ESP-JETA
ESP Journal of Engineering & Technology Advancements
ISSN: 2583-2646 / Volume 1 Issue 1, September, 2021 / Page No: 244-252
Paper Id: JETA-V1I1P126 / Doi: 10.56472/25832646/JETA-V1I1P126

Original Article

Improving Legacy Software Quality through AI-Driven Code Smell Detection

Abhinav Balasubramanian
Independent Researcher, USA.

Received Date: 18 June 2021 Revised Date: 21 July 2021 Accepted Date: 24 August 2021

Abstract: *Legacy systems are often plagued by code smells - indicators of poor design and implementation choices - that compromise software quality and increase technical debt. Addressing these issues is essential for ensuring system maintainability and long-term health. Traditional static code analysis tools, while widely used, are prone to generating false positives and require considerable manual effort, making them less suitable for large, complex codebases.This paper proposes a conceptual framework for using machine learning to detect code smells efficiently and accurately. The framework leverages static code metrics, such as cyclomatic complexity, method length, and coupling, as features for supervised learning models like decision trees and gradient boosting. By combining software metrics with machine learning, this approach aims to improve detection precision and reduce the burden of manual review.The paper also discusses how such a framework could be integrated into development environments to provide developers with actionable insights for refactoring. This proposal highlights the potential of machine learning to support software quality improvement efforts, particularly in the context of legacy systems.*

Keywords: *Artificial Intelligence (AI), Code Smell Detection, Software Maintainability, Software Quality Improvement, Machine Learning in Software Engineering.*

I. INTRODUCTION

Software systems evolve over time, and as they age, they often accumulate technical debt in the form of code smells—indicators of poor design or implementation practices. These smells, such as large classes, long methods, and high coupling, degrade software quality, increase maintenance costs, and reduce overall system reliability. Addressing code smells is particularly critical in legacy systems, which form the backbone of many organizations but are often characterized by outdated designs, complex interdependencies, and limited documentation.

Traditional static code analysis tools are widely used to detect code smells, but they come with significant limitations. While these tools provide quick assessments based on predefined rules, they are prone to generating false positives and often require considerable manual review, which can be both time-consuming and error-prone. Additionally, static tools lack the adaptability needed to handle the nuanced and diverse nature of legacy systems.

To address these challenges, this paper proposes a conceptual framework for leveraging machine learning (ML) to detect code smells in legacy systems more efficiently and accurately. Unlike traditional tools, ML models can learn from labeled datasets of code smells and identify patterns that are difficult to encode in rule-based systems. By analyzing static code metrics such as cyclomatic complexity, method length, and coupling, ML can provide developers with actionable insights tailored to their specific codebase.

This paper also explores the potential integration of such a framework into development environments, enabling real-time feedback and facilitating proactive refactoring efforts. By bridging the gap between code smell detection and developer action, this approach aims to enhance maintainability and reduce the technical debt associated with legacy systems. Thus this paper aims to address the persistent challenges of legacy system maintenance, offering a fresh perspective on leveraging AI in software engineering.

II. RELATED WORK

A. Traditional Approaches

Static code analysis tools have long been employed to detect code smells, utilizing predefined heuristic rules to flag potential design flaws. These tools have proven effective in identifying common issues such as "God Class" or "Long Method." However, a significant limitation lies in their propensity to generate false positives, necessitating extensive manual validation.

Comparative evaluations of static analysis tools for languages like C/C++ [1][2] highlight their effectiveness in detecting certain categories of flaws but also underscore their difficulty in addressing more nuanced and context-dependent code smells. Furthermore, domain-specific approaches, such as the one proposed in [3], advocate for custom rule sets tailored to specific domains. While these strategies improve detection for niche scenarios, they often face challenges in scalability, particularly when applied to large, complex systems.

B. Machine Learning in Software Engineering

In recent years, machine learning (ML) has emerged as a promising alternative to overcome the limitations of traditional static analysis. ML-based methods leverage software metrics to train models capable of detecting code smells with greater precision. For example, research in [4] demonstrates the use of classifiers and ensemble methods, achieving improved detection rates for smells like "Feature Envy" and "Large Class." Similarly, [5] highlights high-performing algorithms that exhibit significant accuracy in detecting multiple code smells across diverse systems. Multi-label classification techniques, as introduced in [6], further enhance detection capabilities by addressing the co-occurrence of smells, allowing for the identification of overlapping design flaws. These advancements show potential not only for improving detection accuracy but also for reducing manual validation efforts by providing actionable insights for refactoring.

C. Research Gap

Despite the progress achieved by ML-based approaches, notable gaps remain. While studies such as [4] and [5] report impressive detection accuracies, their reliance on balanced and curated datasets limits their generalizability to real-world scenarios, particularly those involving imbalanced and complex legacy systems. Moreover, existing efforts like [7], which aim to quantify the severity of code smells, do not incorporate machine learning to enhance adaptability and scalability. This gap is particularly evident in the context of legacy systems, which pose unique challenges, including outdated dependencies, intricate interdependencies, and incomplete documentation. Addressing these issues with tailored AI-driven approaches is essential to improving the maintainability of legacy systems and effectively mitigating technical debt.

III. INDICATORS OF CODE SMELL

Code smells are symptoms of deeper design or implementation issues in the codebase that compromise software quality and maintainability. They are not bugs but indicators of poor coding practices that may lead to technical debt and increased maintenance costs.

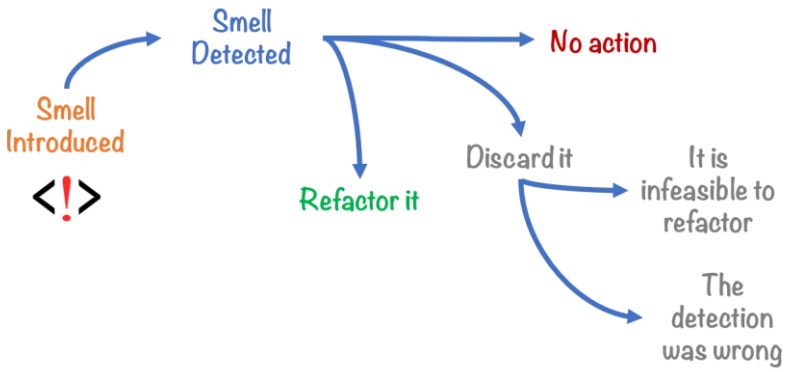

Figure 1: Life Cycle of a Smell [8]

Below, we discuss some common types of code smells, grouped by their characteristics:

A. Class-Level Smells

These smells indicate issues with the design or structure of classes:

a) God Class:

- A class that handles too many responsibilities, violating the Single Responsibility Principle.
- Indicators: Large size, high coupling, and low cohesion.
- Impact: Difficult to maintain and test, with frequent ripple effects when changes are made.

b) Lazy Class:

- A class that does too little to justify its existence.
- Indicators: Low number of methods or properties.

- Impact: Adds unnecessary complexity and overhead to the codebase.

c) High Coupling:

- A class that is overly dependent on other classes.
- Indicators: Excessive dependencies and frequent interactions with multiple classes.
- Impact: Reduces modularity and makes the system harder to modify.

B. Method-Level Smells

These smells highlight issues within individual methods or functions:

a) Long Method:

- A method with an excessive number of lines of code.
- Indicators: High cyclomatic complexity and multiple nested loops or conditionals.
- Impact: Harder to understand, maintain, and reuse.

b) Feature Envy:

- A method that excessively uses data or functionality from another class.
- Indicators: Frequent access to properties or methods of a foreign class.
- Impact: Violates encapsulation, increasing coupling and reducing maintainability.

c) Too Many Parameters:

- A method that accepts an excessive number of arguments.
- Indicators: Methods with more than 3-4 parameters.
- Impact: Reduces readability and increases the likelihood of errors.

C. Structural and Design Smells

These smells affect the overall structure of the codebase:

a) Divergent Change:

- A class that needs to be modified for multiple, unrelated reasons.
- Indicators: Frequent changes to the same class for different functionalities.
- Impact: Violates the Open/Closed Principle and increases maintenance complexity.

b) Shotgun Surgery:

- A single change in functionality that requires modifications across multiple classes.
- Indicators: High dependency spread for simple changes.
- Impact: Increases the risk of errors and reduces system cohesion.

c) Duplicated Code:

- Identical or similar blocks of code appear in multiple places.
- Indicators: Repetition of logic across the codebase.
- Impact: Increases maintenance effort and the risk of inconsistencies.

D. Object-Oriented Smells

These smells are specific to object-oriented design principles:

a) Data Class:

- A class that contains only fields and accessors without behavior.
- Indicators: Lack of methods or reliance on other classes for operations.
- Impact: Violates encapsulation and adds unnecessary dependencies.

b) Inappropriate Intimacy:

- Classes that is too familiar with each other's implementation details.
- Indicators: Frequent access to private members of another class.
- Impact: Increases coupling and reduces modularity.

E. Architectural Smells

These smells affect the higher-level design of the application:

a) Cyclic Dependencies:

- Classes or modules that depend on each other in a circular manner.
- Indicators: Cyclic graphs in the dependency structure.
- Impact: Complicates system understanding and refactoring.

b) Deep Inheritance:

- A class hierarchy that is too deep.
- Indicators: Inheritance chains exceeding 3-4 levels.
- Impact: Reduces readability and increases the complexity of modifications.

Thus Code smells act as "red flags" that indicate areas of the code that are likely to cause problems in the future. Addressing these smells proactively through refactoring improves:

- Maintainability: Reduces the effort needed to understand and modify the code.
- Scalability: Simplifies the addition of new features without breaking existing functionality.
- Code Quality: Enhances readability, reduces bugs, and promotes better software design.

Understanding these indicators provides the foundation for detecting smells using static metrics and machine learning models, as described in the proposed framework.

III. FRAMEWORK FOR AUTOMATED CODE SMELL DETECTION USING MACHINE LEARNING

A. Overview of the Framework

The proposed framework combines static code analysis and machine learning (ML) to provide an adaptive and efficient approach to code smell detection in legacy systems. It focuses on identifying problematic code components, classifying them into specific code smell categories, and offering actionable refactoring suggestions. The framework is divided into three major components:

- Static Code Metric Extraction: Collection and preprocessing of code metrics that quantitatively represent structural properties of the codebase.
- ML-Based Code Smell Detection: Supervised ML models classify code components based on extracted metrics to identify potential smells.
- Actionable Insights and Feedback Mechanism: ML models further analyze predictions to provide prioritized recommendations, severity analysis, and actionable refactoring guidance.
- Each component of the framework operates in a pipeline-like architecture, ensuring modularity and scalability.

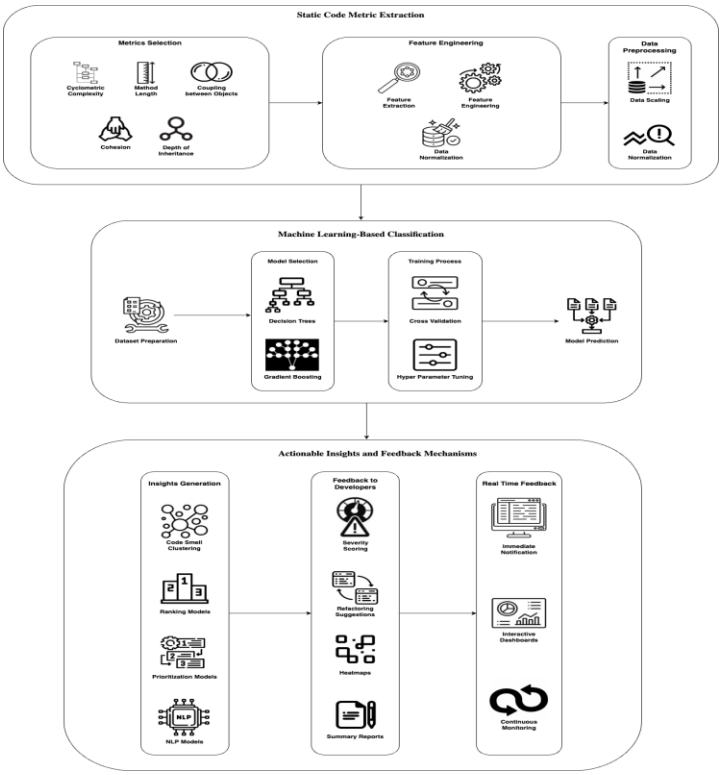

Figure 2: Framework for Automated Code Smell Detection

A. Components of the Framework

a) Static Code Metric Extraction

This is the foundation of the framework, where meaningful structural features of the source code are computed. These metrics serve as input variables for the ML model. The extraction process involves:

i) Metrics Selection

Static code metrics are key indicators of structural and design quality within software. These metrics are extracted from source code without executing it, making them particularly well-suited for analyzing legacy systems. The selected metrics represent various aspects of code complexity, modularity, and maintainability:

- Cyclomatic Complexity: Captures the complexity of a codebase by counting the number of independent paths through a program's control flow. High values suggest overly complex methods that are difficult to understand and test.
- Method Length: Measures the number of lines of code in a method. Excessively long methods often violate the principle of single responsibility, making them harder to maintain and reuse.
- Coupling between Objects (CBO): Indicates the degree of interdependence between classes. High coupling reduces modularity and increases the risk of cascading failures during modifications.
- Cohesion (LCOM): Represents the degree to which the methods in a class are related to one another. Low cohesion suggests a poorly organized class that performs unrelated tasks.
- Depth of Inheritance: Refers to the number of levels in the inheritance hierarchy. Excessive depth can complicate the understanding of object relationships and behavior.

ii) Feature Engineering

Metrics are preprocessed to derive additional meaningful features.

For example:

- Weighted coupling based on the strength of dependency.
- Normalized complexity scores to handle varying codebase sizes.

iii) Data Preprocessing

To ensure compatibility with the ML models

- Scaling: Metrics are normalized to maintain consistency across varying ranges.
- Missing Data Handling: Imputation techniques are applied to handle missing or incomplete metric values.

b) Machine Learning-Based Classification

The ML component forms the heart of the framework, leveraging supervised learning to classify code components based on the likelihood of exhibiting specific code smells. The technical aspects include:

i) Model Selection

Two ML models are central to the framework

- Decision Trees: A simple, interpretable model that maps input metrics to code smell classifications. Each node in the tree represents a decision based on a metric value, making the results easy to understand.
- Gradient Boosting: A more complex ensemble model that combines multiple decision trees to improve classification accuracy. Gradient boosting is particularly effective in handling overlapping and nuanced code smell patterns.

ii) Training Process

The models are trained on a labeled dataset where each data point represents a code component annotated with the presence or absence of specific smells. Key steps include:

- Dataset Preparation: Training data could be curated with labels for common smells like "God Class" and "Feature Envy."
- Cross-Validation: K-fold cross-validation is employed to validate the model and prevent overfitting, ensuring robustness across different datasets.
- Hyperparameter Tuning: Parameters such as the maximum depth of decision trees or the learning rate for gradient boosting are optimized to balance performance and computational efficiency.

iii) Prediction

Once trained, the models classify code components into smell categories, along with a confidence score for each prediction. For example:

- A method may be flagged as a "Long Method" with high confidence.
- A class may be identified as a "God Class" based on its high complexity and low cohesion.

c) Actionable Insights and Feedback Mechanism

This layer translates the results of the detection models into meaningful, actionable insights for developers. It incorporates additional ML models and techniques to refine the predictions and assist in prioritizing refactoring efforts.

i) Insights Generation Using ML

This layer applies additional models to analyze, prioritize, and generate insights based on the detected smells:

1. Clustering Models (e.g., K-Means, DBSCAN):
   - Used to identify clusters of recurring code smells across the codebase.
   - Example: Multiple instances of "Feature Envy" across a specific module may suggest systemic design issues.
2. Ranking and Prioritization Models (e.g., Logistic Regression, Multi-Criteria Decision Making):
   - These models rank code smells based on severity, maintainability impact, and developer-defined criteria.
   - Example: "God Class" with high coupling may be ranked higher than "Lazy Class" for immediate refactoring.
3. Natural Language Processing (NLP) Models:
   - Used to generate developer-friendly descriptions of detected smells and corresponding refactoring suggestions.
   - Example: An NLP model might generate a recommendation like, "Consider splitting this method into smaller methods with distinct responsibilities."

ii) Feedback to Developers

The feedback mechanism ensures that the framework integrates seamlessly into development workflows, providing intuitive and actionable insights:

1. Severity Scoring:
   - Each detected smell is assigned a severity score based on the confidence level of the ML model and the extent of metric deviations.
   - Example: A cyclomatic complexity of 70 may result in a "critical" severity, while a complexity of 25 might be flagged as "moderate."

2. Refactoring Suggestions:
   ML-driven insights are mapped to specific refactoring actions.

Examples include:

- Long Method: Suggest breaking the method into smaller units with single responsibilities.
- High Coupling: Recommend introducing interfaces or dependency injection to decouple classes.
- Low Cohesion: Suggest splitting a class into multiple classes, each with a cohesive purpose.

3. Visualization and Reporting:
   - Heatmaps highlight areas of the codebase with a high density of smells.
   - Dependency graphs visualize tightly coupled classes or modules.
   - Summary reports provide an overview of detected smells, ranked by severity and potential impact.

iii) Real-Time Feedback

For enhanced usability, the feedback mechanism can be integrated into development environments, offering:

- Immediate Notifications: Flagging smells as developers write code, enabling proactive resolution.
- Interactive Dashboards: Allow developers to explore detected smells, view detailed explanations, and track the impact of refactoring on code quality.
- Continuous Monitoring: Integration with CI/CD pipelines ensures that new smells are identified and addressed during code reviews or builds.

d) Advantages of the Framework

- Enhanced Precision: Combines multiple ML models to reduce false positives and improve detection accuracy.
- Scalability: Adapts to large and complex legacy systems with diverse architectures.
- Actionable Results: Provides meaningful, ranked refactoring suggestions to help developers prioritize technical debt reduction.
- Adaptability: Can retrain models as codebases evolve or new code smells are defined.
- Explainability: Interpretable models (e.g., decision trees) ensure that developers trust the predictions and insights.

II. CHALLENGES, CONSIDERATIONS AND FUTURE DIRECTIONS

While the proposed machine learning-based framework for code smell detection offers significant potential, its practical application involves several challenges that must be addressed. These challenges span data availability, model generalization, and developer adoption. Additionally, future advancements and extensions can further enhance the framework's scalability, adaptability, and trustworthiness.

A. Challenges and Considerations

a) Data Challenges:

Machine learning models rely heavily on high-quality labeled datasets for training and validation. However, obtaining such datasets for code smell detection poses significant challenges:

- Code smells are subjective in nature, as their identification often depends on the expertise and judgment of developers.
- Manually labeling large codebases is time-consuming and expensive.
- Publicly available datasets with annotated code smells are limited, making it difficult to train robust and generalizable models.

b) Generalization:

A major consideration for the framework is its ability to generalize across different programming languages, coding styles, and project types:

- Codebases in diverse languages (e.g., Java, Python, C++) exhibit variations in syntax, structure, and complexity. Static metrics may need language-specific adaptations.
- Models trained on one type of project may perform poorly when applied to other domains.
- Ensuring the framework adapts to heterogeneous codebases while maintaining accuracy remains a significant technical challenge.

c) Adoption Challenges:

For developers to trust and adopt AI-driven tools, transparency and interpretability are critical:

- Black-box machine learning models may flag code smells without clear explanations, making it difficult for developers to

understand or trust the predictions.
- Explainability is essential to provide reasoning for detected smells and justify refactoring recommendations.
- Overcoming resistance to automation requires ensuring that the tool produces low false positives and integrates seamlessly into development workflows.

B. Future Directions

To address these challenges and further improve the framework, the following research directions and advancements are proposed:

a) Advanced Learning Techniques:

- Unsupervised and Semi-Supervised Learning: Given the scarcity of labeled datasets, exploring unsupervised or semi-supervised techniques can enable the detection of patterns and smells without extensive annotations. Clustering methods (e.g., K-Means) or anomaly detection algorithms could identify unknown or emerging smells.
- Transfer Learning: Transfer learning approaches can help adapt pre-trained models to new programming languages or codebases with minimal retraining. This would improve the framework's generalization capabilities across domains and ecosystems.

b) Explainable AI (XAI):

Incorporating Explainability into the ML models can address developer trust issues:

- Techniques like SHAP (SHapley Additive exPlanations) or LIME (Local Interpretable Model-Agnostic Explanations) can provide insights into how specific metrics contribute to predictions.
- By presenting clear explanations for flagged smells, such as highlighting contributing factors (e.g., cyclomatic complexity or low cohesion), developers can better understand and act on recommendations.

c) Building Large-Scale Open Datasets:

Future research efforts should focus on curating large, diverse, and open-source datasets with labeled code smells:

- Such datasets would serve as benchmarks for evaluating ML-based frameworks, promoting consistency in model comparisons.

d) Impact Assessment on Technical Debt:

Empirical studies are needed to evaluate the framework's impact on reducing technical debt and improving maintainability:

- Measuring the reduction in code smells over time through automated refactoring suggestions.
- Analyzing developer productivity, such as time saved through automated detection and real-time feedback mechanisms.

e) Adaptive Framework Design:

The framework could be extended to include continuous retraining mechanisms that adapt to evolving codebases:

- As developers refactor code, new metrics and patterns could be incorporated to fine-tune the ML models.
- Feedback loops could enable the tool to learn from developer actions, improving accuracy over time.

Addressing challenges like data scarcity, model generalization, and developer trust is crucial for the successful adoption of the proposed framework. Future advancements, such as leveraging unsupervised learning, enhancing explainability, and building open datasets, can further strengthen the framework's effectiveness and scalability. By exploring these directions, the framework has the potential to become an integral part of modern software engineering practices, helping to systematically reduce technical debt and improve code maintainability.

III. CONCLUSION

Legacy systems, while vital to organizational operations, are often burdened by code smells that degrade software quality, increase technical debt, and hinder long-term maintainability. Traditional static code analysis tools, though widely used, are limited by their rule-based nature, high false positives, and significant reliance on manual validation. This paper proposes a conceptual framework that leverages machine learning (ML) to address these limitations, providing an automated and adaptive solution for code smell detection.

The proposed framework combines static code metrics—such as cyclomatic complexity, method length, coupling, and cohesion—with supervised ML models like decision trees and gradient boosting. By learning from labeled datasets, the framework can identify patterns in code that indicate the presence of smells more effectively than traditional tools. Furthermore,

the integration of advanced techniques, such as clustering for smell pattern detection, severity-based prioritization, and natural language processing (NLP) for actionable refactoring recommendations, ensures that developers receive meaningful insights.

This conceptual framework also proposes integration to modern development environments, such as Integrated Development Environments (IDEs) and CI/CD pipelines, enabling real-time feedback and proactive code quality improvements. This seamless integration has the potential to significantly reduce manual effort, accelerate refactoring decisions, and prevent the accumulation of technical debt.

Despite its potential, the paper also acknowledges the challenges, such as the scarcity of high-quality labeled datasets, the need for generalization across languages, and the importance of explainability to foster developer trust. Future research directions include exploring unsupervised and semi-supervised learning techniques, incorporating explainable AI (XAI) models, and curating large-scale, open-source datasets to benchmark and validate such frameworks.

By addressing these challenges and advancing the proposed framework, this work lays the foundation for leveraging AI-driven approaches to enhance software quality in legacy systems. The integration of machine learning into software engineering practices represents a significant step toward reducing technical debt, improving maintainability, and ensuring the long-term health of software systems.

IV. REFERENCES

[1] A. Fatima, S. Bibi, and R. Hanif, “Comparative study on static code analysis tools for C/C++,” in *15th International Bhurban Conference on Applied Sciences and Technology (IBCAST)*, 2018, pp. 465–469.

[2] H. K. Brar and P. Kaur, “Comparing detection ratio of three static analysis tools,” *International Journal of Computer Applications*, vol. 124, pp. 35–40, 2015.

[3] M. A. K. Maduranga, D. C. Mahagamage, P. Madhavi, J. A. H. Madushan, and C. Wijesiriwardana, “Domain-specific infrastructure for code smell detection in large-scale software systems,” unpublished, 2016.

[4] A. Jesudoss, S. Maneesha, and T. L. N. Durga, “Identification of code smell using machine learning,” in *2019 International Conference on Intelligent Computing and Control Systems (ICCS)*, 2019, pp. 54–58.

[5] F. Fontana, M. Mäntylä, M. Zanoni, and A. Marino, “Comparing and experimenting machine learning techniques for code smell detection,” *Empirical Software Engineering*, vol. 21, pp. 1143–1191, 2016.

[6] T. Guggulothu and S. A. Moiz, “Code smell detection using multi-label classification approach,” *Software Quality Journal*, vol. 28, pp. 1063–1086, 2019.

[7] A. Tahmid, M. N. A. Tawhid, S. Ahmed, and K. Sakib, “Code sniffer: A risk-based smell detection framework to enhance code quality using static code analysis,” *International Journal of Software Engineering and Technology Applications*, vol. 2, p. 41, 2017.

[8] Tushar, "How to track code smells effectively," *Medium*, Apr. 04, 2018. [Online]. Available: https://tusharma.medium.com/how-to-track-code-smells-effectively-48dbf5ba659d.