INTERNATIONAL JOURNAL OF NOVEL RESEARCH AND DEVELOPMENT (IJNRD)
© 2026 IJNRD | Volume 11, Issue 3, March 2026 | ISSN: 2456-4184 | IJNRD.ORG

AI-Driven Refactoring Advisor Using Code Smell Patterns

[1]Veera Venkata Subrahmanaya Varma Mudunuri
PG Student
Department of Computer Science and Engineering
JNTU University, Kakinada

Abstract : A significant aspect in the software development lifecycle is maintenance of the software, while refactoring working as a crucial component for enhancing code quality, readability, and manageability. Identifying appropriate refactoring opportunities is sometimes a difficult and time-intensive process that heavily depends on the developer's expertise. This study proposes an approach for recommending refactoring operations with machine learning, which leverages software metrics and code smell indicators for predictive analysis. The proposed approach consolidates a singular dataset by employing RefactoringMiner to identify refactoring instances in GitHub projects, the CK tool to obtain object-oriented code metrics, and static analysis tools to detect code smells. These attributes are employed to train many supervised learning models, including Decision Tree, Random Forest, Support Vector Machine, and Extreme Gradient Boosting (XGBoost). We employ standard performance metrics such as accuracy, precision, recall, and F1-score to evaluate the models. The XGBoost model has superior performance in experiments, with an accuracy of 94.27%, surpassing other algorithms. The findings indicate that machine learning can identify refactoring tendencies and assist engineers in enhancing their code. The proposed approach may serve as a foundation for intelligent refactoring recommendation systems in contemporary software development environments.

I. Introduction

Software systems are always changing because of new needs, bug corrections, and improvements in technology. It gets harder to keep code quality high as software gets bigger and more complicated. When code isn't structured well, it might be harder to maintain, cost more to maintain, and add to technical debt. Refactoring is a well-known method for changing the internal structure of software without changing how it works on the outside. Refactoring is when engineers change the structure of code to make it easier to read, modular, and maintain while keeping its functions.

Finding the right times to refactor can be hard, even though it has obvious benefits. Most of the time, developers use their knowledge and manual code examination to figure out when and where to refactor. This manual procedure takes a lot of time and is prone to mistakes in big software projects. Also, subjective judgment can cause different development teams to make different choices. Because of this, there is a rising need for automated tools that may help engineers find possible reworking opportunities based on observable code traits.

Software metrics and code smells are good signs that there are design flaws in software systems. Lines of Code (LOC), Weighted Methods per Class (WMC), Coupling Between Objects (CBO), and Lack of Cohesion in Methods (LCOM) are all examples of metrics that show how code is structured. Long Method, God Class, and Feature Envy are all examples of code smells that point out possible design problems. You can use these signs to find patterns that commonly come before refactoring work. It is now possible to gather historical refactoring data and look at how software metrics, code smells, and developer refactoring operations are related thanks to the availability of huge open-source repositories and mining tools.

Recent progress in machine learning has made it possible to use data-driven methods to solve software engineering difficulties. From past software data, machine learning models may learn about complicated relationships and guess what changes will happen to the code in the future. Machine learning can help suggest the best refactoring operations by looking at patterns in code metrics and reworking histories. These smart technologies can make it easier for developers to work, make software easier to maintain, and help with automated code quality improvement.

In this research, we present a machine learning-driven refactoring suggestion framework that forecasts appropriate refactoring operations utilizing software metrics and code smell indicators. The RefactoringMiner program finds refactoring instances in open-source repositories, and the CK tool finds structural code metrics. Static analysis tools give us code smell information, which we then combine with the metrics we got from the code to make a single dataset. We train and test several supervised machine learning

models, such as Decision Tree, Random Forest, Support Vector Machine, and Extreme Gradient Boosting (XGBoost), to find the best one for making predictions.

Experimental results show that ensemble learning methods, especially XGBoost, are better at finding refactoring opportunities. The suggested strategy uses data to find ways to improve code and can help developers keep software systems of high quality.

Here are the key points of this paper:

A single platform for building datasets that brings together refactoring histories, software metrics, and code smell indications from open-source sources.

A model for predicting refactoring based on machine learning that learns how code features and developer refactoring activities are related.

An assessment of various machine learning techniques to identify the optimal method for refactoring recommendations.

Empirical evidence illustrating the efficacy of the suggested framework, with XGBoost attaining the superior prediction performance.

The rest of this paper is set up like this. In Section II, we look at further work that has been done on refactoring detection and software maintenance that uses machine learning. Section III talks about the suggested method and how to make the dataset. The machine learning models utilized in the study are shown in Section IV. In Section V, we talk about the experimental setup and the measures used to judge the results. Section VI shows the findings and an appraisal of how well they worked. Finally, Section VII wraps up the paper and talks about possible ways for future research to go.

II. Related Work

Refactoring has been widely studied as an effective technique for improving software design quality and maintainability. The concept was formally introduced by Fowler [1], who defined refactoring as the process of restructuring existing code without changing its external behavior. Since then, numerous studies have explored automated techniques for identifying refactoring opportunities and improving software quality.

One major area of research focuses on code smell detection, which aims to identify design flaws that may indicate the need for refactoring. Code smells are structural symptoms in software that may negatively affect maintainability and readability. Marinescu [2] proposed detection strategies based on software metrics to identify design problems in object-oriented systems. Similarly, Lanza and Marinescu [3] emphasized the use of object-oriented metrics to characterize software design quality and detect problematic code structures. Studies such as those by Yamashita and Moonen [4] further investigated the relationship between code smells and software maintainability, showing that the presence of certain smells often correlates with increased maintenance effort.

Another important research direction involves mining software repositories to study refactoring activities. Tsantalis et al. [5] introduced RefactoringMiner, a tool capable of detecting refactoring operations from version control histories with high accuracy. By analyzing commit histories, researchers can identify patterns in how developers perform refactoring operations. Several empirical studies have used such tools to analyze refactoring trends in open-source projects and understand the relationship between code evolution and software quality.

With the growth of large-scale software repositories, researchers have increasingly explored machine learning approaches for software quality prediction and refactoring recommendation. Bavota et al. [6] investigated automated techniques to identify extract-class refactoring opportunities using structural and semantic metrics. Similarly, Palomba et al. [7] applied machine learning techniques to rank code smells and prioritize refactoring actions. These approaches demonstrate that data-driven models can effectively capture patterns associated with software design problems.

Machine learning algorithms have also been applied to various software engineering tasks such as defect prediction, maintainability assessment, and code quality analysis. Breiman [8] introduced the Random Forest algorithm, which has been widely used for classification tasks due to its robustness and ability to handle complex feature interactions. Cortes and Vapnik [9] proposed Support Vector Machines, which are effective in handling high-dimensional data and non-linear decision boundaries. More recently, Chen and Guestrin [10] developed the XGBoost algorithm, a gradient boosting framework known for its high predictive accuracy and scalability.

Despite these advances, many existing approaches either focus solely on detecting code smells or analyzing refactoring activities independently. There remains a need for integrated frameworks that combine software metrics, code smell detection, and historical refactoring data to improve prediction accuracy. The approach proposed in this paper addresses this gap by constructing a unified

dataset that integrates multiple sources of code quality information and applying machine learning models to predict appropriate refactoring actions.

III. Proposed Methodology

This section speaks about the suggested way to use machine learning to uncover ways to reorganize software. The framework brings together information from a number of sources, including code smells, software metrics, and past refactoring data. These parts are combined together to produce one dataset that is used to train and evaluate machine learning models that can recommend the best methods to change the structure of code. The proposed system's complete workflow consists of five main steps: selecting a repository, extracting refactoring, computing code metrics, identifying code smells, and generating predictions through machine learning. Figure 1 depicts how the proposed framework is put together.

A. Pick A Repository

We collected open-source Java repositories from GitHub to make a trustworthy dataset for the investigation. Open-source projects provide a lot of historical data on how code has evolved over time. This makes them useful for discovering signs of refactoring and software quality. The repositories that were picked had to meet the following standards: The project needs to be taken care of on a regular basis. The repository needs to have a long enough history of commits. The code structures in the project should be object-oriented so that they can be used for metric analysis. You can acquire both structural code metrics and information about historical refactorings from these repositories.

B. Removing refactoring

The RefactoringMiner tool removed refactoring actions from the commit histories of repositories. RefactoringMiner can automatically detect refactoring operations like Extract Method, Move Method, Rename Method, and Extract Class by looking at the differences between two commits that happened one after the other. The tool goes through the commit history of each repository and writes down the refactoring operations it discovers, as well as the classes and methods that were altered. These labels are what machine learning algorithms use to learn. The framework gathers real-world patterns that predictive models can learn from by looking at what developers have done in the past to alter code.

C. Getting Code Metrics

We used the CK tool, which is based on the Chidamber–Kemerer object-oriented metrics suite, to get software metrics that reflect the code's structure. These measurements look at several areas of how good and complicated a software design is. This study examines these primary metrics:

>LOC (Lines of Code): This tells you how many lines of code are in the class or method.
>WMC (Weighted Methods per Class): This tells you how hard it is to grasp a class.
>CBO (Coupling Between Objects): This shows how classes are related to each other.
>LCOM (Lack of Cohesion in Methods) tells you how well the methods of a class work together.
>DIT (Depth of Inheritance Tree): This shows how deep the inheritance tree is.
>RFC (Response for Class) counts the number of methods that could be called in response to a message. These data illustrate what design flaws need to be rectified.

D. Finding Code Smells

Code smells are signals that the design isn't good and that it's time to refactor. We employed static analysis techniques like PMD to discover code smells in this investigation. PMD looks at source code and discovers patterns that could make it hard to keep up with. This study investigates common code smells, including:

>Long Method
>Class of God
>Jealousy of Features
>Big Class

Each scent that is found is added to the dataset as a feature. These scent indicators work with the structural metrics to give further information about possible ways to change the code.

E. Making the Dataset

We pulled together the results from the previous steps to produce one dataset. Each instance in the dataset represents a class or method and has the following information:

>Software metrics that were removed
>Found indicators of bad coding
>Getting refactoring labels from RefactoringMiner

Cleaning the data, standardizing it, and encoding categorical features are all steps that the dataset goes through before the model is trained. After then, the dataset is divided into two parts: one for training and one for testing. This is done to see how well the model works.

F. Machine Learning-Based Prediction

The processed dataset is used to teach a lot of supervised machine learning models how to predict the right refactoring actions. This research examines the subsequent algorithms:

>Decision Tree
>Random Forest
>Support vector Machines
>XGBoost, or Extreme Gradient Boosting,

These models figure out how code smells, software measures, and historical refactoring operations are connected to each other. Once trained, the models can assist developers locate parts of software systems that could be better by predicting when code needs to be refactored.

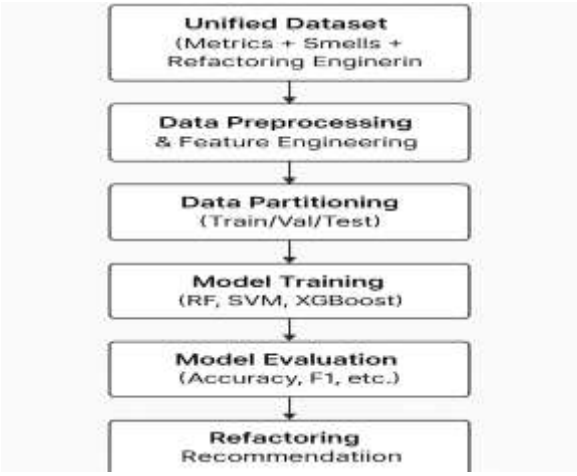

Fig 1: System Architecture

IV. Machine Learning Models

To predict suitable refactoring actions based on software metrics and code smell indicators, several supervised machine learning algorithms were evaluated in this study. These algorithms were selected because of their effectiveness in classification problems and their ability to capture complex relationships within software quality data. The models considered include Decision Tree, Random Forest, Support Vector Machine, and Extreme Gradient Boosting (XGBoost). Each model was trained using the constructed dataset and evaluated to determine its effectiveness in identifying potential refactoring opportunities.

A. Decision Tree

Decision Tree is a widely used classification algorithm that represents decision-making processes in the form of a tree structure. In this model, internal nodes represent feature-based decisions, branches represent possible outcomes of these decisions, and leaf nodes represent the predicted class labels. Decision Trees are particularly useful for interpreting relationships between features and target variables, as the resulting model can be visualized and easily understood.

In the context of this research, the Decision Tree model learns rules that associate specific software metric thresholds and code smell indicators with particular refactoring actions. Although Decision Trees are easy to interpret and implement, they can be prone to overfitting when trained on complex datasets.

B. Random Forest

Random Forest is an ensemble learning algorithm that combines multiple Decision Trees to improve predictive performance and reduce overfitting. Instead of relying on a single decision tree, Random Forest constructs a large number of trees using random subsets of the training data and feature space. The final prediction is determined through a majority voting mechanism across all trees in the forest.

This ensemble approach increases model robustness and helps capture complex relationships between software metrics and refactoring patterns. In this study, Random Forest provides improved accuracy compared to a single Decision Tree by reducing variance and improving generalization capability.

C. Support Vector Machine

Support Vector Machine (SVM) is a supervised learning algorithm that performs classification by finding an optimal hyperplane that separates data points belonging to different classes. The objective of SVM is to maximize the margin between the separating hyperplane and the nearest data points from each class, known as support vectors.

SVM is particularly effective in handling high-dimensional datasets and non-linear classification problems. By applying kernel functions, SVM can transform input data into higher-dimensional spaces where complex decision boundaries can be identified. In this research, SVM is used to model non-linear relationships between code metrics, code smells, and refactoring actions.

D. Extreme Gradient Boosting (XGBoost)

Extreme Gradient Boosting (XGBoost) is an advanced ensemble learning algorithm based on the gradient boosting framework. Unlike Random Forest, which builds trees independently, XGBoost constructs trees sequentially, where each new tree attempts to correct the prediction errors of the previous trees.

XGBoost incorporates regularization techniques to control model complexity and prevent overfitting, making it highly effective for structured datasets. It also supports parallel processing and optimized tree construction, allowing efficient handling of large datasets. In this study, XGBoost demonstrated the highest predictive accuracy among all evaluated models, highlighting its ability to capture complex patterns between software quality indicators and refactoring activities.

V. Experimental Setup

This section describes the experimental environment, dataset preparation process, and evaluation methodology used to assess the performance of the proposed refactoring recommendation framework.

A. Dataset Description

The dataset used in this study was constructed by mining open-source Java repositories hosted on GitHub. These repositories were selected based on their active development history and availability of sufficient commit records. Refactoring operations were extracted from the repository histories using the RefactoringMiner tool, which detects various types of refactoring actions by analyzing differences between consecutive commits.

In addition to refactoring information, structural software metrics were extracted using the CK tool, which computes object-oriented metrics at the class level. Furthermore, code smell indicators were detected using the PMD static analysis tool. The outputs from these tools were combined to form a unified dataset containing software metrics, code smell attributes, and corresponding refactoring labels. Each instance in the dataset represents a software class along with its associated features and refactoring type.

B. Data Preprocessing

Before training the machine learning models, several preprocessing steps were performed to ensure data quality and consistency. Missing or incomplete records were removed, and categorical attributes were encoded into numerical representations suitable for machine learning algorithms. Additionally, numerical features were normalized to ensure that differences in scale did not influence the learning process.

To ensure reliable model evaluation, the dataset was divided into three subsets: training, validation, and testing. Approximately 70% of the data was used for training, 15% for validation, and 15% for testing. This division ensures that the models are evaluated on unseen data and prevents overfitting.

C. Implementation Environment

The experiments were conducted using the Python programming language along with widely used machine learning libraries. Table I summarizes the experimental environment used in this study.

Table I: Experimental Environment

Parameter	Configuration
Programming Language	Python
Machine Learning Libraries	Scikit-learn, XGBoost
Data Processing Libraries	Pandas, NumPy
Software Metric Tool	CK Tool

Parameter	Configuration
Refactoring Detection Tool	RefactoringMiner
Code Smell Detection Tool	PMD
Operating System	Ubuntu / Windows Environment

These tools provide efficient capabilities for data processing, feature extraction, and model training.

D. Evaluation Metrics

To evaluate the performance of the machine learning models, several standard classification metrics were used. These metrics provide a comprehensive assessment of model accuracy and prediction quality.

- Accuracy: Measures the overall proportion of correct predictions made by the model.
- Precision: Indicates the proportion of correctly predicted refactoring instances among all predicted instances.
- Recall: Measures the proportion of actual refactoring instances that were correctly identified by the model.
- F1-Score: Represents the harmonic mean of precision and recall, providing a balanced evaluation of classification performance.

These evaluation metrics were used to compare the performance of the four machine learning algorithms and identify the most effective model for refactoring prediction.

VI. Results and Discussion

This section presents the performance evaluation of the machine learning models used in this study for predicting software refactoring opportunities. The algorithms evaluated include Decision Tree (DT), Random Forest (RF), Support Vector Machine (SVM), and Extreme Gradient Boosting (XGBoost). Each model was trained using the constructed dataset containing software metrics and code smell indicators and evaluated using the testing subset.

A. Model Performance Comparison

The performance of the four machine learning models was evaluated using accuracy as the primary metric. Table II presents the accuracy achieved by each model.

Table 2: Model Accuracy Comparison

Model	**Accuracy (%)**
Decision Tree	83.12
Random Forest	89.76
Support Vector Machine	90.42
XGBoost	**94.27**

The results show that the Decision Tree model achieved the lowest accuracy, primarily due to its tendency to overfit the training data when dealing with complex feature interactions. Although Decision Trees provide interpretable decision rules, they often struggle with datasets that contain non-linear relationships between features.

The Random Forest model improved performance significantly compared to the Decision Tree. By combining multiple decision trees through an ensemble approach, Random Forest reduces variance and improves generalization capability. This allowed the model to better capture relationships between software metrics and refactoring actions.

The Support Vector Machine model demonstrated slightly better performance than Random Forest. By using kernel functions, SVM can identify non-linear decision boundaries within the dataset. This capability enables SVM to effectively model complex relationships between code quality indicators and refactoring activities.

Among all evaluated models, XGBoost achieved the highest accuracy of 94.27%, outperforming the other algorithms. The superior performance of XGBoost can be attributed to its gradient boosting framework, which sequentially builds decision trees to correct

prediction errors from previous iterations. Additionally, the regularization mechanisms incorporated in XGBoost help prevent overfitting and improve model generalization.

B. Visualization of Model Performance

To provide a clearer comparison of the model performance, Fig. 2 presents a bar chart illustrating the accuracy values achieved by each algorithm. The visualization highlights the progressive improvement in performance from Decision Tree to ensemble-based methods such as Random Forest and XGBoost.

The chart clearly indicates that ensemble learning techniques provide better predictive capability for this problem domain. This is expected because ensemble models combine multiple weak learners to produce a stronger and more robust predictive model.

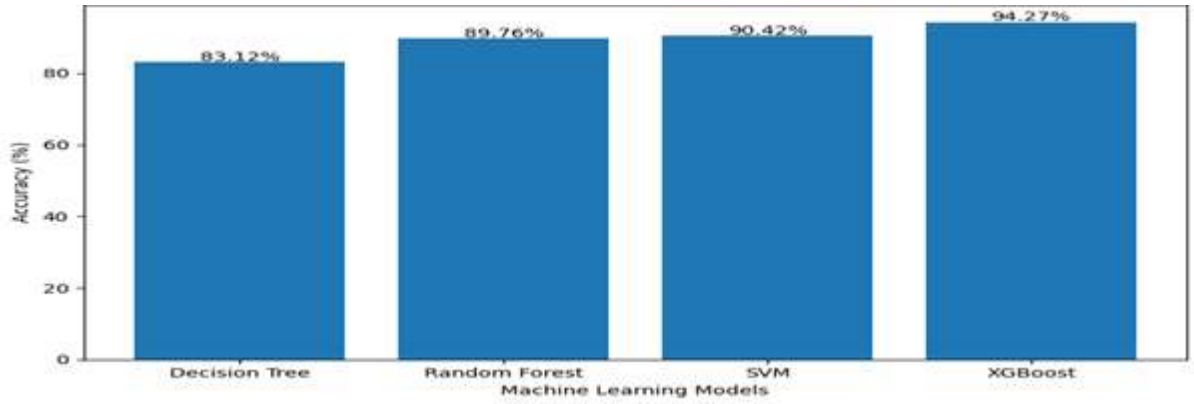

Fig 2: Accuracy Values of Machine Learning Models

C. Discussion of Results

The experimental results demonstrate that machine learning models can effectively learn patterns between software metrics, code smells, and developer refactoring actions. Metrics such as LOC, WMC, and CBO, combined with smell indicators such as Long Method and God Class, provide valuable signals for identifying potential refactoring opportunities.

The results also highlight the importance of using ensemble learning techniques for software quality prediction tasks. While simpler models such as Decision Trees provide interpretability, ensemble models such as Random Forest and XGBoost are better suited for capturing complex interactions between features.

Overall, the findings confirm that the proposed machine learning framework can accurately predict refactoring opportunities and assist developers in identifying areas of improvement in software systems.

VII. Conclusion and Future Work

This paper presented a machine learning-based framework for predicting software refactoring opportunities using software metrics and code smell indicators. The proposed approach integrates information from multiple sources, including refactoring history extracted from GitHub repositories, object-oriented software metrics computed using the CK tool, and code smell indicators detected through static analysis tools. These features were combined to construct a unified dataset that was used to train and evaluate multiple machine learning models.

Four supervised learning algorithms—Decision Tree, Random Forest, Support Vector Machine, and Extreme Gradient Boosting (XGBoost)—were evaluated to determine their effectiveness in predicting refactoring actions. Experimental results demonstrated that ensemble learning techniques outperform simpler models for this problem. Among the evaluated algorithms, XGBoost achieved the best performance with an accuracy of 94.27%, highlighting its ability to capture complex relationships between software quality indicators and refactoring activities.

The findings of this study indicate that machine learning techniques can effectively support developers in identifying potential refactoring opportunities. By analyzing software metrics and code smell patterns, the proposed framework can assist in improving software maintainability and reducing technical debt. Such intelligent systems have the potential to become valuable tools in modern software development environments, particularly when integrated into development workflows or software analysis platforms.

Despite the promising results, several opportunities remain for future research. Future work may involve expanding the dataset by including a larger number of repositories from diverse software domains. Additionally, integrating dynamic software metrics and runtime analysis could further enhance the prediction capability of the model. Another potential direction is the development of real-time refactoring recommendation systems that can be integrated directly into integrated development environments (IDEs) to provide developers with automated code improvement suggestions during the development process.

Overall, this research demonstrates the potential of machine learning techniques for supporting automated software maintenance and refactoring recommendation, providing a foundation for future advancements in intelligent software engineering tools.

References

[1] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA: Addison-Wesley, 1999.

[2] R. Marinescu, "Detection strategies: Metrics-based rules for detecting design flaws," in *Proc. IEEE Int. Conf. Software Maintenance*, 2004, pp. 350–359.

[3] M. Lanza and R. Marinescu, *Object-Oriented Metrics in Practice*. Berlin, Germany: Springer, 2006.

[4] A. Yamashita and L. Moonen, "Do code smells reflect important maintainability aspects?" in *Proc. IEEE Int. Conf. Software Maintenance*, 2013, pp. 306–315.

[5] N. Tsantalis, M. Mansouri, D. Mazinanian, and E. Digkas, "Accurate and efficient refactoring detection in commit history," in *Proc. 40th Int. Conf. Software Engineering (ICSE)*, 2018, pp. 483–494.

[6] G. Bavota, A. De Lucia, R. Oliveto, and M. Di Penta, "Identifying extract class refactoring opportunities using structural and semantic cohesion measures," *J. Systems and Software*, vol. 103, pp. 88–104, 2015.

[7] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, and A. De Lucia, "Detecting bad smells in source code using change history information," in *Proc. IEEE/ACM Int. Conf. Automated Software Engineering*, 2013, pp. 268–278.

[8] L. Breiman, "Random forests," *Machine Learning*, vol. 45, no. 1, pp. 5–32, 2001.

[9] C. Cortes and V. Vapnik, "Support-vector networks," *Machine Learning*, vol. 20, no. 3, pp. 273–297, 1995.

[10] T. Chen and C. Guestrin, "XGBoost: A scalable tree boosting system," in *Proc. ACM SIGKDD Int. Conf. Knowledge Discovery and Data Mining*, 2016, pp. 785–794.

[11] F. Pedregosa et al., "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[12] J. Han, M. Kamber, and J. Pei, *Data Mining: Concepts and Techniques*, 3rd ed. San Francisco, CA, USA: Morgan Kaufmann, 2012.

[13] S. McConnell, *Code Complete: A Practical Handbook of Software Construction*, 2nd ed. Redmond, WA, USA: Microsoft Press, 2004.

[14] W. Li and R. Shatnawi, "An empirical study of the bad smells and refactoring activities in open-source software," *Int. J. Software Engineering and Knowledge Engineering*, vol. 17, no. 1, pp. 1–25, 2007.

[15] M. Jureczko and L. Madeyski, "Towards identifying software project clusters with regard to defect prediction," in *Proc. Int. Conf. Predictive Models in Software Engineering*, 2010.

[16] F. Zhang, Y. Zou, A. Hassan, and B. Adams, "The relationship between code smells and refactoring practices," *Empirical Software Engineering*, vol. 19, no. 2, pp. 501–548, 2013.

[17] G. Bavota and B. Russo, "A machine learning approach to classify refactoring types," *IEEE Transactions on Software Engineering*, vol. 41, no. 9, pp. 892–912, 2015.

[18] J. Pantiuchina, G. Bavota, and B. Russo, "Predicting code refactoring using machine learning techniques," *Empirical Software Engineering*, vol. 26, pp. 1–32, 2021.

[19] T. Zhang and M. Kim, "Automated refactoring recommendation using machine learning," in *Proc. IEEE/ACM Int. Conf. Mining Software Repositories*, 2017.

[20] G. Bavota, M. Di Penta, R. Oliveto, and A. De Lucia, "An empirical study on the developers' perception of software coupling," in *Proc. IEEE Int. Conf. Software Maintenance*, 2012.

[21] Apache Software Foundation, "Apache Commons Lang," [Online]. Available: https://commons.apache.org/proper/commons-lang/

[22] SonarSource, "SonarQube documentation: Code quality and security analyzer," [Online]. Available: https://www.sonarqube.org/

[23] PMD Developers, "PMD source code analyzer," [Online]. Available: https://pmd.github.io/

[24] M. Aniche, "CK: Tool for calculating object-oriented metrics," [Online]. Available: https://github.com/mauricioaniche/ck

[25] N. Tsantalis, "RefactoringMiner: A tool for detecting refactorings in Git repositories," [Online]. Available: https://github.com/tsantalis/RefactoringMiner

[26] T. Mens and T. Tourwe, "A survey of software refactoring," *IEEE Transactions on Software Engineering*, vol. 30, no. 2, pp. 126–139, 2004.

[27] M. Fowler and K. Beck, "Refactoring: Improving the design of existing programs," *ACM SIGPLAN Notices*, vol. 35, no. 10, pp. 1–10, 2000.

[28] D. Dig and R. Johnson, "The role of refactorings in API evolution," in *Proc. Int. Conf. Software Maintenance*, 2006.

[29] M. Kim, T. Zimmermann, and N. Nagappan, "An empirical study of refactoring challenges and benefits at Microsoft," *IEEE Transactions on Software Engineering*, vol. 40, no. 7, pp. 633–649, 2014.

[30] G. Bavota, A. De Lucia, R. Oliveto, and M. Di Penta, "Mining refactoring opportunities in object-oriented systems," *IEEE Transactions on Software Engineering*, vol. 39, no. 5, pp. 701–718, 2013.

Copyright & License:

© Authors retain the copyright of this article. This work is published under the Creative Commons Attribution 4.0 International License (CC BY 4.0), permitting unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.