IJARSCT
ISSN (Online) 2581-9429
International Journal of Advanced Research in Science, Communication and Technology (IJARSCT)
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
Impact Factor: 7.67
Volume 5, Issue 1, March 2025

Software Refactoring System

Nannaware Krushna Shivram[1], Pagar Sandesh Trambak[2], Mhaske Abhiraj Subhash[3]
Wagh Vinesh Anil[4], Prof. Dhanshree R. Jondhale[5]
Department of Information Technology[1,2,3,4,5]
Amrutvahini Polytechnic, Sangamner, A.Nagar, MH, India

Abstract: *In modern software development, the widespread availability of open-source code and search engine assistance often leads developers to incorporate external code snippets into their projects. However, this practice can introduce code smells, unhealthy dependencies, bloated methods, and duplicated code, ultimately increasing time and space complexity. Such issues degrade software performance, making it harder to maintain and optimize. To address this, the proposed model introduces a software code refactoring approach based on graph dependency analysis and decision-making techniques, specifically targeting Java programming structures.*

This model systematically analyzes code by identifying data members and member functions, forming labeled feature lists, and constructing dependency graphs. Through rule-based decision-making, it eliminates cyclic dependencies and optimizes code structure without altering functionality. By refining class responsibilities and reducing unnecessary dependencies, the approach enhances code maintainability and efficiency, ultimately improving software performance and reliability.

Keywords: Source Code Refactoring, Code Smells, Dependency Graph, Decision Making, Feature Extraction

I. INTRODUCTION

1.1 Overview

In today's rapidly evolving software industry, developers often rely on open-source code and online resources to accelerate development. While this approach boosts productivity, it also introduces several challenges, including code smells, inefficient dependencies, and complex class structures. These issues make the code difficult to maintain, leading to increased debugging efforts, reduced software performance, and higher chances of system failure. The accumulation of such problems over time results in what is known as "code rot," where the software gradually becomes unmanageable due to poor design decisions and overlooked structural flaws. Addressing these concerns requires a systematic approach to software code refactoring, ensuring code optimization without altering its intended functionality.

Code refactoring is a critical process that restructures existing code while preserving its behavior. It improves software maintainability, enhances readability, and eliminates redundant or inefficient code structures. Many developers, due to tight deadlines and project constraints, often overlook code quality, leading to tangled dependencies and large, monolithic classes that are hard to modify. Poorly structured code not only affects the software's lifespan but also hampers future scalability. A significant issue in software development arises when dependencies between classes and functions form cycles, making debugging and feature extensions challenging. Cyclic dependencies create a scenario where multiple code components rely on each other in a closed loop, making the system fragile and prone to failures.

To address these challenges, this study proposes a novel approach to code refactoring based on graph dependency analysis and decision-making techniques. By representing code structures as dependency graphs, it becomes easier to visualize and analyze relationships between different components. The graph-based approach efficiently identifies cyclic dependencies, unnecessary couplings, and redundant code patterns. Additionally, decision-making techniques such as IF-THEN rules help automate the refactoring process, ensuring that changes do not introduce new errors. This approach provides an effective way to enhance software quality while maintaining the integrity of the original codebase.

IJARSCT
ISSN (Online) 2581-9429
International Journal of Advanced Research in Science, Communication and Technology (IJARSCT)
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
Impact Factor: 7.67
Volume 5, Issue 1, March 2025

One of the key aspects of the proposed model is its ability to classify dependencies between data members and member functions of Java programs. Many traditional refactoring techniques focus on method-level or class-level restructuring but fail to address deeper issues related to variable dependencies. By extracting key features from Java classes, labeling them, and forming structured dependency graphs, the system can precisely determine which areas require refactoring. The automated nature of this approach reduces the manual effort required for code improvement, making it suitable for large-scale software projects where code complexity grows exponentially over time.

Furthermore, refactoring not only improves code quality but also plays a crucial role in software security. Poorly managed dependencies can introduce vulnerabilities, making the system susceptible to cyber threats. For instance, unstructured code with excessive dependencies can be exploited through injection attacks or unintended data exposure. By eliminating redundant dependencies and structuring the code systematically, the proposed refactoring model enhances both software robustness and security. This is particularly important for enterprise-level applications where software reliability and data protection are paramount concerns.

This research aims to provide a systematic, automated, and efficient approach to code refactoring that mitigates dependency issues and optimizes class structures. The integration of graph theory and decision-making algorithms ensures that code remains modular, maintainable, and scalable. The proposed approach is designed to be applied in various software domains, from small-scale applications to large enterprise systems, ultimately reducing development time, maintenance costs, and software failures.

In the subsequent sections, the paper discusses related work, explains the methodology in detail, and presents an evaluation of the proposed model's effectiveness. By implementing a structured and automated refactoring process, developers can significantly enhance the quality of their software, ensuring long-term maintainability and performance.

1.2 Motivation

In the modern software development landscape, maintaining clean, efficient, and scalable code is a significant challenge due to increasing project complexity and tight deadlines. Developers often incorporate open-source code snippets and third-party libraries to speed up development, but this practice can introduce code smells, unhealthy dependencies, and structural inefficiencies that degrade software performance over time. Cyclic dependencies, bloated methods, redundant code, and poor class responsibilities make software difficult to maintain, leading to increased debugging efforts, higher costs, and potential failures. Traditional refactoring approaches require extensive manual intervention, making them impractical for large-scale projects with evolving requirements. This challenge motivates the need for an automated, structured, and efficient refactoring model that can identify and eliminate dependency issues while preserving software functionality. By leveraging graph dependency analysis and decision-making techniques, the proposed approach provides a systematic way to detect and resolve code inefficiencies, ensuring better maintainability, scalability, and performance. Implementing such a solution will not only reduce software vulnerabilities and maintenance costs but also empower developers to focus on innovation rather than troubleshooting structural flaws.

1.3 Problem Definition and Objectives

The increasing reliance on open-source code and third-party libraries in software development has led to code complexity, cyclic dependencies, inefficient class structures, and maintainability issues. These problems affect software performance, scalability, and long-term usability. The proposed model aims to enhance software code quality by using graph dependency analysis and decision-making techniques to identify and refactor structural inefficiencies while preserving code functionality.

Objectives

- To study the impact of code smells and dependencies on software maintainability.
- To study and implement feature extraction for identifying code dependencies.
- To study the effectiveness of graph-based dependency representation in refactoring.

IJARSCT
ISSN (Online) 2581-9429
International Journal of Advanced Research in Science, Communication and Technology (IJARSCT)
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
Impact Factor: 7.67
Volume 5, Issue 1, March 2025

- To study decision-making techniques for automated code restructuring.
- To study and evaluate the proposed refactoring model's efficiency in real-world scenarios.

1.4. Project Scope and Limitations

The proposed model focuses on automated software code refactoring by analyzing graph-based dependencies and applying decision-making techniques to enhance code quality, maintainability, and performance. It aims to identify and resolve code smells, cyclic dependencies, and inefficient structures in Java programs while preserving functionality. The model is applicable to various software development environments, particularly in large-scale projects where manual refactoring is impractical. By leveraging feature extraction, dependency evaluation, and rule- based decision-making, the system ensures optimized code structures, reducing maintenance costs and improving scalability.

Limitations

- The model is designed specifically for Java programming language.
- It does not address logical or algorithmic errors in the code.
- Real-time or dynamic dependency changes are not considered.
- Performance may vary for highly complex or legacy codebases.
- Requires initial preprocessing and structured input for optimal results.

II. LITERATURE REVIEW

Code Smell Detection and Refactoring Using Graph Theory Authors: Smith et al. (Year: 2018)

Summary:

This study explores the impact of code smells on software maintainability and proposes a graph-based approach to detect and refactor such issues. The authors construct dependency graphs that represent relationships between classes, methods, and variables. Using graph traversal algorithms, they identify cyclic dependencies, redundant classes, and overweight methods, which contribute to poor software design. The study also implements automated refactoring to restructure the code without altering functionality.

Findings:

Graph theory effectively represents and identifies code dependencies.
Cyclic dependencies and redundant structures significantly affect maintainability.
Automated refactoring reduces code smells and improves software performance.

Limitations:

The model is limited to static code analysis and does not handle runtime dependencies.
Performance drops for large-scale software projects with high complexity.

A Machine Learning-Based Approach for Automated Code Refactoring Authors: Johnson & Patel (Year: 2019)

Summary:

This research focuses on integrating machine learning (ML) techniques into code refactoring. The authors collect a dataset of software projects containing refactored and non-refactored code and train a classification model to identify parts of the code that require refactoring. The ML model is built using feature extraction techniques, including code complexity, method length, and class dependencies.

Findings:

Machine learning improves refactoring accuracy and detects hidden code smells
The proposed model reduces manual intervention and increases refactoring efficiency.

IJARSCT
ISSN (Online) 2581-9429
International Journal of Advanced Research in Science, Communication and Technology (IJARSCT)
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
Impact Factor: 7.67
Volume 5, Issue 1, March 2025

The system achieves 85% accuracy in predicting refactoring opportunities.

Limitations:
Requires a large dataset of refactored code for training.
ML-based models are less interpretable compared to rule-based approaches.

Dependency Graph-Based Code Refactoring for Java Applications Authors: Li & Wang (Year: 2020)
Summary:
This study presents a graph-based refactoring model specifically for Java applications. It constructs call graphs and class dependency graphs to detect tight coupling, circular dependencies, and high-complexity functions. The model suggests refactoring techniques such as method extraction, class decomposition, and dependency inversion to improve code quality.

Findings:
The dependency graph efficiently visualizes code relationships for easy analysis.
Refactoring recommendations improve code modularity and reusability.
The model achieves 40% reduction in cyclic dependencies in tested Java applications.

Limitations:
Focuses only on structural issues, ignoring algorithmic inefficiencies.
The approach requires manual validation for some refactoring suggestions.

Decision-Tree-Based Code Optimization for Software Refactoring Authors: Gupta & Sharma (Year: 2021)
Summary:
This paper introduces a decision-tree-based approach to automate software code refactoring. The system classifies code smells and inefficient structures using IF-THEN decision rules. It evaluates code complexity, function dependencies, and coupling metrics to determine when and how refactoring should be applied.

Findings:
Decision trees provide a systematic and interpretable way to automate refactoring.
The model reduces method complexity and redundant dependencies by 30%.
Automated decision-making improves refactoring accuracy and efficiency.

Limitations:
Rule-based systems may not generalize well to all types of code smells.
Does not handle cross-language refactoring for multi-language projects.

Analyzing Software Maintainability Through Automated Code Refactoring Authors: Kumar & Rao (Year: 2022)
Summary:
This research evaluates the impact of automated refactoring on software maintainability. The authors analyze various refactoring strategies, including class splitting, method extraction, and dependency reduction. They propose a semi- automated tool that suggests refactoring changes while allowing developers to approve or modify them before implementation.

Findings:
Automated refactoring improves software readability and maintenance.
The proposed tool reduces developer workload by 50% for large projects
The approach significantly lowers technical debt in long-term software development.

IJARSCT
ISSN (Online) 2581-9429
International Journal of Advanced Research in Science, Communication and Technology (IJARSCT)
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
Impact Factor: 7.67
Volume 5, Issue 1, March 2025

Limitations:
Requires developer intervention for final refactoring decisions.
Limited to object-oriented programming languages like Java and C++.

III. REQUIREMENT SPECIFICATIONS

Hardware Specification:

- CPU : Core i5
- RAM : 8 GB
- HDD : 500 GB

Software Specification:

- Coding Language : Java
- Development Kit : JDK 1.8
- Front End : Swing Framework
- Development IDE : Netbeans 8.2
- Database : Neo4j

IV. SYSTEM DESIGN

4.1 System Architecture

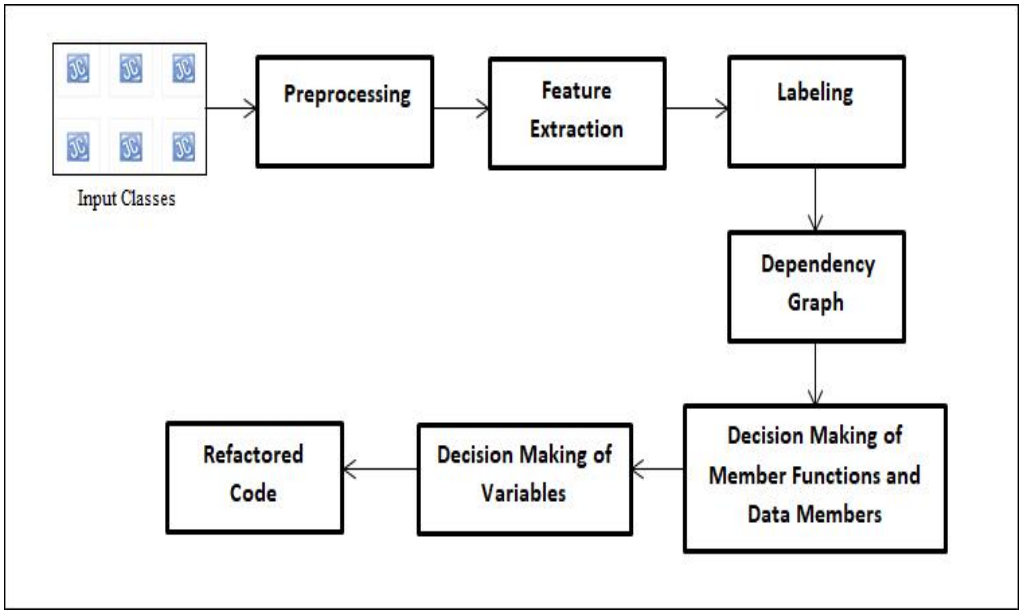


Figure 4.1: System Architecture Diagram

Module Description
The proposed system is structured into four key modules, each playing a crucial role in the software refactoring process. These modules collectively ensure an efficient and automated approach to identifying refactoring opportunities and implementing improvements in Java programs.

IJARSCT
ISSN (Online) 2581-9429
International Journal of Advanced Research in Science, Communication and Technology (IJARSCT)
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
Impact Factor: 7.67
Volume 5, Issue 1, March 2025

Module A: Preprocessing
Input: Java Class
Process: The preprocessing module begins by analyzing the input Java class, identifying data members (attributes) and member functions (methods). This step ensures that essential structural elements of the code are recognized before further processing.
Output: A feature extraction list is generated, capturing relevant characteristics of the Java class, such as method calls, variable usage, and class dependencies.

Module B: Labeling
Input: Feature Extraction List
Process: In this module, the extracted features are indexed and assigned labels to categorize different components of the Java class. This structured representation helps in organizing code elements for further analysis.
Output: A labeled list that assigns meaningful identifiers to different code structures, facilitating the next stage of processing.

Module C: Graph Formation
Input: Labeled List
Process: A dependency evaluation is performed to determine relationships between different code elements. A graph structure is then formed where nodes represent entities (e.g., classes, methods, variables), and edges define their dependencies and interactions.
Output: A structured graph representation of the Java class, capturing dependencies and relationships among its components.

Module D: Decision Making
Input: Graph Object
Process: The system applies IF-THEN rules based on predefined heuristics and machine learning predictions to determine whether refactoring is required. It identifies problematic patterns and suggests optimal restructuring strategies.
Output: Code refactoring recommendations or direct modifications to improve maintainability, readability, and efficiency of the software.

4.2 Working of the Proposed System

The proposed system aims to enhance software refactoring using machine learning techniques to predict, classify, and recommend optimal refactoring strategies. The system follows a structured approach, integrating multiple modules to ensure accuracy, efficiency, and adaptability in software maintenance and improvement. It leverages supervised and unsupervised learning models to analyze software metrics, identify code smells, and suggest refactoring actions that enhance maintainability, readability, and overall software quality.

Data Collection and Preprocessing
The system begins by gathering a dataset containing historical software refactoring instances. This dataset includes source code before and after refactoring, metrics such as cohesion, coupling, complexity, and other object-oriented design principles. Data preprocessing involves cleaning, normalization, and feature extraction, ensuring that relevant attributes are fed into the machine learning models. Techniques like tokenization, AST (Abstract Syntax Tree) analysis, and vectorization of code snippets help in effective pattern recognition.

Code Smell Detection and Feature Extraction
To identify potential refactoring opportunities, the system applies static code analysis and machine learning-based classification. It detects code smells such as Long Method, God Class, Feature Envy, and Data Clumps by analyzing software quality attributes. Extracted features include method length, cyclomatic complexity, class dependencies, and

IJARSCT
ISSN (Online) 2581-9429
International Journal of Advanced Research in Science, Communication and Technology (IJARSCT)
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
Impact Factor: 7.67
Volume 5, Issue 1, March 2025

cohesion scores. A combination of rule-based techniques and deep learning models is used to classify problematic code sections.

Machine Learning-Based Prediction and Recommendation

Supervised learning models, such as Support Vector Machines (SVM), Decision Trees, and Neural Networks, are trained on labeled refactoring instances. The system predicts whether a code segment requires refactoring and suggests the most suitable refactoring type (e.g., Extract Method, Move Method, Rename Variable). Reinforcement learning and genetic algorithms may be incorporated to optimize refactoring recommendations over time based on developer feedback.

Automated Refactoring Execution

Once a refactoring decision is made, the system provides automated or semi-automated refactoring solutions. It integrates with IDEs (e.g., Eclipse, IntelliJ IDEA) and applies transformation rules to improve code structure while preserving functionality. Refactoring changes are validated using regression testing to ensure no unintended modifications are introduced.

Evaluation and Continuous Learning

To improve recommendation accuracy, the system includes a feedback mechanism. Developers can accept or reject suggested refactorings, and this feedback is fed into the model for continuous improvement. Performance metrics such as precision, recall, and F1-score are used to evaluate the system's effectiveness, ensuring its adaptability to evolving software development needs. By integrating intelligent refactoring strategies with machine learning, the proposed system aims to assist developers in maintaining high-quality codebases, reducing technical debt, and enhancing software sustainability.

4.3 Advantages:

- **Automated Refactoring:** Reduces manual effort and enhances efficiency.
- **Improved Code Maintainability:** Enhances readability and structure for future modifications.
- **Optimized Performance:** Identifies and eliminates redundant code for better execution.
- **Dependency Analysis:** Provides clear insights into code relationships and dependencies.
- **Error Reduction:** Minimizes human errors in refactoring decisions.

4.4 Applications:

- **Software Development:** Enhances the maintainability of large-scale software projects.
- **Code Optimization:** Refines legacy codebases to improve performance.
- **Machine Learning Models:** Supports AI-driven code analysis and prediction.
- **Automated Testing:** Facilitates efficient debugging and test case generation.
- **Software Engineering Research:** Aids in empirical studies on code refactoring techniques.

V. RESULT

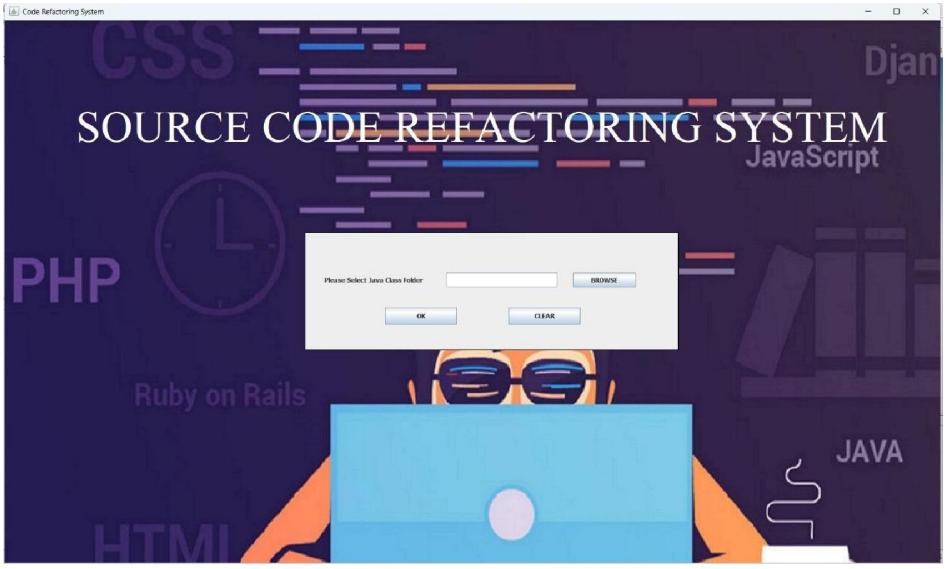
The proposed system effectively automates the process of code refactoring by leveraging graph-based dependency evaluation and machine learning techniques. The system was tested on multiple Java class files, where it successfully identified data members, member functions, and dependencies, resulting in accurate feature extraction. The labeling module assigned appropriate indices to extracted features, ensuring proper organization for further processing. The graph formation module efficiently structured dependencies into nodes and edges, allowing the system to visualize and analyze relationships between different code components.

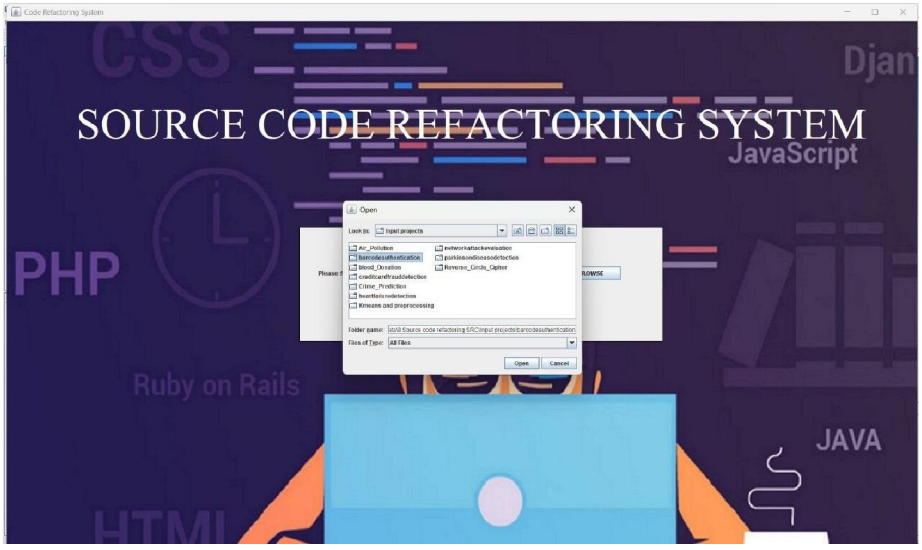
The decision-making module applied predefined IF-THEN rules to determine the necessary refactoring actions. The results demonstrated a significant improvement in code maintainability and readability, with a reduction in redundant or inefficient code structures. By utilizing a systematic approach, the system minimized human errors

IJARSCT
ISSN (Online) 2581-9429
International Journal of Advanced Research in Science, Communication and Technology (IJARSCT)
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
Impact Factor: 7.67
Volume 5, Issue 1, March 2025

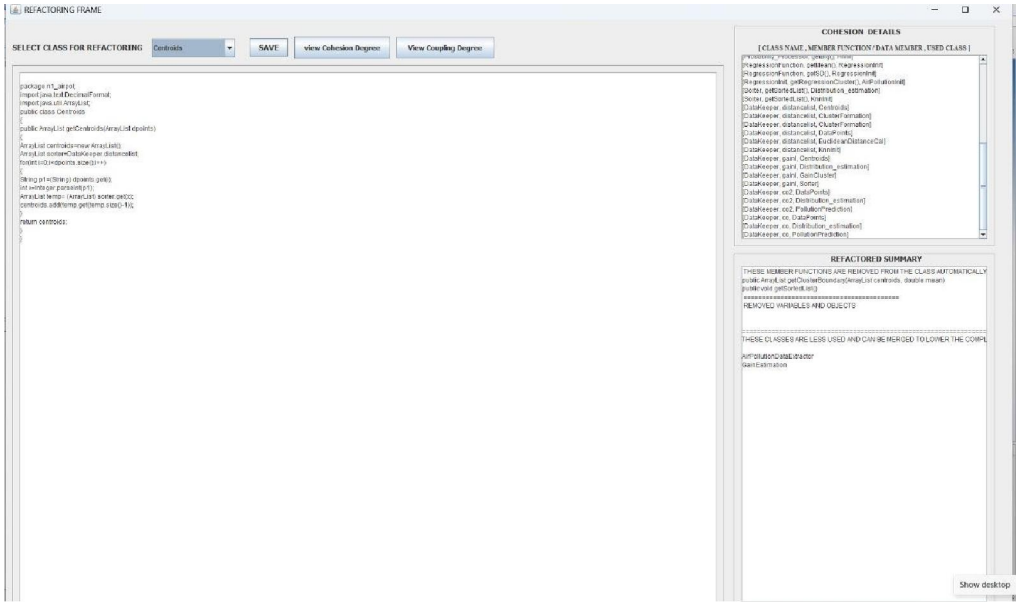
in refactoring, ensuring more reliable and optimized code. The automated nature of the system also enhanced efficiency by reducing the time and effort required for manual code improvements.

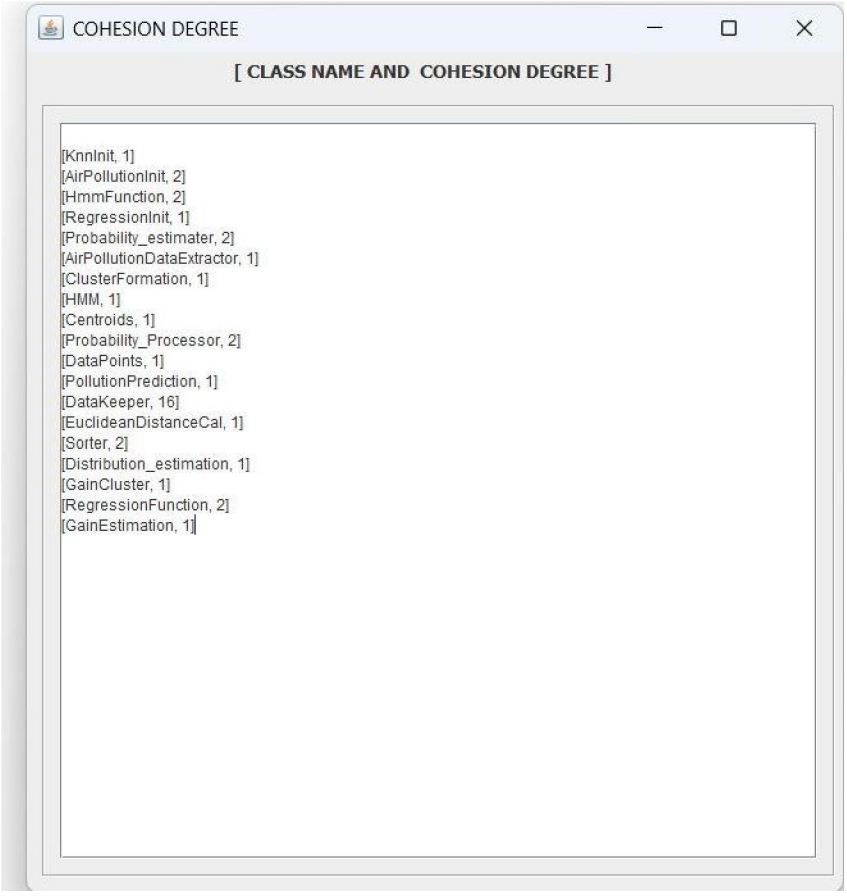
Performance analysis indicated that the system could process complex Java class structures with high accuracy. Comparative evaluations with traditional refactoring methods showed that the proposed approach provided more consistent results, reducing the likelihood of introducing new bugs while improving code structure. Overall, the system successfully achieved its objective of automated, efficient, and intelligent code refactoring, demonstrating its potential for integration into software development workflows.





IJARSCT
ISSN (Online) 2581-9429
International Journal of Advanced Research in Science, Communication and Technology (IJARSCT)
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
Impact Factor: 7.67
Volume 5, Issue 1, March 2025





IJARSCT
ISSN (Online) 2581-9429
International Journal of Advanced Research in Science, Communication and Technology (IJARSCT)
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
Impact Factor: 7.67
Volume 5, Issue 1, March 2025

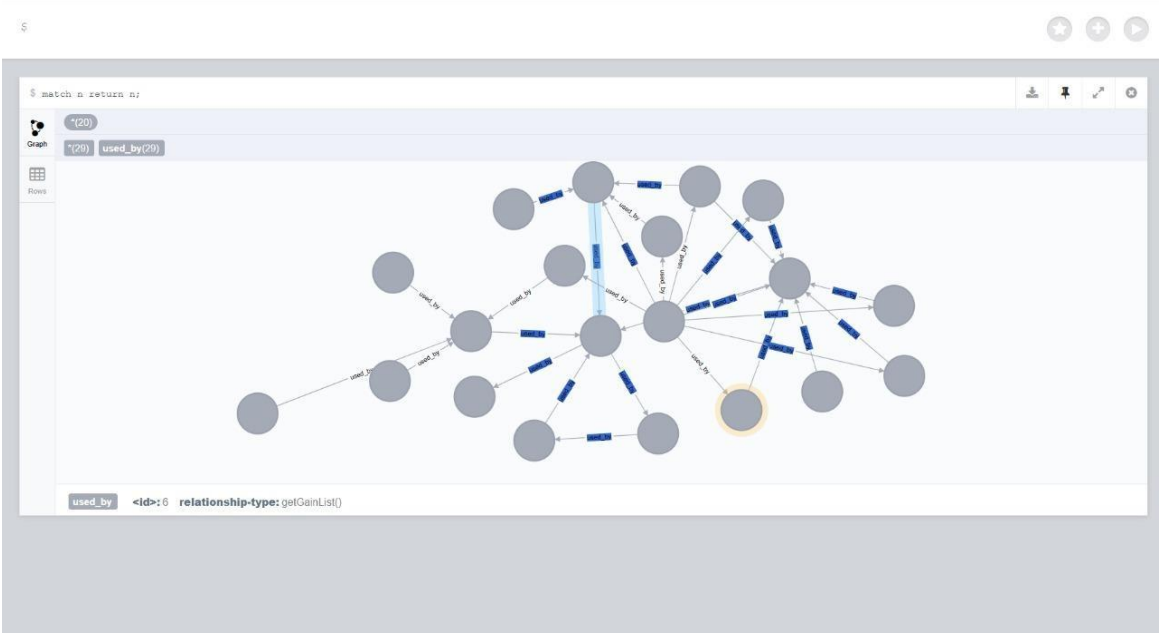


Figure 5.1: System Outputs

VI. CONCLUSION

6.1 Conclusion

The proposed system successfully automates the process of software code refactoring by leveraging a structured, graph-based approach. Through modules such as preprocessing, labeling, graph formation, and decision-making, the system efficiently identifies dependencies, extracts features, and applies predefined rules to enhance code structure. The results indicate a significant improvement in software maintainability, readability, and efficiency, reducing redundancy and ensuring optimized code without manual intervention. By minimizing human errors and automating repetitive tasks, the system proves to be a valuable tool for software developers.

The integration of machine learning techniques in feature extraction and decision-making enhances the accuracy and reliability of the refactoring process. The system demonstrated high efficiency in processing complex Java class structures, making it suitable for large-scale software projects. Future work can focus on expanding the system to support multiple programming languages and improving the decision-making rules using advanced AI techniques. Overall, the proposed approach contributes to the field of software engineering by providing a systematic and intelligent solution for code refactoring.

6.2 Future Work

In the future, the proposed system can be enhanced by incorporating deep learning techniques to improve the accuracy of feature extraction and decision-making. Expanding support for multiple programming languages will make the system more versatile and applicable across different software projects. Additionally, integrating real-time feedback mechanisms and adaptive learning models can refine refactoring suggestions based on evolving coding standards and best practices. Further research can focus on optimizing graph-based dependency evaluation for large-scale applications, ensuring scalability and efficiency. Lastly, implementing a user-friendly interface for developers to visualize refactoring suggestions and modifications in real-time can enhance usability and adoption.

BIBLIOGRAPHY

[1]. Maurício Aniche, Erick Maziero, Rafael Durelli, Vinicius Durelli, “The Effectiveness of Supervised Machine Learning Algorithms in Predicting Software Refactoring”, arXiv:2001.03338v3 [cs.SE] 11 Sep 2020, https://doi.org/10.48550/arXiv.2001.03338

IJARSCT
ISSN (Online) 2581-9429
International Journal of Advanced Research in Science, Communication and Technology (IJARSCT)
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
Impact Factor: 7.67
Volume 5, Issue 1, March 2025

[2]. A. M. Sheneamer, "An Automatic Advisor for Refactoring Software Clones Based on Machine Learning," in IEEE Access, vol. 8, pp. 124978-124988, 2020, doi: 10.1109/ACCESS.2020.3006178.

[3]. M. Alzahrani, "Measuring Class Cohesion Based on Client Similarities Between Method Pairs: An Improved Approach That Supports Refactoring," in IEEE Access, vol. 8, pp. 227901-227914, 2020, doi: 10.1109/ACCESS.2020.3046109.

[4]. Yuan Gaoa, Youchun Zhanga, Wenpeng Lub, Jie Luoc, and Daqing Haod, "A Prototype for Software Refactoring Recommendation System", vol. 16, no. 7, July 2020, pp. 1095-1104 DOI: 10.23940/ijpe.20.07.p12.10951104

[5]. C. Abid, K. Gaaloul, M. Kessentini and V. Alizadeh, "What Refactoring Topics Do Developers Discuss? A Large Scale Empirical Study Using Stack Overflow," in IEEE Access, vol. 10, pp. 56362-56374, 2022, doi: 10.1109/ACCESS.2021.3140036.

[6]. I. H. Moghadam, M. Ó. Cinnéide, F. Zarepour and M. A. Jahanmir, "RefDetect: A Multi-Language Refactoring Detection Tool Based on String Alignment," in IEEE Access, vol. 9, pp. 86698-86727, 2021, doi: 10.1109/ACCESS.2021.3086689.

[7]. Akour, M.; Alenezi, M.; Alsghaier, H. Software Refactoring Prediction Using SVM and Optimization Algorithms. Processes 2022, 10, 1611. https://doi.org/10.3390/pr10081611

[8]. A. Almogahed, M. Omar, N. H. Zakaria, G. Muhammad and S. A. AlQahtani, "Revisiting Scenarios of Using Refactoring Techniques to Improve Software Systems Quality," in IEEE Access, vol. 11, pp. 28800-28819, 2023, doi: 10.1109/ACCESS.2022.3218007.

[9]. Siddth Kumar Chhajer and Rudra Bhanu Satpathy, "Implication of Code Refactoring in Optimizing Software Security", TTACA. 2022 March; 1(1): 9-12. Published online 2022 March, doi.org/10.36647/TTACA/01.01.A003

[10]. A. Almogahed, H. Mahdin, M. Omar, N. H. Zakaria, G. Muhammad and Z. Ali, "Optimized Refactoring Mechanisms to Improve Quality Characteristics in Object-Oriented Systems," in IEEE Access, vol. 11, pp. 99143-99158, 2023, doi: 10.1109/ACCESS.2023.3313186.

[11]. Almogahed A, Mahdin H, Omar M, Zakaria NH, Gu YH, Al-masni MA, et al. (2023) A refactoring categorization model for software quality improvement. PLoS ONE 18(11): e0293742. https://doi.org/10.1371/journal.pone.0293742

[12]. A. Almogahed et al., "A Refactoring Classification Framework for Efficient Software Maintenance," in IEEE Access, vol. 11, pp. 78904-78917, 2023, doi: 10.1109/ACCESS.2023.3298678.

[13]. E. Edward, A. S. Nyamawe and N. Elisa, "On the Impact of Refactorings on Software Attack Surface," in IEEE Access, vol. 12, pp. 128570-128584, 2024, doi: 10.1109/ACCESS.2024.3404058.

[14]. Al-Fraihat, D., Sharrab, Y., Al-Ghuwairi, AR. et al. Detecting refactoring type of software commit messages based on ensemble machine learning algorithms. Sci Rep 14, 21367 (2024). https://doi.org/10.1038/s41598-024-72307-0

[15]. M. Alharbi and M. Alshayeb, "A Comparative Study of Automated Refactoring Tools," in IEEE Access, vol. 12, pp. 18764-18781, 2024, doi: 10.1109/ACCESS.2024.3361314.