Edelweiss Applied Science and Technology
ISSN: 2576-8484
Vol. 9, No. 7, 566-579
2025
Publisher: Learning Gate
DOI: 10.55214/25768484.v9i7.8672

Development of code smell detection model based on graph neural network to detect long method, large class, and duplicated code

Joko Slamet[1*], Siti Rochimah[2]
[1.2]Institut Teknologi Sepuluh Nopember, Surabaya, Indonesia; 6025231063@student.its.ac.id (J.S.) siti@its.ac.id (S.R.).

Abstract: This study aims to develop and evaluate a GNN-based model for automatically detecting code smells in Python by leveraging the structural information derived from Abstract Syntax Trees (ASTs). The study employed a quantitative approach by training a Graph Neural Network (GNN) model on code represented as Abstract Syntax Trees (ASTs). The proposed GNN model demonstrated high effectiveness in detecting specific types of code smells using a processed dataset, achieving 96.3% accuracy for Long Method detection and 95.2% for Large Class detection. Furthermore, the model effectively identified Long Method, Large Class, and Duplicated Code when tested on real-world datasets. On the ERPNext dataset, it reached 95.95% accuracy, 99.84% precision, 95.95% recall, and 97.78% F1-score. On the Odoo dataset, it attained 93.29% accuracy, 99.74% precision, 93.29% recall, and 96.29% F1-score. These results show that the proposed GNN model outperformed traditional machine learning algorithms such as Decision Tree, Random Forest, Support Vector Machine (SVM), Stochastic Gradient Descent (SGD), Multi-Layer Perceptron (MLP), and Linear Regression. The results confirm that the GNN-based approach effectively detects code smells in Python programs, surpassing classical machine learning techniques. This model provides a practical tool for enhancing code quality and maintainability. Future work could explore real-time integration of this model into development environments and expand detection to other code smell types.

Keywords: *Code smell detection, Duplicated code, Graph neural network, Large class, Long method, Python programming language.*

1. Introduction

In recent years, maintaining high software quality has become a critical concern in large-scale software development. One of the most persistent threats to code quality is the presence of code smells, which are symptoms in the source code that may indicate deeper design or structural issues [1]. While not directly causing program failure, code smells tend to degrade maintainability, increase technical debt, and introduce potential bugs in the long term [2]. Among the various types of code smells, Long Method, Large Class, and Duplicated Code are considered the most impactful due to their high frequency and detrimental influence on readability, testing, and modularity [3-5].

Code smell detection has traditionally relied on manual code review or rule-based static analysis tools, which are limited in scalability and often subjective. In response, many recent studies have applied machine learning (ML) techniques to automate code smell detection, using classifiers such as Decision Trees, Random Forests, Support Vector Machines (SVM), and Multilayer Perceptron trained on hand-engineered code metrics [6]. While these models offer improvements over rule-based tools, their dependence on manually extracted features introduces bias, limits generalization, and poses scalability issues in real-world scenarios.

To address these limitations, the software engineering community has shifted toward deep learning approaches that eliminate the need for manual feature engineering. Techniques such as Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs) have shown promising results in

History: Received: 2 May 2025; Revised: 12 June 2025; Accepted: 16 June 2025; Published: 8 July 2025
* Correspondence: 6025231063@student.its.ac.id

detecting code smells directly from raw code representations [7-9]. However, these approaches often fail to fully capture the rich structural and semantic relationships inherent in source code, which are crucial for detecting complex smells such as Large Class and Long Method.

Recent developments in Graph Neural Networks (GNNs) offer a more expressive framework for modeling code structure, particularly when applied to representations such as Abstract Syntax Trees (ASTs) or Function Call Graphs (FCGs). GNNs can learn contextual relationships between code elements, making them well-suited for detecting patterns that indicate code smells [10, 11].

Despite their potential, few studies have explored the use of GNNs for code smell detection in Python, a language increasingly used in modern software systems. The majority of existing works still focus on Java and use conventional ML-based detection techniques. For instance, Sandouka and Aljamaan [6] created a labeled Python dataset and evaluated various traditional ML models to detect Long Method and Large Class smells, but their models relied on static code metrics and lacked structural learning capability. In addition to this benchmark dataset, we also perform experiments using real-world codebases from ERPNext and Odoo two large-scale Python-based open-source enterprise resource planning (ERP) systems. These additional experiments aim to evaluate the generalization capability and robustness of the proposed model in more practical, industry-level scenarios.

The key contribution of this study lies in the novel application of Graph Neural Networks for detecting Long Method, Large Class, and Duplicated Code smells in Python source code. Unlike traditional ML approaches, our method utilizes AST-based graph representations to better capture structural and contextual relationships in the code. Furthermore, we provide a comprehensive empirical evaluation by comparing our GNN-based approach with previous machine learning methods. This demonstrates the effectiveness and adaptability of the proposed model in diverse software environments.

2. Literature Review

Several studies have investigated the problem of code smells in software systems, particularly focusing on their impact on software maintainability, complexity, and error-proneness. A growing body of research has explored both traditional and modern approaches to automatically detect code smells, especially Duplicated Code, Long Method, and Large Class, which are among the most frequently encountered and studied types in the literature.

Long Method and Large Class are smells closely related to violations of the Single Responsibility Principle (SRP). Long Method is characterized by lengthy and overly complex procedures that reduce code readability, hinder unit testing, and complicate debugging [5]. Large Class refers to classes overloaded with multiple responsibilities, often becoming central points of complexity and tightly coupled with other parts of the system [12]. Meanwhile, Duplicated Code is widely recognized for its prevalence and ease of detection. According to Kaur et al, it is the most studied code smell, with 35 out of 74 publications specifically addressing it Kaur [4]. Duplicated Code refers to repeated code blocks that often arise due to copy-paste practices. Although straightforward to identify, this smell poses serious maintainability risks. If changes are made to one copy of the duplicated code without synchronizing others, inconsistencies and bugs may occur.

To mitigate issues in software quality analysis, early studies have applied traditional machine learning (ML) techniques for code smell detection. These methods typically rely on manually engineered features, selected by practitioners and automatically extracted using static analysis tools such as Radon. Radon computes various software metrics, including cyclomatic complexity, Halstead complexity, and raw metrics such as lines of code, number of functions, and classes. These features are then used to train classification models such as Decision Tree, Random Forest, Support Vector Machine (SVM), Stochastic Gradient Descent (SGD), and Linear Regression [6].

The Decision Tree algorithm is known for its interpretability and rule-based structure, making it effective for identifying threshold-driven patterns in code smells. Khleel and Nehéz [13] demonstrated its effectiveness in detecting Long Method and Data Class smells [13]. Random Forest, as an ensemble of decision trees, reduces overfitting and improves generalization by averaging multiple model outputs,

Edelweiss Applied Science and Technology
ISSN: 2576-8484
Vol. 9, No. 7: 566-579, 2025
DOI: 10.55214/25768484.v9i7.8672

as shown by Sandouka and Aljamaan [6] in their evaluation on Python-based datasets [6]. Support Vector Machine constructs an optimal hyperplane in a high-dimensional space to maximize class separation, and Panigrahi, et al. [14] demonstrated that SVM with SMOTE outperforms weighted-SVM, with the RBF kernel achieving the best results among various SVM kernel types [14]. Stochastic Gradient Descent is a linear model optimized using iterative weight updates, offering scalability to large datasets; its role as a baseline model was explored by Liu, et al. [15] in comparative evaluations involving various code smell types [15]. Linear Regression, although traditionally used for regression tasks, has also been employed for binary classification of code smells by mapping feature combinations to probability estimates; Gupta, et al. [16] showed that while it performs reasonably on simple smells, its linear nature limits its accuracy on more complex patterns [16]. While these conventional models offer useful benchmarks, their reliance on handcrafted features and limited contextual understanding of source code highlights the potential of more expressive models such as Graph Neural Networks.

While traditional machine learning models such as Decision Tree, Random Forest, Support Vector Machine, and Stochastic Gradient Descent have shown promise in detecting code smells, their reliance on manually engineered features often limits their ability to capture deeper semantic and structural relationships within source code. Graph Neural Networks (GNNs) have emerged as a powerful alternative due to their ability to model code as graphs, capturing both syntactic and contextual dependencies through representations such as Abstract Syntax Trees (AST) and Control Flow Graphs (CFGs) [17-19]. Recent studies have demonstrated that GNNs outperform traditional models in tasks like vulnerability detection, program classification, and code smell identification by leveraging structural inductive biases and relational information between code elements. However, despite their growing adoption, existing GNN-based approaches are often evaluated on limited types of code smells or small-scale datasets, which restricts their generalizability to large, complex software systems [11]. Moreover, few studies perform comprehensive comparisons between GNNs and conventional models using diverse performance metrics, leaving open questions about the practical advantages of GNNs across varying project contexts. Therefore, this research seeks to fill that gap by employing a GNN-based model for detecting multiple code smells across two large-scale datasets and evaluating its performance using a full set of metrics, such as accuracy, precision, recall, F1-score, and confusion matrix.

3. Methodology

The methodology employed in this study follows a systematic approach consisting of several key steps: dataset preparation, code smell labeling, Abstract Syntax Tree generation, GNN model development, training process, and evaluation, as illustrated in Figure 1. This comprehensive approach ensures a thorough analysis of the effectiveness of Graph Neural Networks in detecting code smells in Python programs.

Edelweiss Applied Science and Technology
ISSN: 2576-8484
Vol. 9, No. 7: 566-579, 2025
DOI: 10.55214/25768484.v9i7.8672

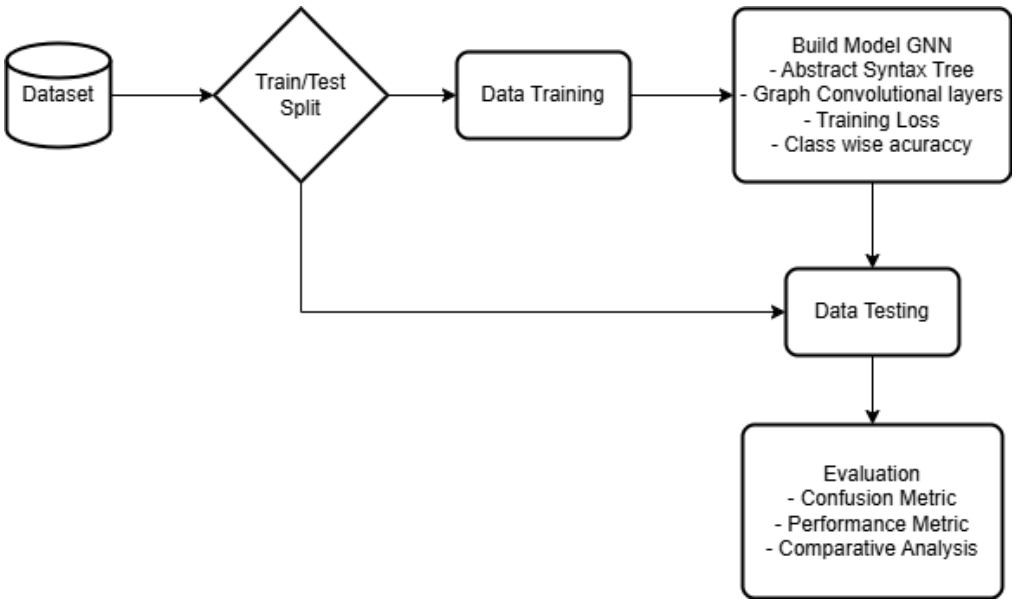


Figure 1.
Methodology for Code Smell Detection Using Graph Neural Network

3.1. Dataset

This study employs two types of datasets to support the development and evaluation of a code smell detection model: processed datasets and raw source code datasets. The first type, processed datasets, is adopted from the work of Sandouka and Aljamaan [6] who focused on detecting code smells using machine learning techniques. These datasets, provided in CSV format, include the Python Large Class Smell Dataset and the Python Long Method Smell Dataset, both of which contain pre-extracted code metrics and binary labels indicating the presence or absence of code smells. Each entry in these datasets includes features such as lines of code (LOC), logical lines of code (LLOC), number of comments, blank lines, and the is_code_smell label. These structured and labeled datasets offer a reliable basis for training and validating our Graph Neural Network (GNN) model.

In addition to the processed data, this study also incorporates raw source code datasets collected from two popular open-source ERP systems, ERPNext and Odoo. These repositories were cloned from their official GitHub sources and selected due to their maturity, active development communities, and widespread use. ERPNext has approximately 7.3k forks and 21.8k stars, while Odoo has around 27.2k forks and 42.1k stars. To ensure relevance and manageability, only specific folders were extracted: the controller folder from ERPNext and the tools folder from Odoo. These modules were chosen for their structural complexity and representativeness of real-world software systems, making them suitable for code smell detection research, particularly for identifying Long Method, Large Class smells, and Duplicated Code [20].

Table 1.
Sample metrics from processed datasets.

loc	lloc	comments	blanks	difficulty	effort	bugs	LargeClass
469	247	10	53	5.764423	6945.013	0.401602	1
141	84	1	31	1	18.09474	0.006032	1
473	259	51	59	7.418478	8019.327	0.360331	1
1267	721	78	161	6.986014	29168.97	1.391779	1
174	114	16	19	4.5	1708.999	0.126592	1
11	8	0	3	1.5	216	0.048	0
109	92	2	11	4.846154	15166.08	1.04317	0
34	27	0	5	1.8	202.6584	0.037529	0

Edelweiss Applied Science and Technology
ISSN: 2576-8484
Vol. 9, No. 7: 566-579, 2025
DOI: 10.55214/25768484.v9i7.8672

As shown in Table 1, instances with higher metrics, tend to be labeled as code smells (1), while instances with lower metrics typically do not contain smells (0). This pattern helps in training machine learning models to recognize potential code smells based on these structural metrics [21].

The labeling process is only conducted in experiments that utilize the raw dataset. In this study, the identification of code smells is performed using a metric-based approach. These metrics provide an objective framework to define thresholds for detecting different types of code smells, such as Long Method, Duplicated Code, and Large Class.

Long Method refers to a function or method that contains an excessive number of lines of code. The detection of the Long Method can be performed using Equation (1), where Lines of Code (LOC) represents the total number of lines in a function, and the threshold denotes the code length limit. According to the study by Nandani, et al. [22] a method is classified as a Long Method if it exceeds 80 lines of code [22].

$$Long\ Method => True\ if\ LOC > threshold \quad (1)$$

Duplicated Code occurs when there are identical or highly similar blocks of code repeated across different parts of a program. This study adopts a String Matching or Substring Similarity approach to detect code duplication. The method compares different code blocks within a project using similarity algorithms, particularly the Longest Common Subsequence (LCS). The LCS algorithm is used to measure the similarity between two code sequences, A and B. The formula for LCS similarity is shown in Equation (2).

$$LCS(A,B) = \frac{lenght\ of\ longest\ subsequence\ of\ A\ and\ B}{\min(len(A),len(B))} \quad (2)$$

The numerator refers to the longest subsequence that appears in both A and B while preserving their order. The denominator represents the minimum length of the two sequences A and B, serving as a normalization factor. If two or more code blocks have a similarity of 80% or higher, they are considered duplicates [23].

$$Large\ Class = True\ if\ WMC > threshold \quad (3)$$

Large Class refers to a class that assumes excessive responsibilities or contains an overwhelming number of functions, thereby violating the Single Responsibility Principle. Large Class detection is performed using Equation (3). Weighted Method Count (WMC) represents the number of methods within a class, while the threshold defines the method count limit. According to Turkistani & Liu, a class is considered a Large Class if it contains more than 47 methods [24].

Edelweiss Applied Science and Technology
ISSN: 2576-8484
Vol. 9, No. 7: 566-579, 2025
DOI: 10.55214/25768484.v9i7.8672

3.2. Abstract Syntax Tree

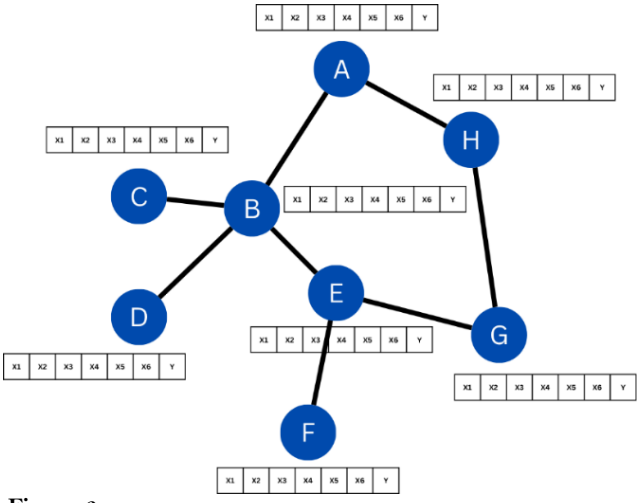


Figure 2.
Abstract Syntax Tree.

After dataset preparation, the study extracts call graphs from each source code using the Abstract Syntax Tree (AST), as shown in Figure 2. In this process, each function within the code is represented as a node, while function calls from one function to another serve as edges (relationships) within the graph, following the approach outlined by Ward, et al. [18]. This representation captures the structural and semantic relationships between functions, enabling a deeper understanding of the program's behavior. The AST facilitates a systematic decomposition of source code into a tree structure, allowing for efficient parsing and analysis. By transforming source code into graph form, it becomes suitable input for Graph Neural Networks (GNN), which can leverage topological patterns to detect code smells. This method ensures that both syntactic structure and functional interaction are preserved in the modeling process.

3.3. Graph Neural Network

The generated function call graph is then converted into a format compatible with PyTorch Geometric to facilitate the training process of the Graph Neural Network (GNN). Each node in the graph is equipped with features that represent the structural information of the corresponding function, such as the number of calls made by the function, the number of parameters received, or the function's complexity based on specific metrics [25]. This information provides additional context for the GNN to understand the characteristics of each function in the graph, ultimately improving the model's ability to accurately detect and classify code smells.

The Graph Neural Network (GNN) model used in this project is the Graph Convolutional Network (GCN), specifically designed to work with directed graphs, where the direction of each edge plays a crucial role in defining relationships between nodes. Figure 3 illustrates the architecture of the GNN model, showing how graph data is processed through multiple hidden layers. This model employs three convolutional layers to process and analyze data in the Function Call Graph (FCG), which represents each function in the program as a node, with edges indicating function calls between functions [26].

Edelweiss Applied Science and Technology
ISSN: 2576-8484
Vol. 9, No. 7: 566-579, 2025
DOI: 10.55214/25768484.v9i7.8672

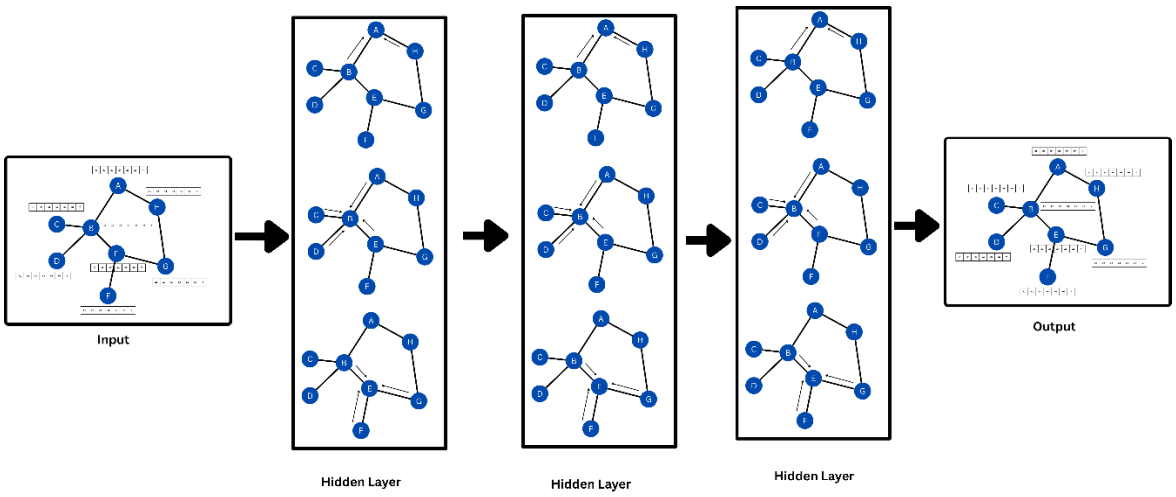


Figure 3.
Graph Neural Network Architecture showing input processing through hidden layers with ReLU activation.

3.2.1. Message Passing and Node Representation

The core mechanism behind GNNs is message passing. In this process, each node updates its representation by receiving and aggregating information from its neighboring nodes. This is done iteratively across multiple layers, allowing a node to gradually incorporate more context from its surrounding subgraph [27]. As a result, the final node embeddings encode not only the features of the node itself but also the structure and features of its neighborhood. This iterative aggregation enables the network to learn how information flows through the graph, which is critical for tasks like classification, prediction, and anomaly detection on graph-structured data.

3.2.2. Graph Convolutional Networks (GCNs)

One widely adopted variant of GNN is the Graph Convolutional Network (GCN). GCNs apply a form of graph convolution where each node's representation is updated by combining it with normalized contributions from its neighbors. Unlike traditional convolutional layers used in image processing, graph convolution operates on irregular structures and does not require a fixed grid layout. GCNs are efficient and scalable, making them suitable for large graphs. They also maintain permutation invariance, meaning that the order of the nodes does not affect the result—a crucial property for graph data [28].

3.2.3. Output Representations and Learning Objectives

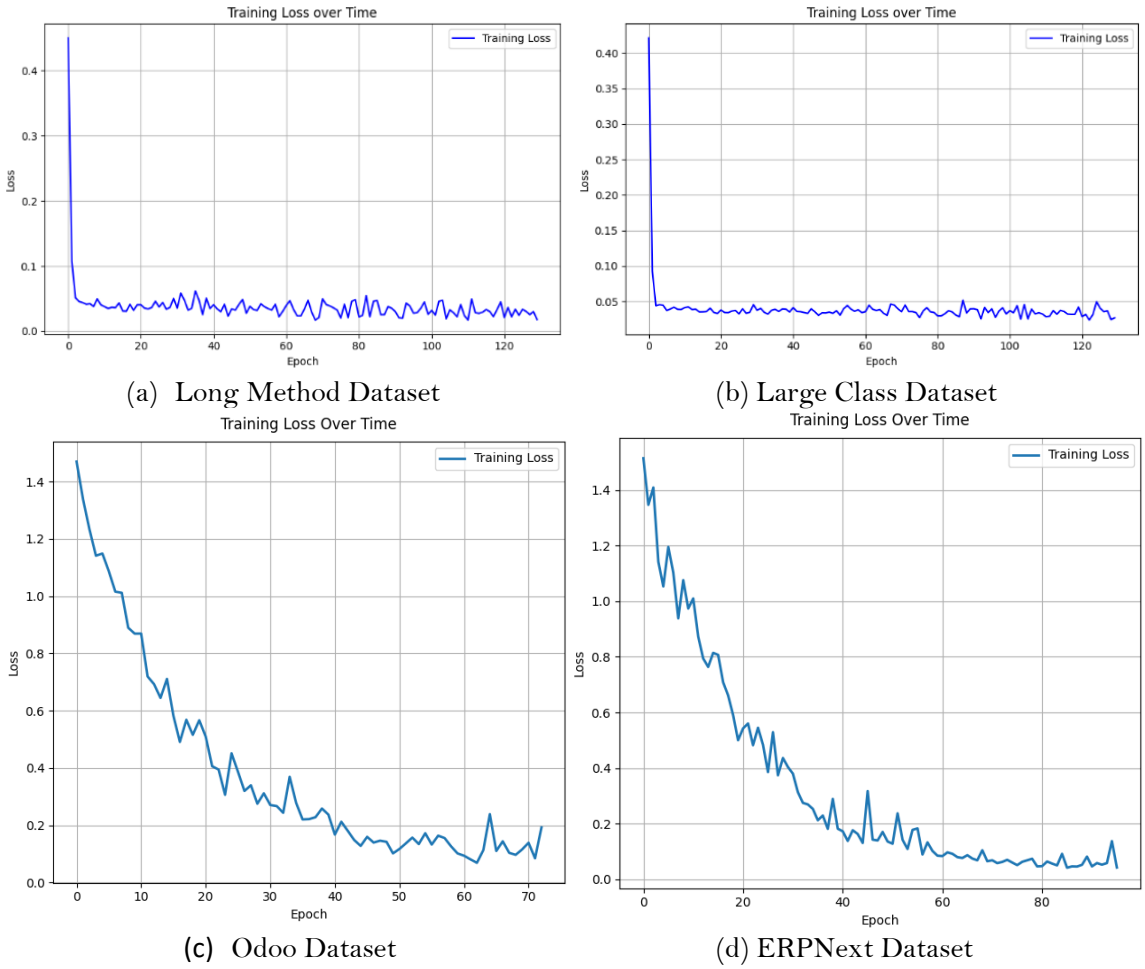
Depending on the task, GNNs can be used to produce representations at the node level, edge level, or whole-graph level. For example, in node-level tasks such as code smell detection, the final output of each node is passed through a classifier to determine whether it exhibits certain characteristics or anomalies. For graph-level tasks, the node embeddings are aggregated into a single graph representation, which is then used for prediction or classification. Training is typically done using supervised learning with labeled data, where the model is optimized to minimize the difference between predicted and actual labels [29].

3.4. Training Model GNN

Figure 4 presents the training loss over time for four different datasets: Long Method, Large Class, Odoo, and ERPNext. The Long Method (a) and Large Class (b) datasets are binary classification tasks, where the label is 1 if the code contains the corresponding code smell (Long Method or Large Class) and 0 otherwise. In both cases, the training loss drops sharply in the early epochs and quickly stabilizes

Edelweiss Applied Science and Technology
ISSN: 2576-8484
Vol. 9, No. 7: 566-579, 2025
DOI: 10.55214/25768484.v9i7.8672

at a low value, indicating fast convergence and effective learning of simple binary patterns. In contrast, the Odoo (c) and ERPNext (d) datasets are multi-class classification tasks with four classes: Long Method, Large Class, Duplicated Code, and No Smell. The training loss in these datasets decreases more gradually and shows more fluctuations, reflecting higher complexity due to the increased number of classes and greater variation in code structure.



(a) Long Method Dataset (b) Large Class Dataset

(c) Odoo Dataset (d) ERPNext Dataset

Figure 4.
Training Loss Over Time.

Figure. 5 illustrates the class-wise accuracy of our GNN model across training epochs for different code smell types. The graph shows four distinct learning curves representing No Smell, Long Method, Large Class, and Duplicated Code categories. The model demonstrates rapid improvement in accuracy during the first 40 epochs, with Duplicated Code (red line) and Large Class (green line) detection showing faster convergence compared to Long Method (orange line) and No Smell (blue line). This suggests that structural patterns associated with Large Class and Duplicated Code are more distinctive and easier for the model to recognize. The No Smell category requires more training epochs to reach high accuracy, indicating that distinguishing clean code from code with smells is a more nuanced task for the model.

Edelweiss Applied Science and Technology
ISSN: 2576-8484
Vol. 9, No. 7: 566-579, 2025
DOI: 10.55214/25768484.v9i7.8672

(a)ERP Dataset (b)Odoo Dataset

Figure. 5.
Class-wise accuracy of the GNN model across training epochs for different code smell types.

The training process shows some fluctuations, particularly for Large Class detection, which exhibits occasional drops in accuracy even after epoch 60. These fluctuations may be attributed to the relative scarcity of Large Class instances in the training data. Nevertheless, all categories eventually achieve accuracy above 95% by the end of training, demonstrating the effectiveness of the GNN approach for code smell detection [30, 31].

4. Results and Discussion

In this section, it is explained the results of the research and at the same time, a comprehensive discussion is provided. Results can be presented in figures, graphs, tables, and other formats that make the reader understand easily. The discussion can be made in several sub-sections.

4.1. Comparative Analysis with Conventional Machine Learning Models

The experimental results demonstrate that our GNN model significantly outperforms traditional machine learning approaches for code smell detection. Tables 3 and 4 present a comparative analysis of accuracy metrics for Long Method and Large Class detection, respectively.

Our evaluation compares the GNN model against several established machine learning algorithms, including Decision Tree, Random Forest, Support Vector Machine (SVM), Stochastic Gradient Descent (SGD), Multilayer Perceptron (MLP), and Linear Regression. Each model was trained and evaluated using identical datasets to ensure a fair comparison. The traditional models rely on manually extracted features such as lines of code (LOC), logical lines of code (LLOC), comment density, and cyclomatic complexity, while our GNN model leverages the structural information encoded in function call graphs. [33].

The key advantage of the GNN approach lies in its ability to automatically learn relevant features from graph representations of code, thereby capturing complex relationships and dependencies that are difficult to express as explicit metrics. This capability is particularly valuable in the context of code smell detection, where the presence of a smell often depends on structural patterns rather than simple metric thresholds.

4.1.1. Long Method Detection Performance

As shown in Table 3, the GNN model achieves the highest accuracy (96.3%) for Long Method detection, surpassing all traditional machine learning models. Decision Tree (95.9%) and Random Forest (95.5%) perform relatively well but still fall short of the GNN's capability. This superior

Edelweiss Applied Science and Technology
ISSN: 2576-8484
Vol. 9, No. 7: 566-579, 2025
DOI: 10.55214/25768484.v9i7.8672

performance can be attributed to the GNN's ability to capture structural dependencies in code through the function call graph representation, which is particularly important for identifying Long Method smells.

Table 3.
Long method detection accuracy: Graph Neural Network vs. Conventional machine learning models.

Research	Model	Accuracy
Sandouka and Aljamaan [6]	Decision Tree	95.9%
	Random Forest	95.5%
	Linear Regression	85.3%
	Support Vector Machine	93.7%
	Multilayer Perceptron	90.2%
	Stochastic Gradient Descent	90.9%
Proposed	Graph Neural Network	96.3%

4.1.2. Large Class Detection Performance

Table 4 presents similar findings for Large Class detection, where the GNN model again achieves the highest accuracy at 95.2%. Interestingly, the performance gap between GNN and the best traditional model (Stochastic Gradient Descent at 94.4%) is smaller for Large Class detection compared to Long Method detection. This suggests that while structural information is valuable for Large Class detection, traditional metrics-based approaches can still be effective for this particular code smell.

The relatively strong performance of traditional models for Large Class detection can be explained by the fact that this code smell is often characterized by straightforward metrics such as the number of methods or lines of code in a class. Nevertheless, the GNN model's superior performance highlights the value of incorporating structural information, even for code smells that have a strong correlation with simple metrics.

Table 4.
Large class detection accuracy: Graph Neural Network vs. Conventional Machine Learning Models

Research	Model	Accuracy
Sandouka and Aljamaan [6]	Decision Tree	90.4%
	Random Forest	92.7%
	Linear Regression	87.7%
	Support Vector Machine	92.5%
	Multilayer Perceptron	91.8%
	Stochastic Gradient Descent	94.4%
Proposed	Graph Neural Network	95.2%

4.2. Evaluation Results of Code Smell Detection Model

Table 5 presents the performance metrics of the GNN model in detecting code smells, specifically Long Method, Large Class, and Duplicated Code. The evaluation results indicate that the detection model performs better on the ERPNext dataset compared to the Odoo dataset across all key metrics. Specifically, the model achieves an accuracy of 95.95% on ERPNext and 93.29% on Odoo, reflecting a higher overall correctness in classification. Both datasets yield high precision, with ERPNext at 99.84% and Odoo at 99.74%, indicating the model's effectiveness in reducing false positives. For recall, ERPNext again shows better performance at 95.95%, compared to Odoo's 93.29%, suggesting a higher ability to identify all relevant code smells. This balance is further validated by the F1-Score, where ERPNext scores 97.78% and Odoo scores 96.29%. These results suggest that the model is more robust and effective when applied to the ERPNext dataset in detecting code smells.

Edelweiss Applied Science and Technology
ISSN: 2576-8484
Vol. 9, No. 7: 566-579, 2025
DOI: 10.55214/25768484.v9i7.8672

Table 5.
Performance comparison of code smell detection model on ERPNext and Odoo datasets.

Dataset	Accuracy	Precision	Recall	F1-Score
ERPNext	95.95%	99.84%	95.95%	97.78%
Odoo	93.29%	99.74%	93.29%	96.29%

4.3. Model Interpretability and Practical Implications

To gain deeper insights into how our GNN model makes predictions, we analyzed the feature importance and attention mechanisms within the graph convolutional layers. This analysis revealed the specific structural patterns that the model focuses on when detecting each type of code smell.

For Long Method detection, the model primarily attends to complex control flow structures, nested loops, and high cyclomatic complexity within methods. We observed that methods with a high number of conditional statements and deeply nested blocks were consistently classified as Long Method smells, even when their raw line count was moderate. This suggests that the GNN model learns to recognize not just the length of methods but also their structural complexity.

For large-class detection, the analysis showed that the model puts significant weight on the number of methods in a class and the complexity of interactions between them. Classes with numerous methods that interact extensively with each other were more likely to be flagged as Large Class smells. Interestingly, the model also identified classes with high coupling to other classes as potential Large Class candidates, suggesting it captures the design principle that classes should have high cohesion and low coupling. For Duplicated Code detection, our analysis revealed that the model focuses on semantic similarities rather than exact textual matches. It successfully identified code segments with similar Abstract Syntax Tree (AST) structures as potential duplications, even when variable names and formatting differed. This demonstrates the advantage of the graph-based approach over traditional token-based similarity measures [17].

These insights have significant practical implications for software development. By understanding which structural patterns contribute most to code smell detection, developers can proactively avoid these patterns during coding. Furthermore, the model can be integrated into Integrated Development Environments (IDEs) as a real-time code quality advisor, alerting developers to potential code smells as they write code and suggesting refactoring strategies. Our experiments also revealed areas where the model could be further improved. For instance, contextual factors such as the domain of the software and project-specific coding conventions can influence what constitutes a code smell. Future work could explore techniques to adapt the model to different project contexts and developer preferences, potentially through transfer learning or few-shot learning approaches.

4.4. Limitations and Future Research Directions

While our GNN model demonstrates superior performance in detecting code smells in Python, several limitations should be acknowledged. First, the model's performance is dependent on the quality and diversity of the training dataset. Our current dataset, though substantial, may not represent all possible variations of code smells in real-world Python projects [32]. Future work should focus on expanding the dataset to include more diverse and complex code samples from various domains and project types.

Second, the model's effectiveness may vary across different programming styles and paradigms. Python supports multiple programming paradigms, including procedural, object-oriented, and functional programming. Our current model primarily focuses on object-oriented code, and its performance on functional or procedural code may require further evaluation [33].

Third, the current approach treats code smell detection as a binary classification problem (smell present or absent) without considering the severity of the smell. In practice, the impact of a code smell depends on its severity and the specific context of the project. Future research could explore multi-level classification or regression approaches to assess the severity of detected code smells [34]

Edelweiss Applied Science and Technology
ISSN: 2576-8484
Vol. 9, No. 7: 566-579, 2025
DOI: 10.55214/25768484.v9i7.8672

An interesting direction for future research is the integration of natural language processing (NLP) techniques to analyze code comments and documentation alongside the structural analysis. This multi-modal approach could provide a more comprehensive understanding of code quality by considering both the structural aspects of code and the developers' intentions expressed in comments.

Additionally, the application of explainable AI techniques could enhance the practical utility of the model by providing more detailed explanations of detected code smells. Such explanations would not only help developers understand why certain code segments are flagged as smells but also guide them in refactoring decisions [35].

Finally, extending the model to support incremental analysis would be valuable for real-time integration with development workflows. Rather than analyzing the entire codebase after each change, an incremental approach would focus only on modified code segments, providing immediate feedback to developers during the coding process and significantly reducing computational overhead for large project

5. Conclusions

This research successfully developed a Graph Neural Network (GNN) model for detecting three prevalent code smells in Python: Long Method, Large Class, and Duplicated Code. As stated in the introduction, our goal was to leverage the structural information in code through graph representations to improve detection accuracy compared to traditional approaches. The results have demonstrated the effectiveness of this approach, with the GNN model achieving superior accuracy compared to conventional machine learning methods.

The proposed Graph Neural Network (GNN) model demonstrated high effectiveness in detecting specific types of code smells using a processed dataset, achieving 96.3% accuracy for Long Method detection and 95.2% for Large Class detection. These results show that the proposed GNN model outperformed traditional machine learning algorithms such as Decision Tree, Random Forest, Support Vector Machine (SVM), Stochastic Gradient Descent (SGD), Multi-Layer Perceptron (MLP), and Linear Regression

Furthermore, the model effectively identified Long Method, Large Class, and Duplicated Code when tested on real-world datasets. On the ERPNext dataset, it reached, the model achieved 95.95% accuracy, 99.84% precision, 95.95% recall, and a 97.78% F1-score. On the Odoo dataset, it reached 93.29% accuracy, 99.74% precision, 93.29% recall, and a 96.29% F1-Score. These results validate the effectiveness of GNNs in learning structural patterns from actual codebases.

The proposed approach addresses the limitations of previous detection methods by eliminating the need for manual feature engineering and capturing complex relationships between code elements through graph representations. By converting Python code into Abstract Syntax Trees and then into function call graphs, our model can learn structural patterns that characterize different code smells, providing a more robust and context-aware detection mechanism.

Future research could focus on extending the model to handle additional code smell types and adapting it to different programming languages and paradigms. The integration of natural language processing techniques to analyze code comments alongside structural analysis presents an interesting direction for multi-modal code smell detection. Additionally, incorporating explainable AI techniques could enhance the practical utility of the model by providing developers with clear explanations for detected code smells and specific refactoring recommendations.

The proposed GNN-based approach has significant implications for improving code quality in software development. By providing accurate and automated code smell detection, it can help developers identify and refactor problematic code early in the development process, leading to more maintainable and robust software systems. The potential integration of this model into development environments could provide real-time feedback to developers, promoting better coding practices and reducing technical debt in Python projects.

Edelweiss Applied Science and Technology
ISSN: 2576-8484
Vol. 9, No. 7: 566-579, 2025
DOI: 10.55214/25768484.v9i7.8672

Transparency:

The authors confirm that the manuscript is an honest, accurate, and transparent account of the study; that no vital features of the study have been omitted; and that any discrepancies from the study as planned have been explained. This study followed all ethical practices during writing.

References

[1] D. Albuquerque, E. Guimarães, M. Perkusich, H. Almeida, and A. Perkusich, "Integrating interactive detection of code smells into scrum: Feasibility, benefits, and challenges," *Applied Sciences*, vol. 13, no. 15, p. 8770, 2023.

[2] P. Gnoyke, S. Schulze, and J. Krüger, "Evolution patterns of software-architecture smells: An empirical study of intra-and inter-version smells," *Journal of Systems and Software*, vol. 217, p. 112170, 2024.

[3] T. Sharma, P. Singh, and D. Spinellis, "An empirical investigation on the relationship between design and architecture smells," *Empirical Software Engineering*, vol. 25, pp. 4020-4068, 2020. https://doi.org/10.1007/s10664-020-09847-2

[4] A. Kaur, "A systematic literature review on empirical analysis of the relationship between code smells and software quality attributes," *Archives of Computational Methods in Engineering*, vol. 27, no. 4, pp. 1267-1296, 2020. https://doi.org/10.1007/s11831-019-09348-6

[5] A. Kovačević *et al.*, "Automatic detection of Long Method and God Class code smells through neural source code embeddings," *Expert Systems with Applications*, vol. 204, p. 117607, 2022. https://doi.org/10.1016/j.eswa.2022.117607

[6] R. Sandouka and H. Aljamaan, "Python code smells detection using conventional machine learning models," *PeerJ Computer Science*, vol. 9, p. e1370, 2023.

[7] Y. Zhang and C. Dong, "MARS: Detecting brain class/method code smell based on metric–attention mechanism and residual network," *Journal of Software: Evolution and Process*, vol. 36, no. 1, p. e2403, 2024. https://doi.org/10.1002/smr.2403

[8] G. Giray, K. E. Bennin, Ö. Köksal, Ö. Babur, and B. Tekinerdogan, "On the use of deep learning in software defect prediction," *Journal of Systems and Software*, vol. 195, p. 111537, 2023. https://doi.org/10.1016/j.jss.2022.111537

[9] O. Fawaz, M. Amaan, S. Sahu, M. Adnan, and A. Gupta, "Experimentation of code smells using deep learning techniques," presented at the 2023 6th International Conference on Contemporary Computing and Informatics (IC3I) (Vol. 6, pp. 369-373). IEEE, 2023.

[10] L. Šikić, A. S. Kurdija, K. Vladimir, and M. Šilić, "Graph neural network for source code defect prediction," *IEEE Access*, vol. 10, pp. 10402-10415, 2022.

[11] J. Zhou *et al.*, "Graph neural networks: A review of methods and applications," *AI open*, vol. 1, pp. 57-81, 2020.

[12] A. T. Imam, B. R. Al-Srour, and A. Alhroob, "The automation of the detection of large class bad smell by using genetic algorithm and deep learning," *Journal of King Saud University-Computer and Information Sciences*, vol. 34, no. 6, pp. 2621-2636, 2022.

[13] N. A. A. Khleel and K. Nehéz, "Detection of code smells using machine learning techniques combined with data-balancing methods," *International Journal of Advances in Intelligent Informatics*, vol. 9, no. 3, pp. 402-417, 2023. https://doi.org/10.26555/ijain.v9i3.981

[14] R. Panigrahi, S. K. Kuanar, and L. Kumar, "Method level refactoring prediction by weighted-svm machine learning classifier," *Studies in Systems, Decision and Control*, vol. 452, 2023. https://doi.org/10.1007/978-981-19-6893-8_8

[15] L. Liu *et al.*, "Revisiting code smell severity prioritization using learning to rank techniques," *Expert Systems with Applications*, vol. 249, p. 123483, 2024.

[16] A. Gupta, B. Suri, V. Kumar, S. Misra, T. Blažauskas, and R. Damaševičius, "Software code smell prediction model using Shannon, Rényi and Tsallis entropies," *Entropy*, vol. 20, no. 5, p. 372, 2018. https://doi.org/10.3390/e20050372

[17] W. Xu and X. Zhang, "Multi-granularity code smell detection using deep learning method based on abstract syntax tree," in *Proceedings of the International Conference on Software Engineering and Knowledge Engineering, SEK*, 2021.

[18] I. R. Ward, J. Joyner, C. Lickfold, Y. Guo, and M. Bennamoun, "A practical tutorial on graph neural networks," *ACM Computing Surveys*, vol. 54, no. 10s, pp. 1-35, 2022. https://doi.org/10.1145/3503043

[19] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and P. S. Yu, "A comprehensive survey on graph neural networks," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 32, no. 1, pp. 4-24, 2020. https://doi.org/10.1109/TNNLS.2020.2978386

[20] D. Nurmukhamet and A. Tick, "Implementing ERP through the use of mobile technologies: A case study," presented at the 2022 IEEE 16th International Symposium on Applied Computational Intelligence and Informatics (SACI), IEEE, 2022, pp. 311–316, 2022.

[21] S. Dewangan, R. S. Rao, A. Mishra, and M. Gupta, "Code smell detection using ensemble machine learning algorithms," *Applied Sciences*, vol. 12, no. 20, p. 10321, 2022.
[22] H. Nandani, M. Saad, and T. Sharma, "DACOS - a manually annotated dataset of code smells," in *Proceedings - 2023 IEEE/ACM 20th International Conference on Mining Software Repositories, MSR 2023*, 2023.
[23] M. Djukanovic, G. R. Raidl, and C. Blum, "Finding longest common subsequences: New anytime A∗ search results," *Applied Soft Computing*, vol. 95, p. 106499, 2020.
[24] B. Turkistani and Y. Liu, "Reducing the large class code smell by applying design patterns," presented at the IEEE International Conference on Electro Information Technology, 2019.
[25] N. A. A. Khleel and K. Nehéz, "Deep convolutional neural network model for bad code smells detection based on oversampling method," *Indonesian Journal of Electrical Engineering and Computer Science*, vol. 26, no. 3, pp. 1725-1735, 2022.
[26] S. Tarwani and A. Chug, "Application of deep learning models for code smell prediction," presented at the 2022 10th International Conference on Reliability, Infocom Technologies and Optimization (Trends and Future Directions)(ICRITO) (pp. 1-5). IEEE, 2022.
[27] B. Khemani, S. Patil, K. Kotecha, and S. Tanwar, "A review of graph neural networks: Concepts, architectures, techniques, challenges, datasets, applications, and future directions," *Journal of Big Data*, vol. 11, no. 1, p. 18, 2024.
[28] X. Shi, F. Lv, D. Seng, J. Zhang, J. Chen, and B. Xing, "Visualizing and understanding graph convolutional network," *Multimedia Tools and Applications*, vol. 80, pp. 8355-8375, 2021.
[29] J. You, J. M. Gomes-Selman, R. Ying, and J. Leskovec, "Identity-aware graph neural networks," in *Proceedings of the AAAI conference on artificial intelligence (Vol. 35, No. 12, pp. 10737-10745)*, 2021.
[30] R. Sandouka and A. Hamoud, "Dataset python code smells detection using conventional machine learning models," 2024. https://zenodo.org/records/7512516#.ZEg3EnbMLIU
[31] C. Frappe, "Technologies Pvt Ltd (Frappe), "erpnext," 2024. https://github.com/frappe/erpnext
[32] Mahmoud, "Awesome python application," 2024. https://github.com/mahmoud/awesome-python-applications
[33] M. T. Aras and Y. E. Selçuk, "Metric and rule based automated detection of antipatterns in object-oriented software systems," presented at the 2016 7th International Conference on Computer Science and Information Technology (CSIT), IEEE, 2016, pp. 1–6, 2016.
[34] A. Bhave and R. Sinha, "Deep multimodal architecture for detection of long parameter list and switch statements using distilbert," presented at the 2022 IEEE 22nd International Working Conference on Source Code Analysis and Manipulation (SCAM) (pp. 116-120). IEEE, 2022.
[35] E. A. AlOmar, M. W. Mkaouer, and A. Ouni, "Behind the intent of extract method refactoring: A systematic literature review," *IEEE Transactions on Software Engineering*, vol. 50, no. 4, pp. 668-694, 2024.

Edelweiss Applied Science and Technology
ISSN: 2576-8484
Vol. 9, No. 7: 566-579, 2025
DOI: 10.55214/25768484.v9i7.8672