indicates that object detection-based learning is highly effective for recognizing spatial and structural patterns within UML diagrams.

Overall, the comparison demonstrates that deep learning techniques can successfully automate UML diagram recognition. The results highlight YOLO as the most effective architecture for this task, while lightweight models such as MobileNet-V3 also provide competitive performance with lower computational requirements.

4.14.2 Analysis of Model Performance

The superior performance of YOLO can be attributed to its object detection mechanism, which allows the model to capture important spatial features from UML diagrams. Unlike traditional image classification approaches, YOLO analyzes image regions and learns discriminative visual patterns associated with different UML categories.

MobileNet-V3 also demonstrated strong performance while maintaining a lightweight architecture. This indicates that efficient deep learning models can provide practical solutions for UML recognition applications where computational resources may be limited.

The performance differences among models indicate that architecture selection plays an important role in automated UML understanding. Models with stronger feature extraction capability can better distinguish between visually similar UML diagram categories.

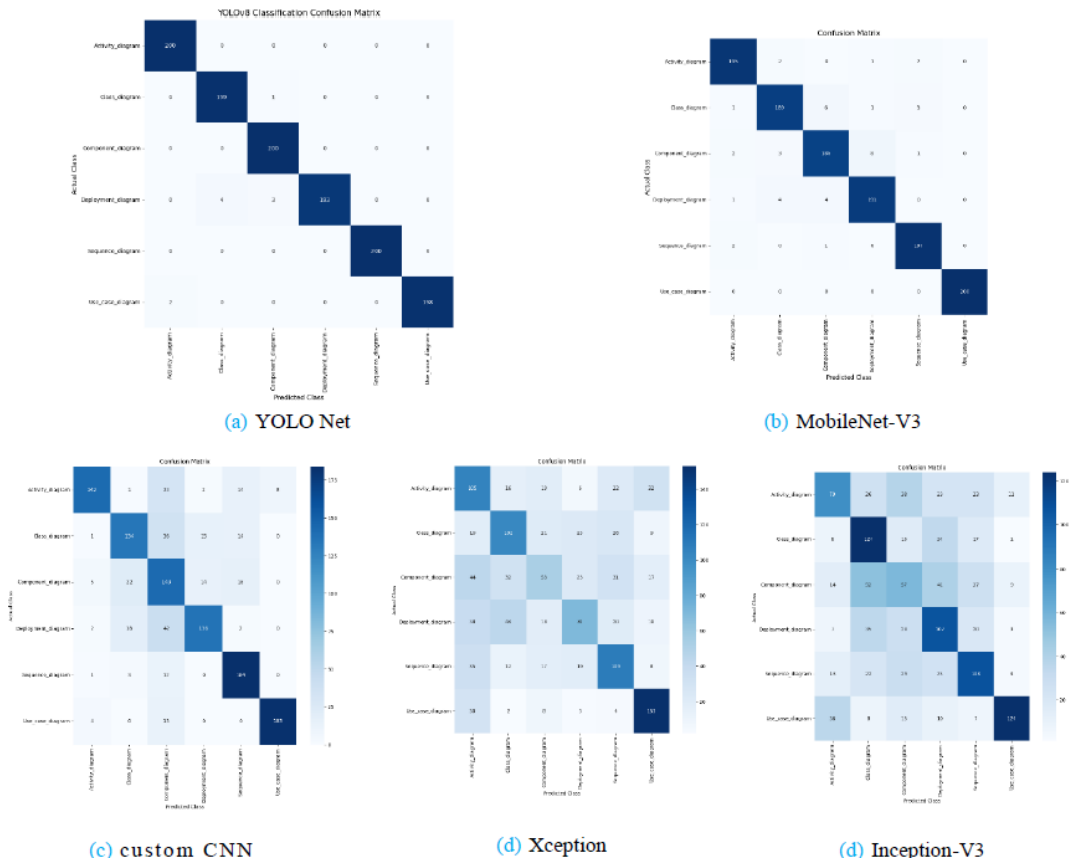
4.15 Confusion Matrix Analysis

Confusion matrix analysis was performed to understand class-wise prediction behavior of the trained models.

The confusion matrix provides detailed information about correct and incorrect predictions among different UML categories. High diagonal values indicate successful recognition, while off-diagonal values represent classification confusion between different UML classes.

The analysis shows that most UML categories were correctly identified, demonstrating the effectiveness of the proposed deep learning-based approach.

Figure 4.8 presents the confusion matrix analysis of the UML recognition models.



(a) YOLO Net (b) MobileNet-V3

(c) custom CNN (d) Xception (d) Inception-V3

Figure 4.8: Confusion matrix analysis of deep learning models for UML diagram recognition.

4.16 Error Analysis

Although the proposed models achieved strong performance, some misclassifications occurred due to similarities among UML diagram structures.

The main causes of classification errors include:

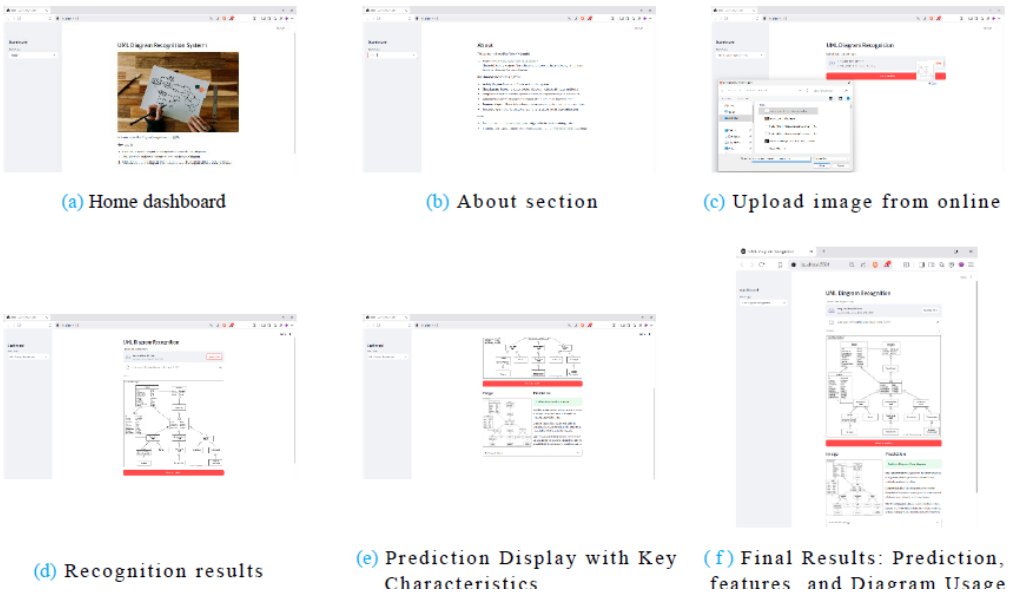
- Similar visual layouts among different UML categories.
- Complex diagram structures containing overlapping elements.
- Variations in diagram styles and formatting.
- Limited representation of some visual patterns.

These challenges indicate that future UML recognition systems may benefit from combining visual feature extraction with semantic understanding approaches.

4.17 UML Recognition Application Development

To demonstrate the practical applicability of the proposed approach, a web application interface was developed for automated UML diagram recognition.

The application allows users to upload UML diagram images and obtain the predicted UML category using the trained deep learning model.



(a) Home dashboard (b) About section (c) Upload image from online

(d) Recognition results (e) Prediction Display with Key Characteristics (f) Final Results: Prediction, features, and Diagram Usage

Figure 4.9: Web-based UML diagram recognition application.

The deployment of the recognition model into a web-based environment demonstrates the practical usability of the proposed framework. Such a system can support software engineers, students, and developers in understanding UML diagrams automatically.

4.18 Summary

This chapter presented the deep learning-based UML diagram recognition approach developed in this thesis. The proposed framework analyzes UML images using multiple deep learning architectures and identifies the most effective model for automated classification.

The experimental results demonstrate that deep learning techniques can effectively support software requirement understanding through visual analysis. The developed recognition system reduces manual effort and provides a foundation for intelligent software engineering applications.

Chapter 5

LLM-Based Design Pattern Recognition

5.1 Introduction

This chapter presents the second major contribution of this thesis, which focuses on automated software design pattern recognition using large language models (LLMs) and code representation learning techniques.

Software design patterns provide reusable solutions to commonly occurring software design problems. Identifying these patterns from existing source code is important for software maintenance, reverse engineering, and architectural understanding. However, manual identification of design patterns from large software systems is challenging because developers need extensive programming knowledge and experience.

Traditional design pattern detection approaches mainly depend on handcrafted rules, software metrics, and structural analysis. Although these approaches have achieved considerable success, they often fail when patterns are implemented using different programming styles or modified structures.

Recent advances in large language models have introduced new possibilities for software code understanding. LLMs can learn contextual and semantic representations from source code, enabling automated analysis of complex software structures.

In this research, multiple pre-trained code language models are evaluated for generating source code embeddings. These embeddings are then classified using a machine learning classifier to identify different software design patterns.

5.2 Proposed Design Pattern Recognition Framework

The proposed design pattern recognition framework consists of four major stages:

1. Source code dataset preparation

2. Code embedding generation using pre-trained language models

3. Design pattern classification using machine learning

4. Performance evaluation and analysis

The overall workflow is illustrated in Figure 5.1.

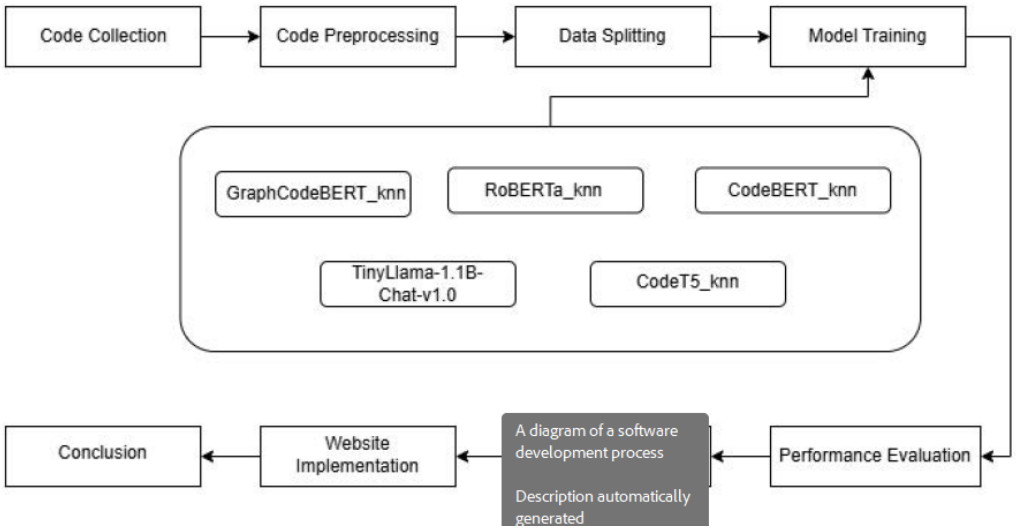


Figure 5.1: Framework of LLM-based design pattern recognition using code embeddings and machine learning classification.

The input source code samples are first processed and transformed into semantic vector representations using pre-trained language models. These representations capture important programming structures and contextual relationships. The generated embeddings are then provided to a classifier for automated design pattern recognition.

5.3 Dataset Description

The experiments were conducted using the P-MARt dataset, which is a benchmark dataset containing Java source code implementations of different software design patterns.

The dataset consists of 93 Java programs categorized into six classes:

- Abstract Factory
- Builder
- Factory Method
- Prototype
- Singleton
- Non-Design Pattern

The inclusion of non-design pattern examples makes the classification problem more realistic because practical software systems contain both pattern-based and normal implementations.

The dataset distribution enables evaluation of the capability of language models to distinguish between different design pattern implementations.

Table 5.1: Design Pattern Dataset Categories

Category	Description
Abstract Factory	Creation of related object families
Builder	Step-by-step object construction
Factory Method	Object creation through inheritance
Prototype	Object cloning mechanism
Singleton	Single instance object creation
Non-Design Pattern	Normal source code implementation

5.4 Source Code Preprocessing

Before generating embeddings, source code samples were prepared to ensure consistent input representation for language models.

The preprocessing stage included:

- Reading Java source files from the dataset.
- Removing unnecessary formatting variations.
- Preparing code sequences compatible with language models.
- Generating tokenized representations.

The objective of preprocessing is to preserve important programming structures while reducing irrelevant variations among source code samples.

5.5 Code Representation and Embedding Generation

The performance of design pattern recognition largely depends on the quality of source code representations. Traditional approaches usually rely on manually defined features such as software metrics and structural properties. However, these approaches may not capture the semantic information embedded within source code.

To overcome these limitations, this research investigates pre-trained language models for generating meaningful code embeddings. These models learn contextual representations from source code and transform programming structures into numerical feature vectors suitable for classification.

5.5.1 Language Models for Code Embedding

Five different pre-trained language models were evaluated in this research:

- GraphCodeBERT
- CodeBERT
- CodeT5
- RoBERTa
- TinyLlama

GraphCodeBERT

GraphCodeBERT is a pre-trained code representation model specifically designed for programming languages. Unlike traditional code language models, GraphCodeBERT incorporates both token-level information and data flow relationships to learn structural dependencies within source code.

The model extends transformer-based code representation learning by integrating data flow information with source code tokens. This enables GraphCodeBERT to capture important relationships among variables, methods, and program structures, which are essential for understanding complex software implementations.

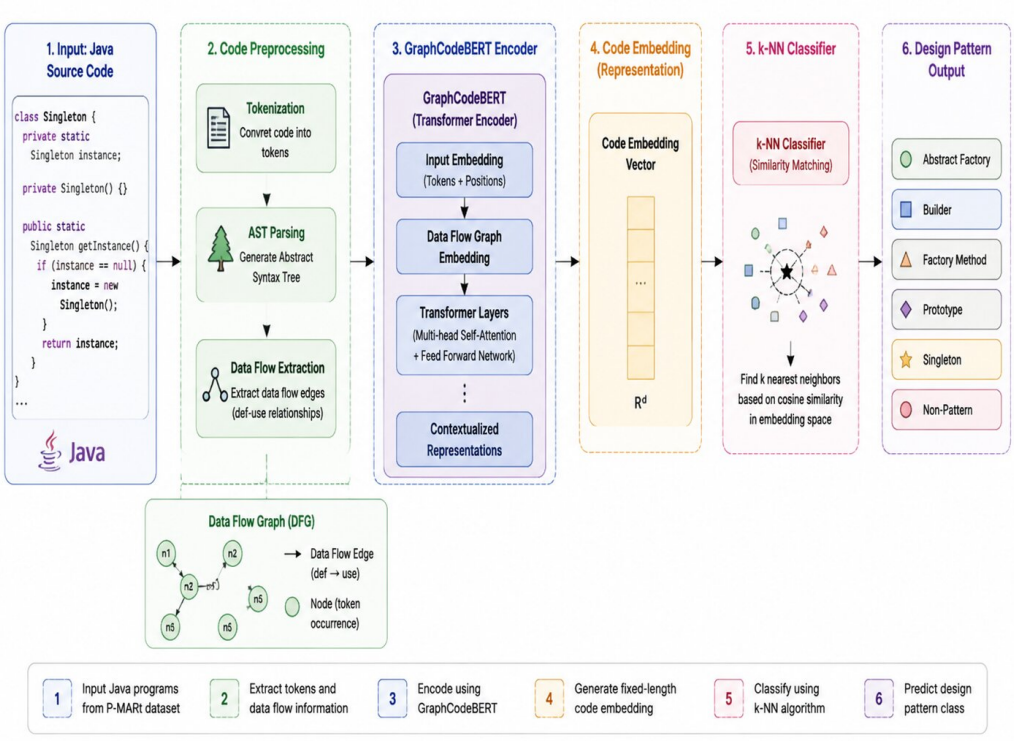


Figure 5.2: GraphCodeBERT architecture for design pattern recognition.

In this research, GraphCodeBERT was used to generate semantic code embeddings from Java source code programs. The generated representations were used to identify software design patterns by providing meaningful feature vectors to the k-Nearest Neighbor (k-NN) classifier.

The overall GraphCodeBERT-based design pattern recognition architecture is illustrated in Figure 5.2. As shown in Figure 5.2, the framework processes Java source code through preprocessing, token representation, data flow extraction, and GraphCodeBERT encoding. The extracted contextual embeddings are then passed to the k-NN classifier for predicting the corresponding design pattern category.

CodeBERT

CodeBERT is a transformer-based pre-trained model developed for learning representations from both natural language and programming language data. It captures contextual information from source code by learning relationships between code tokens and their surrounding contexts.

Unlike traditional feature-based code analysis methods, CodeBERT uses transformer-based representation learning to understand semantic and syntactic characteristics of source code. The model learns meaningful contextual representations from large-scale code and natural language pairs, which enables it to capture important information related to variables, methods, and program structures.

In this research, CodeBERT was utilized to generate meaningful code embeddings for automated design pattern recognition. Java source code samples from the dataset were processed and converted into contextual vector representations using the pre-trained CodeBERT model. These embeddings were then provided as feature representations to the k-Nearest Neighbor (k-NN) classifier for identifying different software design pattern categories.

The overall CodeBERT-based design pattern recognition architecture is illustrated in Figure 5.3. As shown in Figure 5.3, the framework first preprocesses Java source code through tokenization and code representation generation. The processed code tokens are then passed through the CodeBERT transformer encoder to learn context-aware representations. Finally, the generated code embeddings are classified using the k-NN algorithm to predict the corresponding design pattern category.

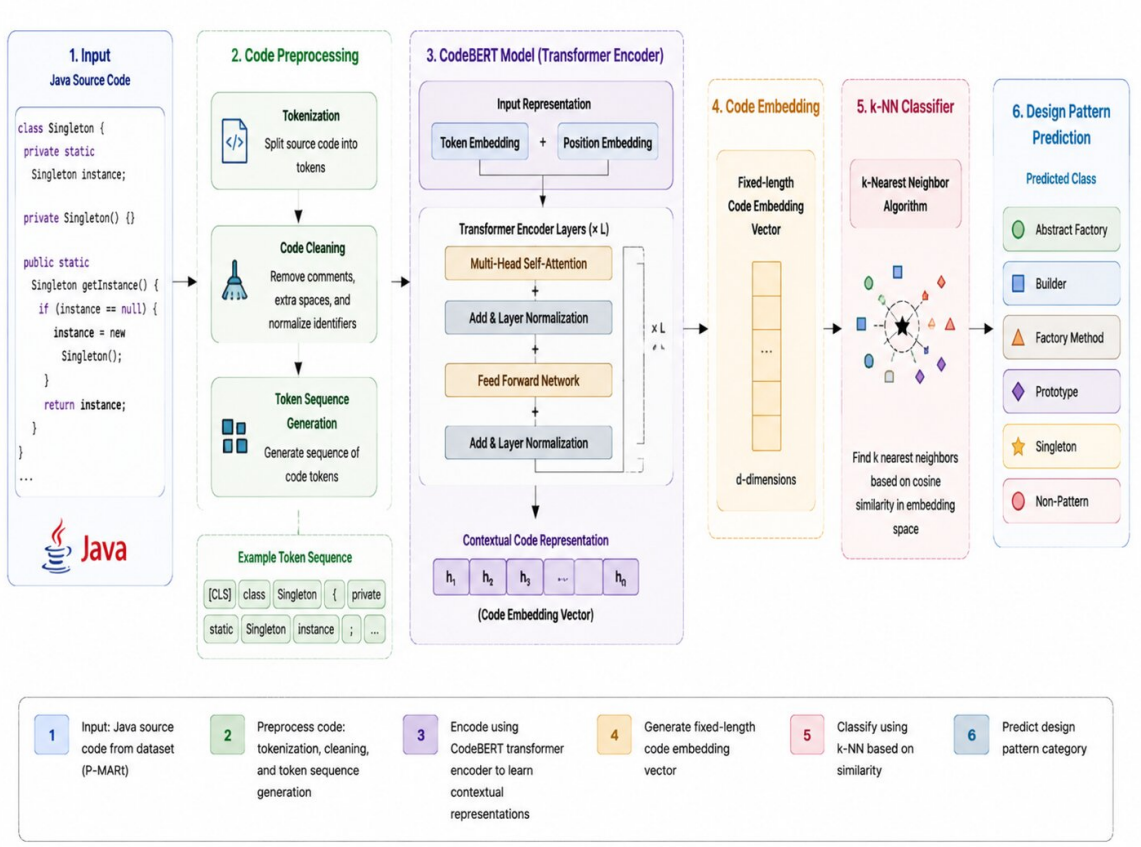


Figure 5.3: CodeBERT architecture for design pattern recognition.

CodeT5

CodeT5 is a unified pre-trained model specifically designed for programming language understanding and code generation tasks. It follows an encoder-decoder transformer architecture that learns meaningful representations from source code by capturing both syntactic structures and semantic relationships.

Unlike traditional code analysis approaches, CodeT5 can understand contextual information from programming languages through large-scale pre-training. The encoder component learns rich code representations, while the decoder component supports sequence-based code understanding and generation tasks.

In this research, CodeT5 was evaluated for automated software design pattern recognition. Java source code samples were processed using CodeT5 to generate context-aware code embeddings. These embeddings were used as feature representations for the k-Nearest Neighbor (k-NN) classifier to distinguish different design pattern implementations.

The overall CodeT5-based design pattern recognition pipeline is illustrated in Figure 5.4. As shown in Figure 5.4, the framework first preprocesses Java source code through tokenization and code structure analysis. The processed representations are then passed into the CodeT5 encoder-decoder model to generate contextual code embeddings. Finally, the generated embeddings are classified using the k-NN algorithm to predict the corresponding design pattern category.

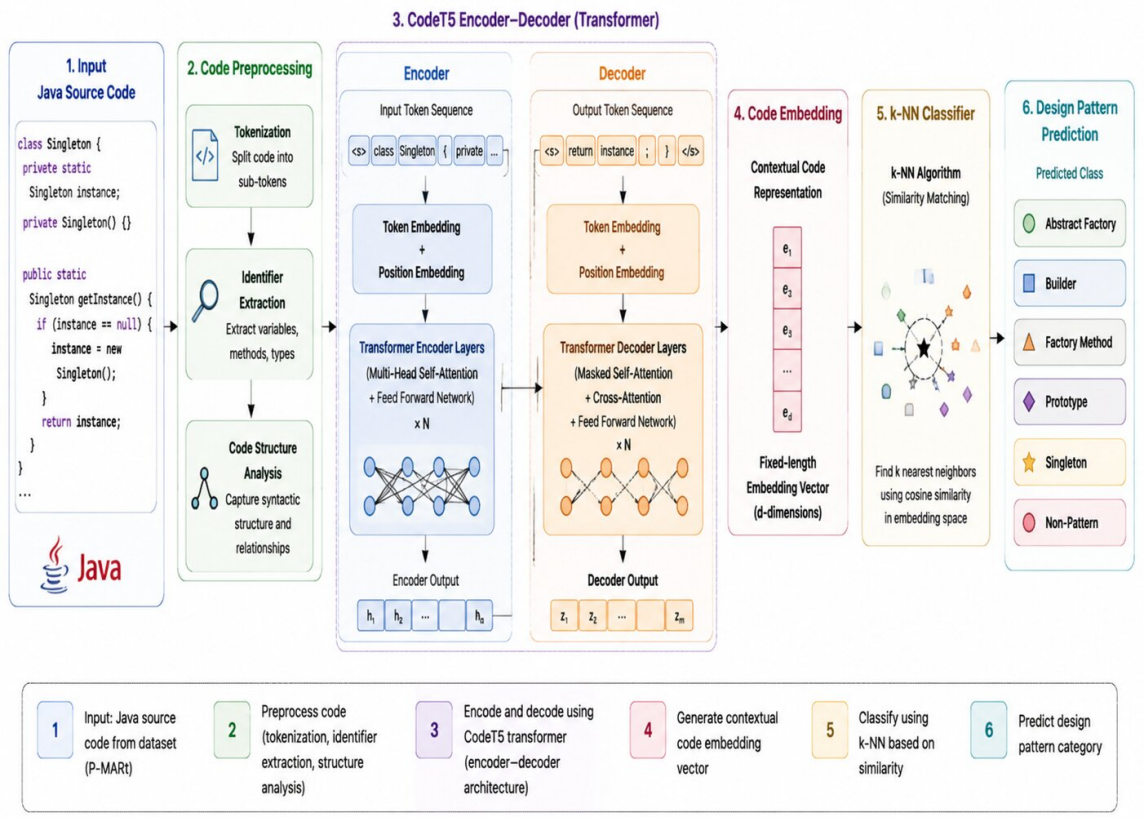


Figure 5.4: CodeT5 pipeline for design pattern recognition.

RoBERTa

RoBERTa (Robustly Optimized BERT Approach) is an enhanced transformer-based language representation model introduced by Liu et al. [67]. It was developed as an optimized variant of BERT by improving the pre-training strategy, including training with larger datasets, longer training durations, larger batch sizes, and dynamic masking techniques. Unlike the original BERT model, RoBERTa removes the next sentence prediction objective and focuses primarily on masked language modeling, which enables it to learn richer contextual representations.

Although RoBERTa was initially designed for natural language understanding tasks, its transformer architecture allows it to capture meaningful structural and semantic patterns from programming languages. Source code contains contextual relationships, syntax structures, and dependency information that can be represented through transformer-based embeddings. Therefore, RoBERTa can be applied as a general-purpose code representation model by treating source code tokens as sequential input.

In this study, RoBERTa was included as a baseline representation model to evaluate the effectiveness of transformer-based embeddings for design pattern recognition from source code. The generated embeddings were used to represent the semantic characteristics of Java programs, which were subsequently provided to the classification model. The performance of RoBERTa-based representations was compared with specialized code-oriented models to analyze whether

a general language model can effectively capture software design pattern characteristics.

TinyLlama

TinyLlama is a lightweight open-source large language model (LLM) developed based on the LLaMA architecture with the objective of achieving efficient language understanding and generation capabilities using significantly fewer computational resources. It contains approximately 1.1 billion parameters and is trained on a large-scale dataset containing diverse textual information. Despite its compact size, TinyLlama adopts modern transformer-based architectural improvements, including rotary positional embeddings, multi-head attention mechanisms, and efficient training strategies, enabling it to learn meaningful contextual representations.

Unlike traditional code-specific models, TinyLlama is a general-purpose language model that can capture semantic relationships from different types of sequential data. Since source code also contains structured information, semantic dependencies, and contextual patterns, LLM-based representations can provide useful features for software engineering tasks. The ability of TinyLlama to understand programming-related text makes it a potential candidate for extracting compact code representations while requiring fewer computational resources compared with larger foundation models.

In this study, TinyLlama was evaluated to investigate whether a smaller and resource-efficient large language model can generate meaningful code representations for software design pattern recognition. The model was used to extract feature embeddings from Java source code samples, which were then utilized by the classification framework for identifying different software design patterns. Its performance was compared with larger transformer-based models, including CodeBERT, GraphCodeBERT, CodeT5, and RoBERTa, to analyze the trade-off between model size, computational efficiency, and representation quality.

5.6 Embedding Extraction Process

The embedding generation process transforms source code samples into high-dimensional numerical vectors.

For each Java source code sample, the input sequence is passed through a pre-trained language model. The hidden representations generated by the model are extracted as code embeddings.

The embedding generation process can be represented as:

$$Embedding = LanguageModel(SourceCode) \tag{5.1}$$

where the language model learns semantic and structural characteristics of the source code and produces a vector representation.

The generated embeddings preserve important software characteristics, including:

- Class relationships
- Method interactions
- Object creation mechanisms
- Inheritance structures
- Implementation patterns

These learned representations are then used as input features for the classification model.

5.7 Embedding Visualization Analysis

To analyze the quality of generated representations, dimensionality reduction was performed using the t-SNE technique.

t-SNE converts high-dimensional embedding vectors into a two-dimensional space, allowing visual analysis of clustering behavior among different design pattern categories.

Figure 5.5 presents the t-SNE visualization of embeddings generated from different language models.

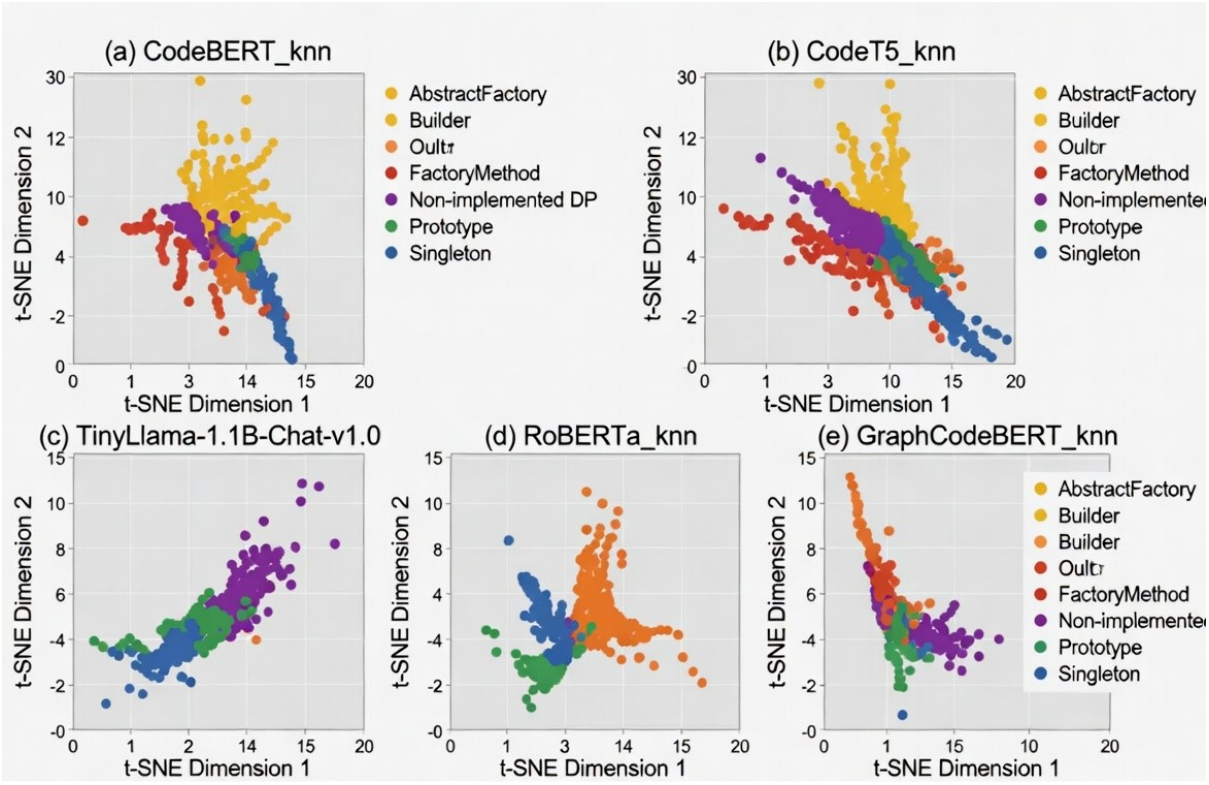


Figure 5.5: t-SNE visualization of code embeddings generated by different language models for design pattern recognition.

The visualization demonstrates that different language models produce different levels of separation among design pattern categories. Models with better semantic understanding generate more distinguishable clusters, resulting in improved classification performance.

The GraphCodeBERT-based embeddings show stronger separation among several design pattern categories because the model considers structural information from source code. This indicates that incorporating program structure can improve automated design pattern recognition.

5.8 Design Pattern Classification Using k-NN

After generating code embeddings from different language models, a k-Nearest Neighbor (k-NN) classifier was applied for automated design pattern recognition.

The objective of using k-NN is to classify unknown source code samples based on their similarity with previously learned code representations. Since the language models transform source code into numerical vectors, distance-based classification becomes effective for identifying similar software design structures.

5.8.1 k-Nearest Neighbor Classification Process

The k-NN classifier determines the class of a new source code sample by calculating its distance from existing training samples.

The classification process consists of the following steps:

1. Generate code embedding vectors using pre-trained language models.
2. Calculate similarity between training and testing embeddings.
3. Select the nearest k samples.
4. Assign the class based on majority voting among neighboring samples.

The Euclidean distance metric was used to measure the similarity between embedding vectors:

$$d(x, y) = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2} \tag{5.2}$$

where x and y represent two embedding vectors and n represents the embedding dimension.

5.8.2 Performance Evaluation of Different Language Models

The performance of different language model embeddings was evaluated using the k-NN classifier. The evaluation was performed using standard classification metrics including accu-

racy, precision, recall, and F1-score.

Table 5.2 presents the comparative performance of different language model-based approaches.

Table 5.2: Performance Comparison of Language Model Embeddings with k-NN

Embedding Model	Accuracy	Precision	Recall	F1-score
CodeBERT + k-NN	0.91	0.91	0.90	0.91
CodeT5 + k-NN	0.95	0.95	0.95	0.95
RoBERTa + k-NN	0.94	0.94	0.94	0.94
TinyLlama + k-NN	0.58	0.752	0.53	0.58
GraphCodeBERT + k-NN	0.97	0.97	0.97	0.969

The experimental results demonstrate that language model-based representations are effective for software design pattern recognition. Among all evaluated approaches, GraphCodeBERT combined with k-NN achieved the highest classification performance.

The superior performance of GraphCodeBERT can be explained by its ability to capture structural information from source code. Unlike models that only consider sequential token information, GraphCodeBERT incorporates relationships between programming elements, enabling better understanding of software design structures.

5.9 Classification Analysis

The classification results indicate that different design patterns have different levels of recognition difficulty.

Patterns such as Singleton and Factory Method contain distinctive implementation characteristics, allowing models to identify them more effectively. However, some patterns share similar object creation mechanisms, which may increase classification difficulty.

The use of contextual code representations helps reduce these challenges by capturing semantic relationships beyond simple syntax matching.

5.10 Explainability Analysis

Although high prediction performance is important, understanding the reason behind model decisions is also necessary for reliable AI-assisted software engineering.

To improve transparency, SHAP-based explainability analysis was performed to identify source code features contributing to design pattern predictions.

Figure 5.6 presents the explainability analysis results.

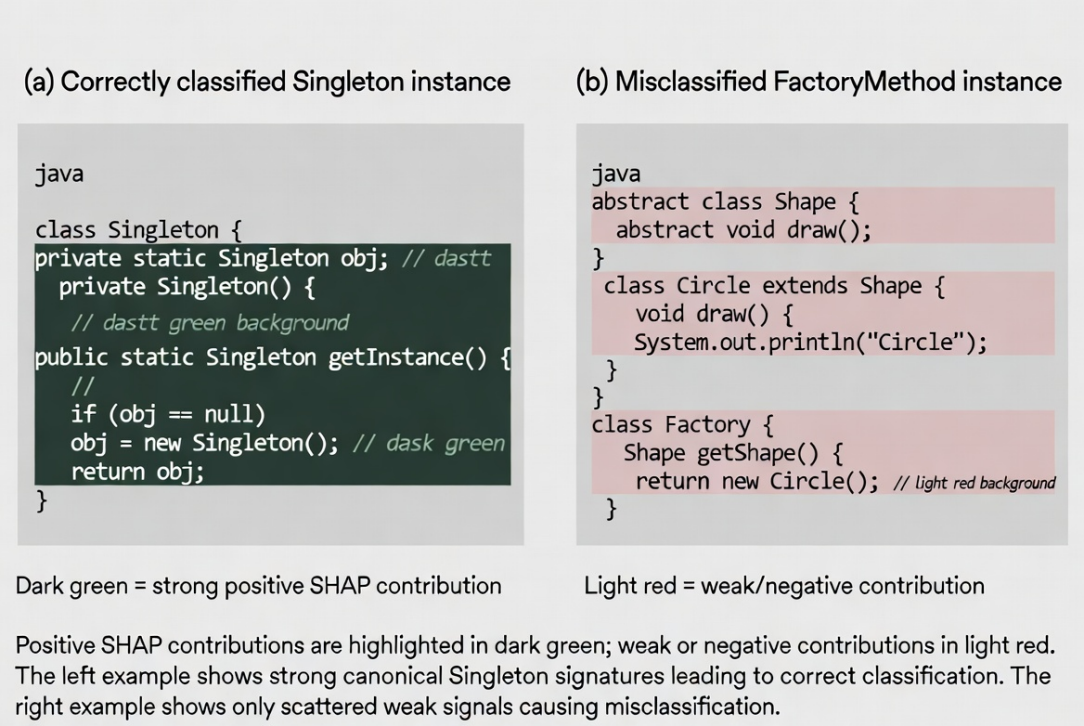


Figure 5.6: SHAP-based explainability analysis of design pattern recognition using source code features.

The analysis shows that specific code structures significantly influence the classification decisions. For example, object creation methods, inheritance relationships, and class structures provide important information for recognizing design patterns.

The explainability analysis improves confidence in the proposed framework by providing insights into how the model reaches its predictions.

5.11 Summary

This chapter presented an LLM-based approach for automated software design pattern recognition. Multiple pre-trained language models were evaluated for generating code representations, and their effectiveness was measured using a k-NN classifier.

The experimental results demonstrate that GraphCodeBERT-based embeddings provide strong performance for design pattern recognition. The combination of LLM-based code understanding and machine learning classification provides an effective solution for automated software design analysis.

Chapter 6

Results and Discussion

6.1 Introduction

This chapter presents a detailed analysis and discussion of the experimental results obtained from the two major research contributions of this thesis. The first contribution focuses on deep learning-based UML diagram recognition, while the second contribution investigates large language model (LLM)-based software design pattern recognition.

The purpose of this chapter is to analyze the achieved performance, effectiveness, and practical implications of the proposed approaches. The results are discussed from the perspective of automated software requirement understanding and software design analysis.

The overall relationship between the two research components is illustrated in Figure 6.1.

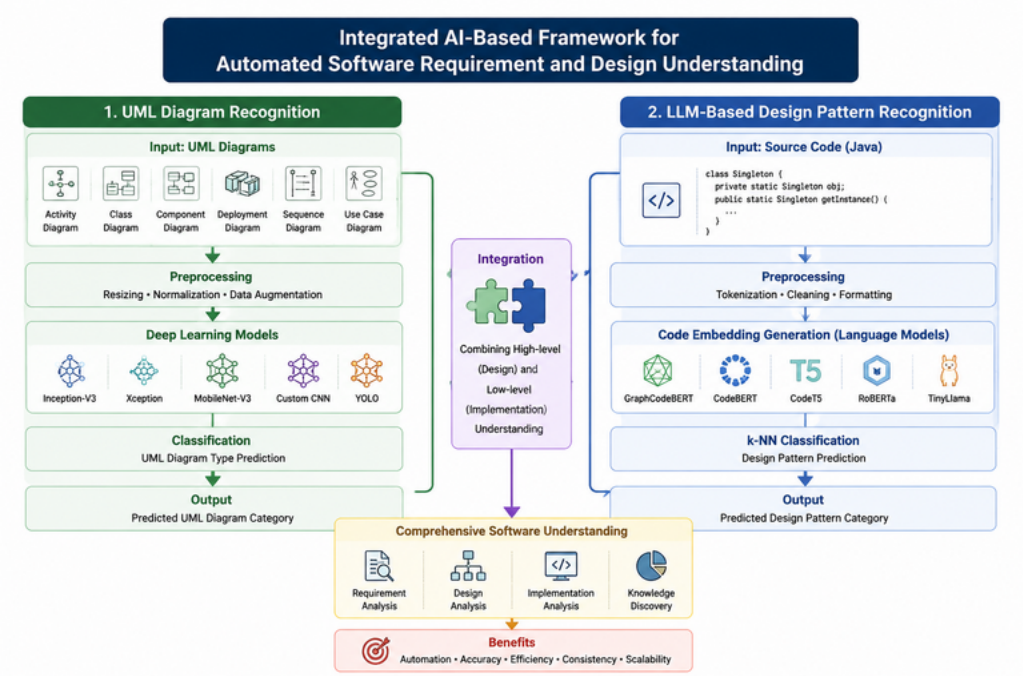


Figure 6.1: Integrated framework combining UML diagram recognition and LLM-based design pattern recognition.

6.2 Discussion of UML Diagram Recognition Results

The first research contribution investigated automated recognition of UML diagrams using deep learning-based computer vision techniques. Multiple architectures were evaluated to determine their effectiveness in recognizing different UML diagram categories.

The experiment considered six UML diagram types: Activity Diagram, Class Diagram, Component Diagram, Deployment Diagram, Sequence Diagram, and Use Case Diagram. The balanced dataset containing 6,666 UML images enabled a fair comparison among different models.

The experimental results demonstrate that deep learning models can effectively learn visual representations from UML diagrams. The evaluated architectures were able to capture important characteristics such as shapes, relationships, layouts, and structural patterns.

6.2.1 Performance Comparison of UML Recognition Models

The performance comparison of different deep learning architectures is presented in Table 6.1. The table summarizes the architecture type, main advantage, and overall recognition performance of the evaluated models.

Table 6.1: Performance Summary of UML Diagram Recognition Models

Model	**Architecture Type**	**Main Advantage**	**Performance**
Inception-V3	Transfer Learning	Deep feature extraction capability	50%
Xception	Transfer Learning	Efficient depthwise separable convolution	49%
MobileNet-V3	Lightweight CNN	Low computational cost and efficient feature learning	96%
Custom CNN	Task-specific CNN	UML-specific feature extraction	76%
YOLO	Object Detection	Spatial feature understanding and region-based recognition	99%

Among all evaluated approaches, YOLO achieved the highest recognition performance with an accuracy of 99%. The strong performance of YOLO indicates that object detection-based learning can effectively capture spatial relationships and structural patterns within UML diagrams.

Unlike conventional image classification approaches that analyze an entire image as a single feature representation, YOLO considers important image regions and learns discriminative visual patterns. This capability is particularly useful for UML diagrams because relationships, connections, and element positioning contain important semantic information.

6.2.2 Comparison with Existing UML Recognition Studies

Existing UML recognition approaches mainly focus on specific diagram types or limited datasets. Many previous studies concentrate on individual UML categories such as class diagrams or element-level recognition.

Table 6.2 presents a comparison between existing studies and the proposed approach.

Table 6.2: Comparison with Existing UML Recognition Studies

Study	**Approach**	**Limitation**
Gosala et al. [17]	CNN-based UML classification	Focused mainly on class diagram recognition
Shcherban et al. [18]	Deep learning-based classification	Limited UML diagram categories
Chen et al. [20]	Semantic UML element extraction	Focused on individual UML components
Proposed Work	Multi-class deep learning framework	Comprehensive recognition of six UML diagram categories

The proposed framework improves upon existing studies by considering multiple UML diagram categories and evaluating several advanced deep learning architectures within a unified recognition framework. The evaluation of different models provides a comprehensive understanding of suitable AI approaches for automated UML diagram analysis.

6.3 Discussion of LLM-Based Design Pattern Recognition Results

The second research contribution investigated the capability of large language models for automated software design pattern recognition. Unlike traditional rule-based approaches, the proposed method utilizes pre-trained code language models to generate semantic representations from source code.

Five pre-trained language models were evaluated in this study:

- GraphCodeBERT
- CodeBERT
- CodeT5
- RoBERTa
- TinyLlama

The generated code embeddings were classified using a k-Nearest Neighbor (k-NN) classifier to identify six software categories, including five design patterns and one non-design-pattern class.

The experimental results demonstrate that LLM-based code representations can effectively capture semantic and structural information from source code. These learned representations provide meaningful features for automated software design pattern recognition.

6.3.1 Performance Analysis of LLM-Based Recognition

The performance comparison of different language model-based embeddings with the k-NN classifier is presented in Table 6.3.

Table 6.3: Performance Summary of LLM-Based Design Pattern Recognition

Embedding Model	**F1-score**
CodeBERT + k-NN	0.91
CodeT5 + k-NN	0.95
RoBERTa + k-NN	0.94
TinyLlama + k-NN	0.58
GraphCodeBERT + k-NN	0.969

Among all evaluated approaches, GraphCodeBERT combined with the k-NN classifier achieved the highest recognition performance with an F1-score of 0.969. This result indicates that structural-aware code representation plays an important role in identifying software design patterns.

The superior performance of GraphCodeBERT can be attributed to its ability to incorporate data flow information along with token-level representations. Software design patterns depend heavily on relationships among classes, objects, methods, and program structures. Therefore, models capable of capturing these dependencies can provide better recognition capability.

6.4 Integrated Analysis of Software Understanding

The two research contributions analyze different levels of software knowledge. The UML recognition component focuses on high-level software representation, where requirements and system designs are expressed visually. In contrast, the LLM-based component analyzes implementation-level information from source code.

Together, these approaches provide a comprehensive perspective of software understanding by connecting visual software artifacts with implementation details.

Table 6.4: Integrated Comparison of Research Contributions

Aspect	**UML Recognition**	**Design Pattern Recognition**
Input	UML Images	Source Code
Representation	Visual Features	Code Embeddings
AI Method	Deep Learning	LLM + Machine Learning
Classifier	CNN / YOLO	k-NN
Purpose	Requirement Understanding	Design Understanding

6.5 Practical Implications

The proposed AI-based software understanding framework provides several practical benefits for software engineering activities. By combining visual understanding of UML diagrams with source code analysis, the framework can support developers and software engineers in different software development tasks.

The major practical implications include:

- Automated analysis of software documentation and UML artifacts.
- Support for reverse engineering of existing software systems.
- Assistance in understanding legacy software projects.
- Reduction of manual effort required for software analysis.
- Intelligent educational tools for learning software engineering concepts.

The results demonstrate the potential of artificial intelligence-assisted software engineering systems for improving software comprehension and developer productivity. The proposed approaches can serve as supportive tools for analyzing complex software artifacts efficiently.

6.6 Limitations and Future Improvements

Although the proposed approaches achieved promising performance, several limitations remain that provide opportunities for future improvement.

For UML diagram recognition, the dataset mainly contains predefined UML diagram categories collected for experimental evaluation. Future research can investigate larger industrial-scale datasets containing diverse diagram styles, complex software architectures, and real-world project artifacts.

For design pattern recognition, the experiments were conducted using Java source code implementations from the P-MARt dataset. Future studies can extend the framework to multiple programming languages, including Python, C++, and C#, to evaluate cross-language design pattern recognition capability.

Furthermore, although LLM-based approaches provide strong semantic understanding of source code, their computational requirements remain high. Future research can explore lightweight language models, efficient embedding generation methods, and optimized deployment strategies for practical applications.

Another possible research direction is integrating both approaches into a unified software intelligence system. Such a system could connect UML-level design understanding with source-code-level analysis to provide more comprehensive software engineering assistance.

6.7 Summary

This chapter presented the experimental results, comparative analysis, and practical implications of the proposed AI-based software understanding framework.

The UML recognition experiments demonstrated that deep learning approaches can effectively classify UML diagram images. Among the evaluated models, YOLO achieved the highest recognition accuracy, showing strong capability in learning spatial and structural patterns from UML diagrams.

The design pattern recognition experiments demonstrated that LLM-based code embeddings can effectively capture software design knowledge. The combination of GraphCodeBERT representations with the k-NN classifier achieved the best performance, highlighting the importance of structural code understanding.

Overall, the combination of deep learning-based visual recognition and LLM-based source code analysis provides a comprehensive approach for automated software requirement and design understanding. The proposed framework demonstrates the potential of artificial intelligence techniques in improving software analysis, maintenance, and development processes.

Chapter 7

Conclusion and Future Work

7.1 Introduction

This chapter concludes the research presented in this thesis on automated software requirement and design understanding using artificial intelligence techniques. The research investigated two complementary approaches: deep learning-based UML diagram recognition and large language model (LLM)-based software design pattern recognition.

The primary objective of this thesis was to develop intelligent approaches that reduce manual effort in software analysis by automatically understanding both visual software artifacts and source code implementations.

The proposed framework combines computer vision techniques and advanced code representation learning methods to provide a comprehensive perspective of software system understanding.

7.2 Summary of Research Contributions

The major contributions of this thesis are summarized as follows:

1. A deep learning-based UML diagram recognition framework was developed for automatically identifying multiple UML diagram categories from visual images.

2. A comprehensive evaluation of different deep learning architectures, including Inception-V3, Xception, MobileNet-V3, Custom CNN, and YOLO, was conducted for automated UML recognition.

3. A balanced UML image dataset containing six major UML diagram categories was prepared and utilized for evaluating deep learning-based recognition models.

4. An LLM-based design pattern recognition framework was developed by combining pre-trained code language models with machine learning-based classification.

5. Multiple code representation models, including GraphCodeBERT, CodeBERT, CodeT5, RoBERTa, and TinyLlama, were investigated for generating meaningful source code embeddings.

6. A k-Nearest Neighbor (k-NN)-based classification approach was applied for identifying software design patterns from generated code representations.

7. Explainable AI analysis was performed to understand important source code characteristics contributing to design pattern prediction decisions.

7.3 Key Findings

The experimental results demonstrate that artificial intelligence techniques can effectively support automated software understanding tasks.

For UML diagram recognition, deep learning models successfully learned visual features from software diagrams and achieved strong classification performance. Among the evaluated approaches, YOLO achieved the highest recognition accuracy, demonstrating the effectiveness of spatial feature learning for UML diagram analysis.

For software design pattern recognition, the results demonstrate that LLM-based code representations can capture meaningful semantic and structural information from source code. GraphCodeBERT combined with k-NN achieved the best performance with an F1-score of 0.969, indicating that structural-aware code representation improves automated design pattern identification.

The combination of both approaches provides a broader understanding of software systems by connecting requirement-level visual representations with implementation-level source code knowledge.

7.4 Research Significance

This research contributes to the field of AI-assisted software engineering by demonstrating the capability of modern artificial intelligence techniques for automating important software analysis activities.

The UML recognition framework can assist developers and software engineers in automatically understanding software documentation and reducing the effort required for analyzing large collections of UML diagrams.

The design pattern recognition framework can support software maintenance, reverse engineering, and architectural understanding by automatically identifying design knowledge embedded within source code.

Overall, this thesis demonstrates the potential of integrating deep learning and large language models for developing intelligent software engineering applications.

7.5 Limitations

Although the proposed approaches achieved promising results, several limitations remain.

The UML recognition study was evaluated using a specific dataset containing predefined UML diagram categories. Future evaluation using larger industrial datasets containing diverse diagram styles and complex software structures would provide stronger validation.

The design pattern recognition experiment was conducted using Java source code samples from a benchmark dataset. Extending the framework to additional programming languages would improve its general applicability.

Furthermore, LLM-based approaches require considerable computational resources. Developing lightweight models and efficient deployment strategies is necessary for practical real-time software engineering applications.

7.6 Future Work

Several future research directions can be explored based on the findings of this thesis.

- Developing larger and more diverse UML datasets containing real-world industrial software diagrams.
- Integrating vision-language models for advanced UML understanding and automatic diagram-to-code generation.
- Extending design pattern recognition approaches to multiple programming languages such as Python, C++, and C#.
- Developing intelligent IDE-based tools for automatic software design analysis and developer assistance.
- Investigating advanced explainable AI techniques to improve transparency and trust in AI-based software engineering systems.
- Combining UML analysis and source code analysis into a unified software intelligence platform.

7.7 Final Remarks

In conclusion, this thesis presents an integrated artificial intelligence framework for automated software requirement and design understanding.

The research demonstrates that deep learning techniques can effectively analyze visual UML artifacts, while large language models provide powerful representations for understanding source code structures and software design patterns.

By combining these complementary approaches, the proposed framework provides a foundation for future intelligent software engineering systems capable of automatically analyzing, understanding, and assisting software development activities.

Bibliography

[1] A. Vaswani et al., "Attention Is All You Need," *Advances in Neural Information Processing Systems*, vol. 30, 2017.

[2] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, pp. 436–444, 2015.

[3] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," *Advances in Neural Information Processing Systems*, vol. 25, 2012.

[4] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 2818–2826, 2016.

[5] F. Chollet, "Xception: Deep learning with depthwise separable convolutions," *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 1251–1258, 2017.

[6] A. Howard et al., "Searching for MobileNetV3," *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, pp. 1314–1324, 2019.

[7] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You Only Look Once: Unified, Real-Time Object Detection," *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 779–788, 2016.

[8] C. Shorten and T. M. Khoshgoftaar, "A survey on image data augmentation for deep learning," *Journal of Big Data*, vol. 6, article no. 60, 2019.

[9] M. Tan and Q. Le, "EfficientNet: Rethinking model scaling for convolutional neural networks," *Proceedings of the International Conference on Machine Learning (ICML)*, pp. 6105–6114, 2019.

[10] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *International Conference on Learning Representations (ICLR)*, 2015.

[11] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 770–778, 2016.

[12] J. Deng, W. Dong, R. Socher, L. Li, K. Li, and L. Fei-Fei, "ImageNet: A large-scale hierarchical image database," *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 248–255, 2009.

[13] L. van der Maaten and G. Hinton, "Visualizing data using t-SNE," *Journal of Machine Learning Research*, vol. 9, pp. 2579–2605, 2008.

[14] A. Dosovitskiy et al., "An image is worth 16x16 words: Transformers for image recognition at scale," *International Conference on Learning Representations (ICLR)*, 2021.

[15] A. Khan, A. Sohail, U. Zahoora, and A. S. Qureshi, "A survey of the recent architectures of deep convolutional neural networks," *Artificial Intelligence Review*, vol. 53, pp. 5455–5516, 2020.

[16] J. Rumbaugh, I. Jacobson, and G. Booch, "The Unified Modeling Language Reference Manual," Addison-Wesley, 2004.

[17] B. Gosala, S. R. Chowdhuri, J. Singh, M. Gupta, and A. Mishra, "Automatic classification of UML class diagrams using deep learning technique: Convolutional neural network," *Applied Sciences*, vol. 11, no. 9, article no. 4267, 2021.

[18] S. Shcherban, P. Liang, Z. Li, and C. Yang, "Multiclass classification of UML diagrams using deep learning," *Proceedings of the International Conference on Software Engineering and Knowledge Engineering (SEKE)*, pp. 57–62, 2021.

[19] G. Tavares, T. E. Colanzi, and Y. M. G. Costa, "Dataset development for classification of UML diagrams to support software engineering education," *Zenodo Dataset*, 2021.

[20] F. Chen, L. Zhang, X. Lian, and N. Niu, "Automatically recognizing semantic elements from UML class diagram images," *Journal of Systems and Software*, vol. 193, article no. 111431, 2022.

[21] J. Torcal, V. Moreno, J. Llorens, and A. Granados, "Creating and validating a ground truth dataset of Unified Modeling Language diagrams using deep learning techniques," *Applied Sciences*, vol. 14, no. 23, article no. 10873, 2024.

[22] Z. Babaalla, H. Abdelmalek, A. Jakimi, and M. Oualla, "Extraction of UML class diagrams using deep learning: Comparative study and critical analysis," *Procedia Computer Science*, vol. 236, pp. 452–459, 2024.

[23] A. Conrardy and J. Cabot, "From image to UML: First results of image based UML diagram generation using LLMs," *arXiv preprint arXiv:2404.11376*, 2024.

[24] A. Bates et al., "Generating UML models from diagram images using multimodal large language models," *Machine Learning with Applications*, 2025.

[25] T. Buchmann and J. Fraas, "AI-based recognition of sketched class diagrams," *Proceedings of MODELSWARD*, pp. 227–234, 2024.

[26] L. Wang, T. Song, H. N. Song, and S. Zhang, "Research on design pattern detection method based on UML model with extended image information and deep learning," *Applied Sciences*, vol. 12, no. 17, article no. 8718, 2022.

[27] V. Polischuk, "Automatic recognition of UML diagrams in images: Approaches, trends, and challenges," *Technologies and Engineering*, vol. 1, no. 26, pp. 23–35, 2025.

[28] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, "Design Patterns: Elements of Reusable Object-Oriented Software," Addison-Wesley, 1994.

[29] R. Ferenc, A. Beszedes, L. Fulop, and J. Lele, "Design pattern mining enhanced by machine learning," *Proceedings of the IEEE International Conference on Software Maintenance (ICSM)*, pp. 295–304, 2005.

[30] S. Uchiyama, H. Washizaki, Y. Fukazawa, and A. Kubo, "Design pattern detection using software metrics and machine learning," *Proceedings of the International Workshop on Model-Driven Software Migration*, pp. 38, 2011.

[31] S. Alhusain, S. Coupland, R. John, and M. Kavanagh, "Towards machine learning based design pattern recognition," *Proceedings of the UK Workshop on Computational Intelligence*, pp. 244–251, 2013.

[32] M. G. Al-Obeidallah, M. Petridis, and S. Kapetanakis, "A survey on design pattern detection approaches," *International Journal of Software Engineering*, vol. 7, no. 3, pp. 41–59, 2016.

[33] R. Xiong and B. Li, "Accurate design pattern detection based on idiomatic implementation matching in Java language context," *Proceedings of the IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 163–174, 2019.

[34] R. Oberhauser, "A machine learning approach towards automatic software design pattern recognition across multiple programming languages," *Proceedings of the International Conference on Software Engineering Advances*, pp. 27–32, 2020.

[35] M. Kouli and A. Rasoolzadegan, "A feature-based method for detecting design patterns in source code," *Symmetry*, vol. 14, no. 7, article no. 1491, 2022.

[36] R. Moreira, E. Fernandes, and E. Figueiredo, "A comparative study of design pattern detection tools," *Proceedings of the Conference on Pattern Languages of Programs*, pp. 1–16, 2022.

[37] A. Nacef, S. Bahroun, A. Khalfallah, and S. B. Ahmed, "Features and supervised machine learning based method for singleton design pattern variants detection," *International Conference on Evaluation of Novel Approaches to Software Engineering*, pp. 226–237, 2023.

[38] R. Oberhauser and S. Moser, "Design pattern detection in code: A hybrid approach utilizing Bayesian networks, machine learning with graph embeddings, and micropattern rules," *Proceedings of the International Conference on Software Engineering Advances*, pp. 122–129, 2023.

[39] S. Komolov, G. Dlamini, S. Megha, and M. Mazzara, "Towards predicting architectural design patterns: A machine learning approach," *Computers*, vol. 11, no. 10, article no. 151, 2022.

[40] T. Wang, Y. Zhang, X. Gong, and B. Li, "Recover and optimize software architecture based on source code and directory hierarchies," *International Conference on Software Engineering and Knowledge Engineering*, pp. 469–599, 2019.

[41] J. Devlin, M. W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," *Proceedings of NAACL-HLT*, pp. 4171–4186, 2019.

[42] Z. Feng et al., "CodeBERT: A pre-trained model for programming and natural languages," *Proceedings of EMNLP*, pp. 1536–1547, 2020.

[43] D. Guo et al., "GraphCodeBERT: Pre-training code representations with data flow," *International Conference on Learning Representations (ICLR)*, 2021.

[44] Y. Wang et al., "CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," *Proceedings of EMNLP*, pp. 8696–8708, 2021.

[45] W. U. Ahmad, S. Chakraborty, B. Ray, and K. Chang, "Unified pre-training for program understanding and generation," *Proceedings of NAACL-HLT*, pp. 2655–2668, 2021.

[46] T. Brown et al., "Language models are few-shot learners," *Advances in Neural Information Processing Systems*, vol. 33, pp. 1877–1901, 2020.

[47] X. Hou et al., "Large language models for software engineering: A survey," *ACM Transactions on Software Engineering and Methodology*, 2023.

[48] H. Li et al., "Large language models for software engineering: A systematic literature review," *Proceedings of the International Conference on Software Engineering (ICSE)*, 2023.

[49] X. Zhou et al., "DevGPT: Studying developer-ChatGPT conversations," *Proceedings of the IEEE/ACM International Conference on Software Engineering*, 2023.

[50] B. Rozière et al., "Code Llama: Open foundation models for code," *arXiv preprint arXiv:2308.12950*, 2023.

[51] H. Touvron et al., "LLaMA: Open and efficient foundation language models," *arXiv preprint arXiv:2302.13971*, 2023.

[52] A. Q. Jiang et al., "Mistral 7B," *arXiv preprint arXiv:2310.06825*, 2023.

[53] R. Bommasani et al., "On the opportunities and risks of foundation models," *arXiv preprint arXiv:2108.07258*, 2021.

[54] L. Schmid et al., "Software architecture meets large language models: A systematic literature review," *arXiv preprint arXiv:2505.16697*, 2025.

[55] S. M. Lundberg and S. I. Lee, "A unified approach to interpreting model predictions," *Advances in Neural Information Processing Systems*, vol. 30, 2017.

[56] M. T. Ribeiro, S. Singh, and C. Guestrin, "Why should I trust you?: Explaining the predictions of any classifier," *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 1135–1144, 2016.

[57] A. B. Arrieta et al., "Explainable Artificial Intelligence (XAI): Concepts, taxonomies, opportunities and challenges toward responsible AI," *Information Fusion*, vol. 58, pp. 82–115, 2020.

[58] Y. Zhang et al., "Explainable artificial intelligence for software engineering applications," *IEEE Access*, vol. 10, 2022.

[59] H. Le et al., "Artificial intelligence for software engineering: Current trends and future directions," *ACM Computing Surveys*, 2024.

[60] E. Maldonado et al., "Machine learning approaches for software maintenance and evolution," *Journal of Systems and Software*, 2023.

[61] L. Zhang et al., "Deep learning-based program understanding: A survey," *ACM Computing Surveys*, 2023.

[62] R. Li et al., "AI agents for software engineering: A survey," *arXiv preprint arXiv:2401.05000*, 2024.

[63] J. Ramos et al., "Artificial intelligence techniques for intelligent software development: A comprehensive review," *Information and Software Technology*, 2024.

[64] J. Wang, X. Li, and Y. Zhang, "Multimodal large language models for software engineering," *IEEE Software*, 2024.

[65] Y. Xu et al., "Large language model based software engineering automation," *IEEE Transactions on Software Engineering*, 2024.

[66] Z. Wang et al., "Reasoning about code with large language models," *IEEE Transactions on Software Engineering*, 2025.

[67] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 4700–4708, 2017.

[68] R. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 580–587, 2014.

[69] S. Ren, K. He, R. Girshick, and J. Sun, "Faster R-CNN: Towards real-time object detection with region proposal networks," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 39, no. 6, pp. 1137–1149, 2015.

[70] Z. W. Bhuiyan, S. A. R. Haider, A. Haque, M. R. Uddin, and M. Hasan, "IoT based meat freshness classification using deep learning," *IEEE Access*, vol. 12, pp. 196047–196069, 2024.

[71] Z. W. Bhuiyan, S. A. R. Haider, A. Haque, M. Hasan, and M. R. Uddin, "Meat freshness classifier with machine and AI," *2023 IEEE Region 10 Symposium (TENSYMP)*, pp. 1–5, 2023.

[72] Z. W. Bhuiyan, M. H. Limon, M. R. Rana, S. R. Akthar, M. T. Habib, and M. Hasan, "A comparative study of deep learning models for plant leaf and disease recognition in jackfruit, rice, and rose," *International Conference on Advances in Communication Technology and Computer Engineering*, pp. 460–471, Springer, 2024.

[73] Z. W. Bhuiyan, M. R. Rana, M. H. Limon, S. R. Akthar, M. T. Habib, and M. Hasan, "A comparative analysis of deep learning and machine learning models for fruit and fruit disease recognition: Applications to citrus, guava, and mango," *International Conference on Advances in Communication Technology and Computer Engineering*, pp. 398–409, Springer, 2024.

[74] Z. W. Bhuiyan, M. H. Kabir, P. Saha, M. M. Alam, and M. Hasan, "Rural Bangladesh before COVID: Baseline trends and district forecasts from the Life in the Field program," *World Conference on Information Systems for Business Management*, pp. 450–460, Springer, 2025.

[75] M. R. Rana, Z. W. Bhuiyan, M. H. Limon, A. Rahman, M. Hasan, and M. T. Habib, "Review of modern chatbot technologies: Performance, capabilities and applications," *2025 MIPRO 48th ICT and Electronics Convention*, pp. 81–86, IEEE, 2025.

[76] Z. W. Bhuiyan, M. A. Rahman, S. Sultana, and M. T. Habib, "A hybrid machine learning approach for accurate weather forecasting in Dhaka City: Insights for urban planning and disaster management," *2026 5th International Conference on Sentiment Analysis and Deep Learning (ICSADL)*, pp. 1–6, IEEE, 2026.

[77] D. Kundu, S. R. Bhuiyan, Z. W. Bhuiyan, S. Banik, and M. T. Habib, "A machine vision framework for Bangladeshi geographical indication dessert recognition: Comparative analysis with traditional machine learning and deep learning models," *2026 IEEE/ACIS 24th International Conference on Software Engineering Research, Management and Applications (SERA)*, pp. 339–344, IEEE, 2026.

[78] M. K. Sarker, S. Noor-A-Manik, Z. W. Bhuiyan, N. Nahar, M. I. Jabiullah, and M. T. Habib, "Exploring the implication of ML-based evaluation techniques to overcome the software development lifecycle selection challenges faced by project managers," *2026 International Conference on Artificial Intelligence for Sustainable Engineering and Innovation (AISEI)*, pp. 588–592, IEEE, 2026.

[79] Z. W. Bhuiyan, M. R. Rana, M. A. Rahman, M. S. Munir, M. I. Jabiullah, and M. T. Habib, "Comparative evaluation of hybrid machine learning models for predicting weather and air quality in Dhaka City: Insights for urban environmental management," *2026 International Conference on Artificial Intelligence for Sustainable Engineering and Innovation (AISEI)*, pp. 980–984, IEEE, 2026.

[80] Z. W. Bhuiyan, M. A. Rahman, S. Sultana, and M. T. Habib, "Deep learning based multi class recognition of medicinal plant leaf diseases," *2026 International Conference on Artificial Intelligence for Sustainable Engineering and Innovation (AISEI)*, pp. 762–766, IEEE, 2026.

[81] D. Kundu, S. R. Bhuiyan, M. T. H. Siam, Z. W. Bhuiyan, S. K. Dey, and M. T. Habib, "Deep learning-based recognition of rare Bangladeshi fruits using custom CNN and transfer learning models," *2026 International Conference on Artificial Intelligence for Sustainable Engineering and Innovation (AISEI)*, pp. 953–957, IEEE, 2026.

[82] M. I. Hosen, A. K. Alif, Z. Zaman, Z. W. Bhuiyan, N. Nahar, and M. Hasan, "AI-enhanced software testing: A human-in-the-loop approach for precision and efficiency," *2026 5th International Conference on Electrical, Computer & Telecommunication Engineering (ICECTE)*, pp. 1–6, IEEE, 2026.

[83] Z. W. Bhuiyan, Z. B. M. Atib, M. Hasan, A. Sultana, and M. T. Habib, "A comprehensive analysis of multiclass dermatology image recognition using deep and classical machine learning," *2025 28th International Conference on Computer and Information Technology (ICCIT)*, pp. 1774–1779, IEEE, 2025.

[84] Z. W. Bhuiyan, M. M. Rahman, Z. B. M. Atib, M. Hasan, and M. T. Habib, "A unified deep learning approach for lung cancer detection across CT and histopathological modalities," *2025 28th International Conference on Computer and Information Technology (ICCIT)*, pp. 4070–4075, IEEE, 2025.

[85] Z. W. Bhuiyan, Z. B. M. Atib, M. Z. Islam, M. T. Habib, and M. Hasan, "Unified training and evaluation for multi-crop leaf disease recognition with deep and traditional machine learning models," *2025 28th International Conference on Computer and Information Technology (ICCIT)*, pp. 4142–4147, IEEE, 2025.

[86] Z. W. Bhuiyan, S. Ghosh Nil, Z. B. M. Atib, M. Z. Islam, M. Hasan, and M. T. Habib, "Deep learning for pest recognition in potato and tomato crops: A comparative study of convolutional neural networks," *2025 28th International Conference on Computer and Information Technology (ICCIT)*, pp. 2046–2051, IEEE, 2025.

[87] Z. W. Bhuiyan, A. Ahmed Eraj, P. Mondol, M. A. Rahman, S. Khan, and M. T. Habib, "Automated UML diagram recognition using deep learning: A comparative analysis of CNN and object detection approaches," *International Conference on Computational Vision and Bio Inspired Computing (ICCVBIC)*, 2026.

[88] Z. W. Bhuiyan, A. Ahmed Eraj, P. Mondol, M. A. Rahman, S. Khan, and M. T. Habib, "Large language model based design pattern recognition using code embeddings and machine learning," *International Conference on Computational Vision and Bio Inspired Computing (ICCVBIC)*, 2026.

[89] M. A. Rahman, A. A. Khan, M. M. Hasan, M. S. Rahman, and M. T. Habib, "Deep learning modeling for potato breed recognition," *IEEE Transactions on AgriFood Electronics*, vol. 2, no. 2, pp. 419–427, 2024.

[90] M. A. Rahman, M. A. Raihan, M. S. Uddin, M. Hasan, and M. T. Habib, "A comprehensive analysis of classic machine learning and deep learning modeling for breed recognition of bananas," *IEEE Access*, vol. 13, pp. 194562–194579, 2025.

[91] M. A. A. Khan, M. A. Rahman, M. L. Hossain, and M. T. Habib, "Machine vision based local hyacinth bean breed recognition using convolutional neural network," *2024 International Conference on Advances in Computing, Communication, Electrical, and Smart Systems (iCACCESS)*, pp. 1–6, 2024.

[92] U. Ghosh, S. Rokoni, M. A. Rahman, M. S. Rahman, and M. T. Habib, "CNN modeling for recognizing rice leaf diseases," *2024 15th International Conference on Computing Communication and Networking Technologies (ICCCNT)*, pp. 1–6, 2024.

[93] P. Das, M. A. Rahman, M. T. Habib, M. A. A. Khan, and M. M. Islam, "An in-depth analysis for machine-vision-based mango leaf diseases recognition," *2025 6th International Conference on Mobile Computing and Sustainable Informatics (ICMCSI)*, pp. 1571–1576, 2025.

[94] M. M. Islam, M. Akter, M. S. Uddin, M. A. Rahman, and M. T. Habib, "Machine vision–based insect recognition in agriculture: A comprehensive review," *Engineering Reports*, vol. 8, no. 4, p. e70729, 2026.

[95] M. M. Islam, M. A. Rahman, M. Akter, M. S. Uddin, and M. T. Habib, "Machine vision-based image recognition: A curated image dataset," *2026 Third International Conference on Innovations in Cybersecurity and Data Science (ICICDS)*, pp. 1383–1388, 2026.

[96] M. A. Rahman, Z. B. M. Atib, M. S. Hossen, S. H. Sourov, M. A. Alim, and M. T. Habib, "Machine vision based chili species recognition," *2026 6th International Conference on Image Processing and Capsule Networks (ICIPCN)*, pp. 449–454, 2026.

[97] M. M. Islam, M. Akter, M. S. Uddin, S. M. Anik, and M. A. Rahman, "Machine-vision-based paddy pest recognition using deep learning and classical machine learning methods," *The Journal of Engineering*, vol. 2026, no. 1, p. e70177, 2026.

Appendix A

Additional Experimental Details

A.1 UML Recognition Experimental Configuration

This appendix presents the experimental settings used for the UML diagram recognition study.

Table A.1: UML Recognition Configuration

Parameter	**Value**
Dataset Size	6,666 images
Classes	6
Input Size	224×224 pixels
Train/Validation/Test Split	80% / 10% / 10%
Learning Strategy	Supervised Learning
Optimizer	Adam
Metrics	Accuracy, Precision, Recall, F1-score

A.2 UML Dataset Categories

The dataset contains six UML diagram categories:

- Activity Diagram
- Class Diagram
- Component Diagram
- Deployment Diagram
- Sequence Diagram
- Use Case Diagram

A.3 Design Pattern Recognition Configuration

This section summarizes the experimental setup of the LLM-based design pattern recognition study.

Table A.2: Design Pattern Recognition Configuration

Parameter	**Value**
Dataset	P-MARt Repository
Language	Java
Programs	93
Categories	6
Representation	LLM Embeddings
Classifier	k-NN
Metrics	Precision, Recall, F1-score

A.4 Design Pattern Categories

The evaluated design pattern categories are:

- Abstract Factory
- Builder
- Factory Method
- Prototype
- Singleton
- Non-Implemented Design Pattern

Appendix B

Publication Statements

This appendix presents the publications resulting from the research contributions of this thesis. The publications are related to deep learning-based UML diagram recognition and LLM-based design pattern recognition.

B.1 Publication 1

Title	A Task-Specific Deep CNN Framework for UML Diagram Recognition in Requirements Engineering
Authors	Zarif Wasif Bhuiyan, Md. Ataur Rahman, Farzana Sadia, Md. Tarek Habib
Conference	7th International Conference on Computational Vision and Bio Inspired Computing (ICCVBIC 2026)
Research Area	Deep Learning-Based UML Diagram Recognition for Automated Software Requirement Understanding
Status	In-press
Contribution	This publication presents a deep CNN-based framework for automated UML diagram recognition. The study evaluates multiple deep learning architectures for UML diagram classification and demonstrates the potential of AI-based approaches for software requirement understanding.

B.2 Publication 2

Title	A Comprehensive Evaluation of LLMs for Design Pattern Recognition in Software Systems
Authors	Md. Ataur Rahman, Zarif Wasif Bhuiyan, Md. Tarek Habib
Conference	7th International Conference on Computational Vision and Bio Inspired Computing (ICCVBIC 2026)
Research Area	Large Language Model-Based Design Pattern Recognition for Automated Software Design Understanding
Status	In-press
Contribution	This publication investigates LLM-based approaches for automated design pattern recognition from source code. The study evaluates different code representation models and classification techniques for improving software design understanding.