Riya Sachan, et.al. International Journal of Engineering Research and Applications
www.ijera.com
ISSN: 2248-9622, Vol. 15, Issue 9, September 2025, pp 53-58

RESEARCH ARTICLE OPEN ACCESS

A Reinforcement Learning-Based Techniques for Automated Code Smells Detection and Refactoring

Riya Sachan[1], Rakesh Kumar Tiwari[2], Onkar Nath Thakur[3], Dr. Mayank Pathak[4]

MTech Scholar[1] , Assistant Professor[2&3], professor[4]
DepartmentofComputerScience&Engineering[1,2,3&4]
Technocrats Institute of Technology & Science, Bhopal. India[1,2,3&4]

Abstract—
Code smells negatively impact the maintainability, readability, and overall quality of software. Traditional detection methods primarily rely on rule-based approaches and manual intervention, making them less adaptable and inflexible in handling diverse codebases. These methods often fail to capture the underlying semantics of the code, limiting their ability to provide meaningful suggestions—such as appropriate method names. The current existing researches don't focus on Reinforcement learning approach for code smells detection and refactoring. The few existing studies that do apply RL typically concentrate on a single refactoring operation—such as the Extract Method—rather than encompassing comprehensive smell detection or supporting multiple refactoring strategies. This paper proposes a reinforcement learning (RL)-based framework for automatic detection and refactoring of multiple code smells in Java programs, including Long Method, God Class, and Feature Envy. The proposed system integrates structural features (AST paths, code metrics) and semantic embeddings (via Code BERT) to construct rich state representations. A multi-objective reward function guides the RL agent using improvements in code quality metrics, static tool confirmations (e.g PMD), and human readability indicators. Furthermore, we introduce an explainability module that translates refactoring actions into natural language suggestions for developers. The framework is implemented entirely in Python and validated using a curated dataset of real-world Java codebases. Results demonstrate enhanced detection accuracy and context-aware refactoring capabilities.

***Keywords*:** *Code Refactoring, Reinforcement learning, Code BERT, Abstract Syntax Tree(AST)*

Date of Submission: 25-08-2025 Date of acceptance: 03-09-2025

I. Introduction

Refactoring is the process of improving the overall design and structure of the code without changing its overall behavior. The goal of refactoring is to improve maintainability and facilitate future functionality extension, making it essential for adapting to ever-evolving software requirements. However, despite its benefits, many developers hesitate to refactor due to the time and effort involved [4].The code smell detection model identifies software issues that may cause severe problems in the future. Identifying code smells is always recommended, which helps to reduce the software maintenance cost and increases the code reusability.[3]Code smells, first introduced by Fowler, are indicators of sub-optimal code structures that can lead to software degradation if left unaddressed. These smells include duplicated code, long methods, and large classes, which contribute to increased maintenance effort and potential software defects [2].Automatic code generation refers to the process of writing code automatically through algorithms or programs.[5] By removing code smells the practice aims to improve maintainability, encapsulating quality attributes such as readability, flexibility, and testability[1].Code smells are parts of code that are not wrong but may add more complexity to code, which poses a technical debt and which is harder to maintain.[6].Code smells are one of the most accepted approaches to identifying design problems in the source code.[7]Overly complicated methods, duplicated code, and improper encapsulation are a few examples of code smells that impair flexibility and maintainability.

- *Long Method*: An approach that is difficult to comprehend and uphold because it is overly drawn out and takes on too many tasks.
- *Large Class*: A class that has too many duties, which goes against the Single Responsibility Principle and makes it challenging to oversee or grow.

- *Duplicate Code*: Code blocks that are repeated across the codebase raise maintenance costs and the possibility of inconsistent changes.
- *Inappropriate Intimacy*: When a class utilizes another class's methods or attributes excessively, it creates a tight coupling and less modularity.
- *Feature Envy*: A method that shows misplaced responsibilities by extensively relying on accessing the methods or attributes of another class rather than concentrating on its own.
- *Switch Statements*: Polymorphism could be used to improve extensibility and maintainability in favour of the overuse of if-else or switch structures.
- *Data Clumps*: For better organization, groups of data fields that commonly occur together could be enclosed into a separate class.

Software refactoring techniques are categorized into several types, including code-based refactoring, design refactoring, and architectural refactoring [1].A reinforcement learning (RL)-based framework is proposed for the detection and refactoring of multiple code smells, including Long Method, God Class, and Feature Envy, in Java programs.

II. Related Background Work

Code Smells

Code smells are indicators of potential design flaws in source code that may not directly cause bugs but degrade software quality, readability, and maintainability. Introduced by Fowler, code smells include patterns such as **Long Method**, **God Class**, **Feature Envy**, and **Duplicated Code**[2].Code smells are warning signs which refer to certain patterns or characteristics observed in source code that indicate potential design faws or violate basic design rules such as abstraction, hierarchy encapsulation, etc.[7]In this dataset, we identified and analyzed four code smells: *God Class*, *Long Method*, *Feature Envy*, and *Refused Bequest*. *Naive Bayes*, *Logistic Regression*, *Multilayer Perceptron*, *Decision Trees*, *K-Nearest Neighbors*, *Random Forest*, and *Gradient Boosting Machine*.[8]These smells often result from rapid development, poor modularization, or evolving requirements.

Refactoring

Refactoring is the process of restructuring existing code without altering its external behavior to improve internal structure. Common techniques include **Extract Method**, **Move Method**, **Inline Method**, and **Rename Variable**. Refactoring aims to reduce code complexity, enhance readability, and support maintainability. Manual refactoring is error-prone and time-consuming, motivating research into automated techniques.Refactoring is an important software development activity that employs various techniques to enhance the structure and quality of source code without altering its functionality.Extract method refactoring is one of the most commonly applied refactoring techniques that involves moving a coherent code fragment from a method into a new, aptly named method.[14]

Feature Envy

A method that is more interested in the data of another class(the envied class) than that of the class it is actually in. This anti-pattern represents a symptom of the method's misplacement, andis characterized by a lot of accesses to foreign attributes and methods.[13]

Reinforcement Learning (RL)

Reinforcement Learning (rl) is a branch of machine learning focused on training agents to take actions in an environment to maximize some notion of cumulative reward often involving a series of decisions.[15]*Reinforcement Learning* is a branch of machine learning where an agent learns optimal policies by interacting with an environment. The RL model is defined by a tuple (S, A, R, T, γ), where:

- **S**: Set of states (e.g., code representations like ASTs or token sequences),
- **A**: Set of actions (e.g., refactoring operations),
- **R**: Reward function quantifying code quality improvement,
- **T**: Transition function modeling state changes,
- **γ**: Discount factor for future rewards.

In the context of code smell detection and refactoring, the RL agent observes code states and learns policies to perform transformations that maximize software quality metrics (e.g., maintainability, modularity). Reinforcement Learning for Sequence Generation Reinforcement Learning is a branch of machine learning focused on training agents to take actions in an environment to maximize some notion of cumulative reward often involving a series of decisions .It uses a model known as the Markov Decision Process (MDP) , which deals with decision-making where each action is determined by steps, and outcomes are influenced by randomness. In RL, an agent (i.e., an autonomous entity that takes action in the given environment) improves its decisions through trial-and-error interactions with its environment, learning from the rewards it receives based on its actions.[1]

Code Representations

Effective RL depends on how source code is represented. Popular choices include:

- **Abstract Syntax Trees (ASTs):** Tree-based syntactic structures.
- **Control Flow Graphs (CFGs):** Represent execution paths.
- **Embeddings:** Learned vector representations from models like **CodeBERT**, **GraphCodeBERT**, or **Token2Vec**, capturing both syntax and semantics[16]

Resources to be searched

Selection of proper resources to search for relevant literature plays a significant role in an Research Work . We selected the following resources to search for all the available literature relevant to our research questions:

IEEE Xplore digital library (http://ieeexplore.ieee.org)
ACM digital library (https://dl.acm.org)
ScienceDirect (http://www.sciencedirect.com)
SpringerLink (https://link.springer.com)
Scopus (https://www.scopus.com)[10][3]

This review follows a structured methodology to identify, filter, and analyze research on reinforcement learning (RL)-based techniques for code smell detection and automated refactoring. The process was designed to ensure relevance, coverage, and reproducibility.

A comprehensive search was conducted across major databases: **IEEE Xplore**, **ACM Digital Library**, **SpringerLink**, and **Google Scholar**, focusing on works published between **2020 and 2025**. Keywords used included: *"reinforcement learning", "code smell detection", "automated refactoring", "software quality", "RL-based refactoring"*.

III. Review of Literature

The application of machine learning to software maintenance has evolved significantly, with reinforcement learning (RL) emerging as a powerful tool for code analysis and transformation. Early works applied **Q-Learning** to basic code smells such as *Long Method* and *Duplicated Code*, using syntactic structures like Abstract Syntax Trees (ASTs) to model states and refactoring actions as rewards-based transitions. These studies demonstrated the feasibility of framing code quality improvement as a Markov Decision Process (MDP).

Alabza et al. conducted a review on deep learning-based approaches for bad smell detection, and their focus was to summarise and synthesize the studies that used deep learning for bad smell detection. They collected and analyzed 67 studies until October 2022.[11].To identify the code smell, researchers leverage both manual and automatic detection approaches. Early detection for code smell are conducted manually.[12]

More recent studies have leveraged **Deep Reinforcement Learning (DRL)** algorithms, including **DQN**, **PPO**, and **A3C**, enabling agents to learn from complex, high-dimensional code representations. Models such as **CodeBERT**, **Code2Vec**, and **GraphCodeBERT** have been integrated to encode semantic and syntactic information. This fusion has improved the ability to detect smells like *God Class*, *Feature Envy*, and *Shotgun Surgery* with higher precision.

Several works propose **hybrid systems** that combine RL with static analysis or rule-based heuristics to balance learning efficiency with domain constraints. However, most are constrained to Java datasets and often lack generalization across languages or projects. Evaluation is typically based on code metrics (e.g., cohesion, coupling), reward values, or manually validated transformations.

This literature points to RL's growing maturity in software refactoring while revealing critical gaps in scalability, interpretability, and real-world applicability.

Comparison of Related Work with Proposed Research

No.	Authors & Year	Type	Uses RL	Smell Detection	Refactoring	Multi-Smell Support	Remarks
1	Palit& Sharma (2024)	Research Paper	Yes	No (assumes need for refactoring)	Extract Method (PPO)	No	Focus on single refactoring action using PPO; lacks integrated smell detection
2	Yadav et al. (2024)	Survey	No	Yes (ML techniques reviewed)	No	Yes	Extensive review of ML methods and datasets; no original RL

							contribution
3	Maini et al. (2024)	Research Paper	No	Yes (heuristic + metrics)	Optimized sequences	Partial	Optimization of refactoring order, but static, not learning-based
4	Xu et al. (2025, MANTRA)	Research Paper	No	No	Yes (via LLM + RAG)	Partial	Uses collaborative LLM agents, no learning or adaptive decision-making
5	Ye et al. (2025)	Research Paper	Yes	No	Guided generation	No	Combines procedural guidance with RL, but not used for code smell repair
6	Ali et al. (2025)	Research Paper	No	Yes (transformer-based)	No	Yes	Detects smells using transformer model; no refactoring action involved
7	Nasraldeen & Nehez (2024)	Research Paper	No	Yes (ML + data balancing)	No	Yes	Focus on improving detection accuracy, not on repair or refactoring
8	Cruz et al. (2025)	Research Paper	No	Yes (feedback-enhanced ML)	No	Yes	Continuous feedback loop improves ML detection, no automated correction
9	Skipina et al. (2023)	Research Paper	No	Yes (Feature Envy, Data Class)	No	No	Specific to certain smells, uses classical ML models
10	Azeem et al. (2019)	SLR & Meta-Analysis	No	Yes	No	Yes	Review and meta-analysis of ML models; no novel technique proposed
11	Proposed Method (2025)	Research Paper	Yes (PPO/DQN)	Yes (via metric analysis + RL states)	Yes (multi-action dynamic refactoring)	Yes	Fully integrated RL-based system that detects and resolves smells using adaptive feedback-driven refactorings

IV. Conclusion and Future Work

This review shows that reinforcement learning (RL) has strong potential to help with automatically finding and fixing code smells. Unlike traditional rule-based tools, RL methods can learn from experience and adapt to different situations in code. These approaches are especially useful for fixing common problems like long methods or feature envy. They often use deep learning with structured code formats like abstract syntax trees (ASTs) and code features from pretrained models.

Still, there are several challenges. It is hard to design reward functions that guide the RL agent properly, and it's difficult to make these models work well across many types of code. Many current methods also haven't been tested enough on real-world projects or large, complex systems.

After studying more than 10 important research papers and many ideas from past work, this paper proposes a new RL-based approach to solve these issues. The method combines automatic smell detection with dynamic, self-healing refactoring in one system. It uses software quality measures and testing to make sure the refactoring is correct. Unlike older studies that focus on single smells or static rules, this new approach learns and improves over time.

In the future, researchers should look into RL models that handle multiple goals (like performance and readability), combine symbolic and neural techniques for better understanding, and include developer feedback in the process. Making tools that work across programming languages and creating shared benchmark datasets will also help move this field forward.

Reference

[1]. Indranil Palit, Tushar Sharma. Generating refactored code accurately using reinforcement learning [2024] https://arxiv.org/abs/2412.18035

[2]. Pravin Singh Yadav, Rajwant Singh Rao, Alok Mishra and Manjari Gupta. Machine Learning-Based Methods for Code Smell Detection: A Survey[2024] https://www.theaspd.com/ijes.phpa

[3]. Ritika Maini, Navdeepkaur and Amardeepkaur. Optimized Refactoring Sequence for Object-Oriented Code Smells. [2024] https://www.mdpi.com/2076-3417/14/14/6149

[4]. Yisen Xu, Feng Lin, Jinqiu Yang, Tse-Hsun (Peter) Chen and Nikolaos Tsantalis. MANTRA: Enhancing Automated Method Level Refactoring with Contextual RAG and Multi-Agent LLM Collaboration. [27 MAR 2025] https://arxiv.org/abs/2503.14340

[5]. Yufan Ye, Ting Zhang ,Wenbin Jiang , Hua Huang. Process-Supervised Reinforcement Learning for Code Generation.[3 FEB2025] https://arxiv.org/abs/2502.01715

[6]. Israr Ali, Syed Sajjad, Hussian Rizvi and Syed Hasan Adil. Enhancing Software Quality with AI: A Transformer-Based Approach for Code Smell Detection [2025] https://www.mdpi.com/2076-3417/15/8/4559

[7]. Nasraldeen Anor adamkhleel, karoly Nehez. Improving Accuracy of Code Smells Detection using Machine Learning with Data Balancing Techniques [2024] https://doi.org/10.1007/s11227-024-06265-9

[8]. Daniel Cruz, Amanda Santana andEduardo Figueiredo.Evaluating a Continuous Feedback Strategy to Enhance Machine Learning Code Smell Detection [2025] https://www.sciencedirect.com/science/article/abs/pii/S0167642325000851

[9]. MilicaSkipina, JelenaSlivka and Nikolsluburic and Aleksandarkovacevic. Automatic Detection of Feature Envy and Data Class Code smells using Machine Learning [2023] https://www.sciencedirect.com/science/article/abs/pii/S0957417423033572

[10]. Muhammad IlyasAzeem, Fabio Palomba, Lin Shi and Qing Wang. Machine Learning techniques for Code Smell Detection: A systematic Literature Review and Meta Analysis[2019] https://www.sciencedirect.com/science/article/abs/pii/S0950584918302623

[11]. Amal Alazba, HamoudAljamaan, and Mohammad R. Alshayeb. 2023. Deep learning approaches for bad smell detection: a systematic literature review. Empirical Software Engineering 28 (2023). https://api.semanticscholar.org/CorpusID:258591793

[12]. Yang Zhang, Chuyan Ge, Haiyang Liu, and Kun Zheng. 2024. Code smell detection based on supervised learning models: A survey. Neurocomputing 565 (2024), 127014. https://doi.org/10.1016/j.neucom.2023.127014

[13]. Antoine Barbez, FoutseKhomh, and Yann-GaëlGuéhéneuc. 2019. A Machine-learning Based Ensemble Method For Anti-patterns Detection. ArXiv abs/1903.01899 (2019). https://api.semanticscholar.org/CorpusID:67877051

[14]. M. Fowler, P. Becker, K. Beck, J. Brant, W. Opdyke, and D. Roberts. 1999. Refactoring: Improving the Design of Existing Code. Addison-Wesley Professional.

[15]. Ashish Kumar Shakya, Gopinatha Pillai, and SohomChakrabarty. 2023. Reinforcement learning algorithms: A brief survey. Expert Systems with Applications 231 (2023), 120495.

[16]. Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang.

2024. Large Language Models for Software Engineering: A Systematic Literature Review. arXiv:2308.10620 [cs.SE] https://arxiv.org/abs/2308.10620