

An in-Depth Analysis for Source Code Refactoring System

Suraj Mahindrakar¹ and Dr. Prashant Kumbharkar²

¹⁻²Department of Computer Engineering, JSPM's, Rajarshi Shahu College of Engineering, Tathawade, Pune -33
Email: suraj.mahindrakar@gmail.com, pbk.rscoe@gmail.com

Abstract—Currently, software developers are tempted to incorporate online codes into their software as a result of the tremendous expansion of open source codes that are available online and the easy availability of assistance from search engines. The majority of the time, this is the cause of the code rotting, unhealthy dependencies between classes or packages, improper allocation of class responsibilities, overweight methods or classes, duplicated code, and a wide variety of other types of confusion and clutter of various kinds. Unwanted time and space complexity are brought about as a result of this, which ultimately causes the software to function at a level that is lower than what was expected. It is possible that this is a significant factor in software malfunctioning and other vulnerabilities, and it is the most serious challenge that the software industry is currently facing. A concept of software code refactoring is addressed by many authors by using of the several approaches that are analyzed in this research article. Most of the software code refactoring systems are based on the analysis of some parameters like comments, dead methods, data members and classes. But very few talks about the overall refactoring model. This research paper proposes a model to unleash the software code refactoring which analyzes all the basic and necessary parameters of complex Java project source code such as Data members, Member functions, classes, variables and comments and many more using the weighted graph technique to refactor the code thoroughly.

Index Terms— Source Code refactoring, Code smells, Dependency Graph, Feature Extraction , Decision making, Labelling.

I. INTRODUCTION

An experienced developer will likely utilize code refactoring more often than any other technique. One of the most important concepts in software development is refactoring, which involves reorganizing code or sections of code without altering the meaning or operation of the code itself. Because the changes are so tiny and insignificant, refactoring helps the developer avoid breaking the code's functionality and adding their own defects. Typically, it consists of a set of routine, fundamental steps.

Most of the code written by developers is typically somewhat sloppy, which is why reworking it is a regular practice. Which means it has a number of errors or a poor smell, both of which can make the code hard to update and maintain—something no developer wants. Both the code and the software life cycle are badly impacted. The discrepancies are usually not even acknowledged up until the inclusion of new features causes a malfunction. Time constraints and unrealistic, obviously unsustainable development cycles are the main causes of these anomalies. If you try to rush a project, you will end up with unmanageable, unclear code. Getting it fixed isn't

going to be easy; the developer will have to set aside time to modify the code, which will add even more pressure to an already tight schedule.

In most cases, when developers write code, they end up creating a self-dependent class that depends on another class they derived from. This causes a plethora of issues since cyclic dependencies are notoriously difficult to resolve and typically necessitate extensive workarounds. Some other sets of codes rely on these lines to work, so making large changes to them can cause a lot of trouble.

One class's output could affect the outcome of another when those classes are interdependent. Such a situation becomes problematic when the second one's output is critical for the class to resume normal operation. The former class can't do this since it relies on the latter class's output. These two classes are inseparable and dependent on each other, creating an endless loop that has no escape. Since refactoring fixes flawed or unclear code, it may provide some answers. Finding the code's dependencies and preventing closed or cyclic dependencies is one of the features of automatic refactoring tools. Because breaking the cycle of reliance can be quite difficult.

Because of this, the refactoring prevents the code from falling victim to cyclic dependencies. Similar to class dependencies, data member dependencies are also common. This happens when one data member depends on another.

This is a representation of the data members' movement from one function to another, which happens throughout the code. Under typical conditions, the data members rely on the other function for input. In a cyclic dependency on a function, it could work if the member functions are mutually exclusive, focused on each other, and execute a single action. As their dependency implies, the code won't be able to handle such case and lead to a catastrophic failure if the member functions are utilized elsewhere, which can be a headache to handle. Code restructuring is crucial for reducing these impacts. Data structures known as graphs allow for the representation of information through nodes. The process of creating a graph is not linear. Connected nodes represent a relationship between two pieces of data. Anyone can link any two nodes in a graph using these connecting lines, which are called edges.

In its description, Damian A. Tamburri et al. [1] More than ever before, software engineering need a community of practice. Balancing factors such as distance, culture, and worldwide engineering techniques are typically crucial to its success. Project costs or "social" debt could rise in this case due to a plethora of unanticipated socio-technical events, such as mass workforce turnover. Industrial research led us to the discovery of "community smells," or patterns of poor performance throughout a software development team's social and organizational structures, which serve as antecedents to these undesirable socio-technical incidents. In this paper, author present CODEFACE4SMELLS, an automated method for identifying four types of community smells. These types reflect socio-technical issues that have been detrimental to software engineering and organizational research, and understanding their impact is the first goal of this paper.

[2] Hui Liu et al. To identify code odors, author offer a method based on deep learning in this study. The method's core tenet is that sophisticated deep learning algorithms can pick out good characteristics for code scent detection and convert them to a binary prediction (smelly or not).author provide an automated method for producing labeled training data for code smell detection in order to maximize the potential of deep learning based code smell detection. The generated dataset is far bigger than conventional datasets produced manually, making the use of deep learning algorithms for code smell detection a realistic possibility. Feature envy, long methods, huge classes, and misplaced classes are the four types of code odors that author first apply the suggested approach to. Two sections make up the assessment. Author test it out on open-source apps that have inserted scents in the first section. Part two involves testing it on unaltered versions of open-source software's source code. From what author can tell from the evaluations conducted in both sections, the suggested method outperforms the current best practices when it comes to spotting feature envy, lengthy methods, big classes, and lost classes. It would be fascinating to see how the suggested method works for detecting other types of code smells in the future. Integrating the suggested method into popular integrated development environments (IDEs), particularly open-source ones, will be beneficial in the long run. One positive aspect of the integration is that it could help developers find places to refactor. And yet, it could show how well the suggested method works in practice.

[3] In this study, Sadi alawadi et al. observed... When it comes to improving data privacy, federated learning has come a long way as a distributed AI-based approach. However, there was little to no research that used federated learning to simultaneously detect code odors and preserve privacy when this study was being written. This paper's reported research has thus far investigated a privacy-aware technique, suggesting an important for organizations Federated Learning Code Smell Detection (FedCSD). Organizations can maintain data privacy while ensuring software quality using this notion, which mostly involves exchanging knowledge rather than data.

In this research, author showed how to use FL to train ML to find code smells in distinct organizations' code bases independently of each other. Author also provided a code review life cycle that incorporates author's approach, which has been improved. In addition, author used a number of datasets that focused on code smells in various companies; the results demonstrated improved accuracy in both the evaluation metrics and the global model for all companies. Because it can identify more global smells than the centralized model does on an individual basis, the FedCSD global model performs better than the cross-evaluation models.

[4] Researchers Tarek alkhair et al. Within the scope of this article, author sought to understand the effects of odors on the correlations between design patterns and flaws. Ten Java systems ranging in size from small to medium were part of author's investigation. According to the results, pattern classes are more likely to have defects than non-pattern classes. Since smelly pattern classes attract more faults than non-smelly pattern and non-pattern classes, it is possible that odors play a role as a contextual component in this relationship. The results of the study are three-fold: • Previous studies' findings were confirmed when author looked at the binary relationship between patterns and flaws. Author found that patterns are positively associated with the existence of faults. But when author account for odors as a confounding variable in this correlation, author find that only odorous patterns are positively associated with faults, whereas non-odorous patterns exhibited mixed findings. Pattern classes exhibit more faults than non-pattern classes, according to author's research. By factoring in odors, author find that classes with strong scents are more likely to have defects than those without, and that pattern classes, whether smelly or not, have a greater defect distribution than classes without patterns, though at different rates.

[5] Muhammad Anis Al Hilmi et al. For this study, author conducted a thorough review of the existing literature on code smells. From the 354 publications found during the search, author selected 69 primary studies that met author's predetermined inclusion and exclusion criteria. The studies chosen were published between 2013 and July 2022. In this study, the primary emphasis was on identifying code smells. These were primarily detected through the utilization of a machine learning approach, with some studies also employing deep learning techniques. Studies also explore the effects of code smells, refactoring, and the relationship between software metrics and security aspects. Author identify active and influential researchers in this field. Several detection methods have been identified, including competitive evolutionary search and multi-objective genetic programming. Studies also evaluate various detection tools. There are several challenges that need to be addressed in this study. These challenges include the need for a better understanding of subjective information, improving agreement between detectors, ensuring the applicability of the study to various contexts and datasets, enhancing developer knowledge, clarifying priorities, and establishing clear thresholds. It is evident that detecting code smells is a complex task, highlighting the necessity for advanced approaches and tools.

This literature review's structure can be broken down into the following components: The second section, which is a review of the literature, looks at previous research; the third section outlines the approach that is advised; and the fourth section offers suggestions for additional research.

II. LITERATURE SURVEY

[6] The authors of this study are Fabio Palomba et al. The organizational and technical structures of software are intricately linked . It is possible that the debts present in both structures, social and technical debt, may be closely intertwined. In this paper, author delve into the connection between social and technical debt, specifically focusing on code and community smells. In this paper, author present a mixed-method empirical convergence evaluation to provide evidence of the correlation between these two phenomena in software engineering. Previous work has already offered some evidence of this correlation . In a recent survey, author found that community-related issues, such as the presence of community smells, are seen as signs of code smells. This suggests that there are other factors that affect the maintainability of technical implementation flaws.

[7] Albuquerque, Danyllo, et al. author conducted an empirical evaluation of ID and NID approaches for code smell detection using experimental tasks completed by 16 participants with varying degrees of experience. Developers utilizing ID improved the precision values by around 15%, assisting in the detection of more code smells, in relation to the effectiveness results (RQ1). Additionally, the results demonstrated that ID enhances code scent detection recall values by approximately 60%, which helps to prevent code odors from going unnoticed. By facilitating a more thorough investigation of scent occurrences and offering real-time feedback, the ID approach enhances the efficacy of code smell detection. It was also noted that the results linked to these measures could be directly impacted by the participants' level of job experience. Values for precision and recall measures were found to be higher as the individuals' employment experience increased.

[8] Akira Mori et al. author used phylogenetic methods to shed light on software evolution and maintenance in this article. Using these updated versions, author expanded on the research of Hashimoto and Mori on earlier Emacs versions. Applying a new reticulate evolution approach allowed us to spot ongoing work on multilingual improvements. In addition, author detailed a novel phylogenetic experiment that shed light on the early stages of development of the open-source 3D printer firmware, a period when no records of its history were available. One benefit of the distance-based approach used in the paper is that an AST differencing tool automatically computes the distance matrix, so there's no need to manually inspect the code. For preliminary data collection on the software's evolutionary features, the approach is useful. By conducting tests on an artificial SPL named Argo UML-SPL and a real-world example of an automotive real-time operating system called TOPPERS/ATK2, author explored the possible applications of split networks computed using the Neighbor-Net approach.

[9] Abdullah almogahed et al. A comprehensive analysis of the components that produce varied effects of refactoring strategies on quality attributes is presented in this paper, which gives the results of an experimental study. Possible uses of refactoring techniques have been considered, researched, and evaluated. Seven separate case studies have made use of ten different refactoring strategies. The ten refactoring techniques—Extract Method, Move Method, Introduce Parameter Object, Remove Setting Method, Pull Up Field, Pull Up Method, Push Down Field, Push Down Method, Extract Subclass, and Extract Interface—are all capable of producing different effects, and the scenarios in which they are used greatly influence this. The quality qualities can be positively, negatively, or left unchanged depending on the scenario you choose for each of these refactoring. Author determined the optimal use cases for each of the ten refactoring strategies by conducting a multi-case analysis.

[10] Vahid Alizadeh et al. In this study, author presented a software refactoring recommendation tool that developers may interact with. The tool would adapt to their feedback and code modifications, then offer refactoring to them in real time. Developers can take use of search-based refactoring tools using author's interactive method, even if they don't have a clue about optimization or multi-objective optimization algorithms. Interactions with developers actually carry out the implicit study of non-dominated refactoring options. To narrow the search field and find better options, author use the developer feedback. A group of software engineers participated in a human study to assess author's tool's efficacy; they compared it to cutting-edge refactoring methods and provided feedback on its performance. In addition to suggesting a new approach for software engineers to interactively modify their systems, author's evaluation results show that author's tool makes software restructuring more applicable. In order to draw conclusions regarding the methodology's generalizability, future work will include validating author's approach with more types of refactoring, programming languages, and code smells.

[11] The study conducted by Musaad alzahrani et al. The paper presents a cohesion metric that is designed to support the Extract Class refactoring. The metric was evaluated using cohesion properties and compared with other cohesion metrics based on classes from three open source systems. Furthermore, the paper presented a refactoring approach that utilized the recently introduced cohesion metric to automatically detect Extract Class refactoring possibilities. An analysis of actual classes from an open source system demonstrated the potential value of the suggested refactoring approach.

[12] Gupta, Ruchin, et al. In this paper, author have developed a tool to identify temporary field code smell, which has not been addressed by any existing tools. Additionally, author have created a separate branch for the "Designate Java" and added the code for temporary field detection. Finally, author have submitted this branch to the original repository for the benefit of the entire community. Author have conducted tests on the TFfinder tool using eight open-source JAVA projects. The results showed that there is a presence of temporary field code smell in all the projects studied, but it is at a very low percentage, ranging from 3% to 22%. In addition, the percentage of temporary field smells varied between 60% and 96% when considering non-cohesive classes of projects. It is determined that classes lacking cohesion are the cause of temporary field odor. While the tool has only been tested on 8 projects, TFfinder can be used to test any number of projects. In the future, author have plans to test the tool on a significant number of projects.

[13] The authors of this paper are Soumaya Rebai et al. In this paper, author have introduced a fresh approach to facilitate interactive refactoring by merging the investigation of quality enhancements (objective space) and refactoring locations (decision space). author's approach enabled developers to efficiently explore the Pareto front of refactoring solutions through the use of multi-objective search. By clustering the decision space, developers were able to pinpoint diverse refactoring solutions within the same cluster in the objective space. This led to improvements in desired quality attributes. As part of author's evaluation process, author enlisted the help of human subjects to assess the effectiveness of author's tool. They were tasked with evaluating the tool and

comparing it to the latest refactoring techniques available. author's evaluation results demonstrate that the combination of decision and objective spaces enabled developers to efficiently communicate their preferences and reach appropriate refactorings that aligned with their expectations.

According to a source, [14] Iman Hemati Moghadam et al. In this paper, author explored the subject of detecting refactorings implemented between two program versions. Author have developed a unique algorithm for detecting refactoring that is not limited to any specific programming language. One way to achieve this is by representing the input programs as a sequence of characters and abstracting away the specific details of each programming language. Additionally, author can rely on entities' names and their relationships to match entities that have changed between the original and refactored programs. The effectiveness of the implemented tool, known as RefDetect, is showcased by applying it to various Java and C++ applications. Based on author's extensive research, RefDetect stands out as a groundbreaking language-neutral refactoring detection tool that has achieved an exceptional level of accuracy.

[15] According to a source, the information provided is supported by evidence. Simone Romano et al. Before delving into the multi-study investigation presented in this paper, author conducted a preliminary mining study to provide the necessary motivation. This comprehensive study aims to address several key questions: the factors influencing the introduction of dead code by developers, their attitudes and strategies for dealing with it, and the potential impact of dead code on software. Author conducted semi structured interviews with professional developers, followed by three off-line experiments and one on-line experiment involving graduate and undergraduate students.

[16] Authors Kyaw Ei Swe et al. describes the bug localization process, the developer utilizes the bug information to make necessary modifications to the source code. Fixing the source file with the bug report can be quite time-consuming if done manually. Numerous researchers have proposed various bug localization tools using different approaches. IR-based bug localization approaches are preferred due to their low computation cost among the various bug localization methods. In this paper, author present a bug localization approach aimed at efficiently locating relevant source code files. author's approach utilizes the structures of source code files and bug reports, which are combined with previously fixed bug reports. Conducting an analysis on three open source projects that consist of over three thousand bug reports. author's approach has been proven to be highly effective in bug localization, as evidenced by the evaluation results on these projects.

According to a source, [17] Naifu Zhang et al. This paper presents a distributed source coding framework for model aggregation in distributed learning. This coding framework has been shown to achieve the lowest possible communication cost for distributed learning. Author developed a sophisticated coding scheme using LDPC codes that can be utilized with existing quantization methods, taking into account gradient statistics. Through a semi-analytical Monte Carlo simulation with low computation complexity, author were able to calculate the statistics of quantization variables based on the gradient statistics. Author proposed an adaptive SW coding scheme that estimates the gradient statistics using the observed quantized gradients at the parameter server. The LDPC codes are then dynamically adjusted in each iteration based on the estimation results. The experiment results demonstrate that the coding scheme effectively minimizes communication costs while maintaining model accuracy.

[18] Aakanshi Gupta et al. This study applies Python code scent detection algorithms that use multinomial machine learning. With an 8.01 severity level, the Cognitive Complexity smell is the most serious of the five code smells. The other four odors all fall into the moderate range, with an average intensity of 7. The J48 algorithm achieved an accuracy rate of 92.8% while comparing the performance of multinomial classifiers with AdaBoost and testing the predicted class level severity intensity for each code smell.

[19] Youjia Li Ect.author have improved the model's generation performance through optimization and debugging, and author have built an online code generating service using ChatGPT and PE. In addition, authorthoroughly analyzed and evaluated the resulting code from several angles, proving that this method is effective. author's research shows that ChatGPT can effectively generate source code with good prompts, which is great news for developers and researchers looking for solutions. Still, when it comes to results, ChatGPT and generative AI are just supplementary tools that can help users do tasks, but the real magic happens when humans put their heads together.

[20] X. Yan et al. in this paper describes the text of the bug report is used by the software project's developers and testers to alter the relevant buggy programs. The locating process can be time-consuming for complex and big project codes, though. So, it's crucial to find ways to make software developers spend less time and money using the bug localization approach to automatically find the files that are defective. Using syntactical and semantic information from source files, this research proposes a methodology for software bug localization. One approach is the CRSS technique of source file representation, which captures lexical and syntactic knowledge at

the sentence level as well as code naturalness. Next, author examine the relationship between the source file and the bug report from a textual, semantic, and syntaxual perspective.

III. PROPOSED MODEL

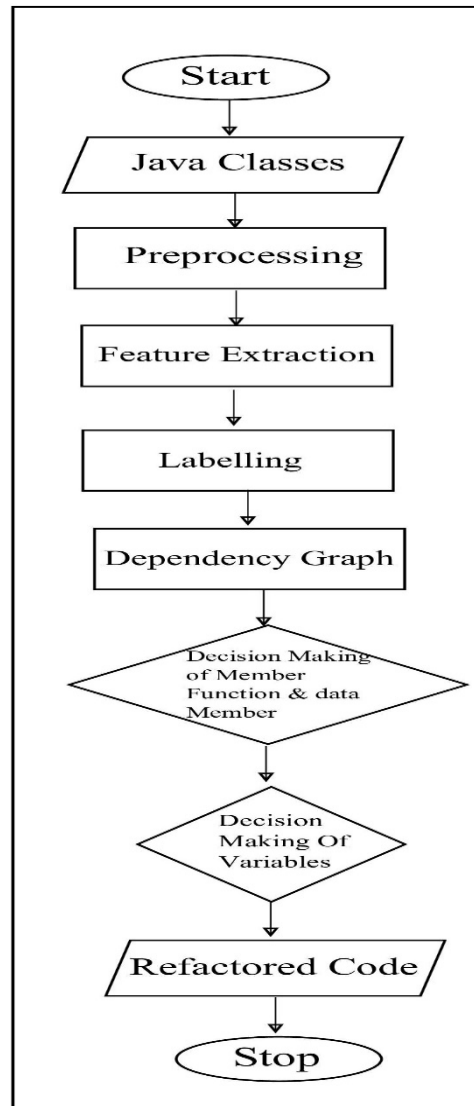


Figure 1: Proposed model

IV. CONCLUSION AND FUTURE SCOPE

As a result of the availability of easy access to source code from a variety of sources, such as Git hub and Stack overflow, and many others, software developers from all over the world use the source code from these resources in their projects. During the process of constructing modules, these codes are frequently utilized for testing and integration purposes. In the process, they simply leave the codes that are not required, which may have form of functions, data members, variables, classes, and objects that are not being utilized. It is possible that the compilation process will have an effect on the software programs, or that a problem may be created in the future, which will cause the applications to function less effectively. Therefore, in order to address these issues, a technique known as source code refactoring is utilized. This approach finally detects the faults in the code, which manifest themselves as classes, objects, functions, and data members that are not being utilized. During the process of refactoring source code, these anomalies are meticulously identified and removed in such a way that

they will not have any impact on the final source code. Additionally, the process improves the working of the source code by ensuring that the time and space complexity is maintained in the appropriate manner.

In the future, this study effort can be improved to work with a wide variety of characteristics, including package dependencies, inherited attributes, recursion functions, threads, and a great deal more, in order to strengthen the concept of source code reworking.

REFERENCES

- [1] D. A. Tamburri, F. Palomba and R. Kazman, "Exploring Community Smells in Open-Source: An Automated Approach," in *IEEE Transactions on Software Engineering*, vol. 47, no. 3, pp. 630-652, 1 March 2021, doi: 10.1109/TSE.2019.2901490.
- [2] H. Liu, J. Jin, Z. Xu, Y. Zou, Y. Bu and L. Zhang, "Deep Learning Based Code Smell Detection," in *IEEE Transactions on Software Engineering*, vol. 47, no. 9, pp. 1811-1837, 1 Sept. 2021, doi: 10.1109/TSE.2019.2936376.
- [3] S. Alawadi et al., "FedCSD: A Federated Learning Based Approach for Code-Smell Detection," in *IEEE Access*, vol. 12, pp. 44888-44904, 2024, doi: 10.1109/ACCESS.2024.3380167.
- [4] T. Alkhaier and B. Walter, "The Effect of Code Smells on the Relationship Between Design Patterns and Defects," in *IEEE Access*, vol. 9, pp. 3360-3373, 2021, doi: 10.1109/ACCESS.2020.3047870.
- [5] M. A. A. Hilmi, A. Puspaningrum, Darsih, D. O. Siahaan, H. S. Samosir and A. S. Rahma, "Research Trends, Detection Methods, Practices, and Challenges in Code Smell: SLR," in *IEEE Access*, vol. 11, pp. 129536-129551, 2023, doi: 10.1109/ACCESS.2023.3334258.
- [6] F. Palomba, D. Andrew Tamburri, F. Arcelli Fontana, R. Oliveto, A. Zaidman and A. Serebrenik, "Beyond Technical Aspects: How Do Community Smells Influence the Intensity of Code Smells?," in *IEEE Transactions on Software Engineering*, vol. 47, no. 1, pp. 108-129, 1 Jan. 2021, doi: 10.1109/TSE.2018.2883603.
- [7] D. Albuquerque et al., "On the Assessment of Interactive Detection of Code Smells in Practice: A Controlled Experiment," in *IEEE Access*, vol. 11, pp. 84589-84606, 2023, doi: 10.1109/ACCESS.2023.3302260.
- [8] A. Mori and M. Hashimoto, "Phylogenetic Analysis of Reticulate Software Evolution," 2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR), Melbourne, Australia, 2023, pp. 498-510, doi: 10.1109/MSR59073.2023.00074.
- [9] Almogahed, M. Omar, N. H. Zakaria, G. Muhammad and S. A. AlQahtani, "Revisiting Scenarios of Using Refactoring Techniques to Improve Software Systems Quality," in *IEEE Access*, vol. 11, pp. 28800-28819, 2023, doi: 10.1109/ACCESS.2022.3218007.
- [10] V. Alizadeh, M. Kessentini, M. W. Mkaouer, M. Ó Cinnéide, A. Ouni and Y. Cai, "An Interactive and Dynamic Search-Based Approach to Software Refactoring Recommendations," in *IEEE Transactions on Software Engineering*, vol. 46, no. 9, pp. 932-961, 1 Sept. 2020, doi: 10.1109/TSE.2018.2872711.
- [11] M. Alzahrani, "Measuring Class Cohesion Based on Client Similarities Between Method Pairs: An Improved Approach That Supports Refactoring," in *IEEE Access*, vol. 8, pp. 227901-227914, 2020, doi: 10.1109/ACCESS.2020.3046109.
- [12] R. Gupta and S. K. Singh, "TFfinder: A Software tool to discover Temporary Field code smell," 2020 2nd International Conference on Advances in Computing, Communication Control and Networking (ICACCCN), Greater Noida, India, 2020, pp. 910-914, doi: 10.1109/ICACCCN51052.2020.9362950.
- [13] S. Rebai, V. Alizadeh, M. Kessentini, H. Fehri and R. Kazman, "Enabling Decision and Objective Space Exploration for Interactive Multi-Objective Refactoring," in *IEEE Transactions on Software Engineering*, vol. 48, no. 5, pp. 1560-1578, 1 May 2022, doi: 10.1109/TSE.2020.3024814.
- [14] H. Moghadam, M. Ó. Cinnéide, F. Zarepour and M. A. Jahanmir, "RefDetect: A Multi-Language Refactoring Detection Tool Based on String Alignment," in *IEEE Access*, vol. 9, pp. 86698-86727, 2021, doi: 10.1109/ACCESS.2021.3086689.
- [15] S. Romano, C. Vendome, G. Scanniello and D. Poshyvanyk, "A Multi-Study Investigation into Dead Code," in *IEEE Transactions on Software Engineering*, vol. 46, no. 1, pp. 71-99, 1 Jan. 2020, doi: 10.1109/TSE.2018.2842781.
- [16] E. E. Swe and H. M. Oo, "Source Code Retrieval for Bug Localization using Bug Report," 2019 IEEE 15th International Conference on Intelligent Computer Communication and Processing (ICCP), Cluj-Napoca, Romania, 2019, pp. 241-247, doi: 10.1109/ICCP48234.2019.8959535.
- [17] N. Zhang and M. Tao, "A Low-Complexity and Adaptive Distributed Source Coding Design for Model Aggregation in Distributed Learning," in *IEEE Open Journal of the Communications Society*, vol. 3, pp. 2444-2460, 2022, doi: 10.1109/OJCOMS.2022.3228813.
- [18] A. Gupta, R. Gandhi, N. Jatana, D. Jatain, S. K. Panda and J. V. N. Ramesh, "A Severity Assessment of Python Code Smells," in *IEEE Access*, vol. 11, pp. 119146-119160, 2023, doi: 10.1109/ACCESS.2023.3327553.
- [19] Y. Li, J. Shi and Z. Zhang, "An Approach for Rapid Source Code Development Based on ChatGPT and Prompt Engineering," in *IEEE Access*, vol. 12, pp. 53074-53087, 2024, doi: 10.1109/ACCESS.2024.3385682.
- [20] X. Yan, S. Cheng and L. Guo, "Bug localization based on syntactical and semantic information of source code," in *Journal of Systems Engineering and Electronics*, vol. 34, no. 1, pp. 236-246, February 2023, doi: 10.23919/JSEE.2023.000010.