Ai-Powered Intelligent Code Dependency Mapping and Automated Refactoring Recommendation System for Large-Scale Software Applications

J. Blessing Bowmi[1], Mr. K. Maniraj[2]

[1]*PG student, Department of Computer Applications, SRM Valliammai Engineering College, Chennai.*

[2]*Assistant Professor, Department of Computer Applications, SRM Valliammai Engineering College, Chennai.*

Abstract—**As software systems grow, they become complex and difficult to maintain, leading to increased technical debt and reduced software quality. Traditional static analysis tools rely on simple rule-based techniques and often fail to detect deeper architectural issues such as circular dependencies and tightly coupled modules. To address this limitation, the proposed system introduces an AI-powered code analysis framework that combines AST parsing and graph-based dependency modeling using NetworkX to analyze complex code structures.**

The system detects code smells, duplication, and architectural flaws, and provides intelligent and context-aware refactoring suggestions. A Random Forest machine learning model evaluates code quality by generating a maintainability score and predicting refactoring risks. Additionally, an Adaptive Learning Engine improves the system over time by learning from developer feedback. By integrating graph analysis with machine learning, the system reduces manual effort, enhances decision-making, and supports safer, more efficient code maintenance, ultimately helping to control technical debt in large-scale software systems.

Index Terms—**Code Refactoring, Dependency Mapping, Code Smells, Static Code Analysis, Machine Learning.**

I. INTRODUCTION

In modern software development, large-scale systems often accumulate technical debt, leading to issues such as code smells, tight coupling, and reduced maintainability. This phenomenon, known as software aging, increases complexity, slows development, and raises maintenance costs. Refactoring is essential to improve code quality, but in large and complex codebases, manual refactoring is time-consuming, error-prone, and difficult to manage.

Traditional static analysis tools like SonarQube and ESL int help detect basic issues but rely on rigid rules and lack deeper architectural understanding. They often generate false positives and fail to provide actionable refactoring solutions. To address these limitations, this research proposes an AI-powered intelligent system that combines AST parsing, graph-based dependency analysis (NetworkX), and machine learning techniques. The system not only detects structural issues but also generates automated, context-aware refactoring suggestions. Additionally, an adaptive learning mechanism improves recommendations over time, making code maintenance more efficient, intelligent, and proactive.

II. LITEREATURE REVIEW

Software maintenance and refactoring are critical aspects of large-scale software development. As applications grow in size and complexity, managing code dependencies and maintaining code quality become increasingly difficult. Several research studies and tools have been developed to address issues related to code analysis, dependency mapping, and automated refactoring.

A. Code Dependency Analysis in Software Systems:
Several studies have focused on analyzing dependencies between modules in large software systems. Dependency analysis helps developers understand relationships between files, functions, and classes. Research by Martin Fowler emphasizes the importance of identifying tight coupling and improving modular design through refactoring

practices. Dependency mapping tools help visualize relationships between software components, enabling developers to detect architectural flaws. However, many traditional dependency analysis tools provide only static visualization and do not recommend automated improvements.

B. Static Code Analysis Techniques:
Static code analysis is widely used to detect bugs and code quality issues without executing the program. Tools such as SonarQube and PMD analyze source code to identify potential problems like duplicate code, unused variables, and complexity issues.
These tools provide detailed reports on code quality metrics, including maintainability and reliability. However, they often require manual interpretation by developers and provide limited automated refactoring guidance.

C. Automated Refactoring Approaches:
Refactoring is a technique used to improve the internal structure of code without changing its external behavior. Refactoring techniques were extensively discussed in the book Refactoring: Improving the Design of Existing Code by Martin Fowler.Automated refactoring tools attempt to identify code smells such as long methods, duplicated code, and large classes. Some Integrated Development Environments (IDEs) provide automated refactoring suggestions. However, most tools are limited to small code changes and do not analyze the overall architecture of large-scale systems.

D. AI-Based Software Engineering Tools:
Recent research has explored the integration of Artificial Intelligence into software engineering processes. AI models can analyze patterns in source code and provide intelligent recommendations.
For example, GitHub Copilot assists developers in generating code suggestions based on context. Similarly, machine learning techniques are being applied to detect software defects and recommend improvements.
Despite these advancements, many AI-based tools focus on code generation rather than comprehensive dependency mapping and automated refactoring.

E. Large-Scale Software Maintainability Research:
Maintaining large-scale software systems is a major challenge due to increasing complexity and interdependencies. Studies in Software Engineering highlight the need for automated systems that analyze software architecture and provide actionable insights for developers.
Research has shown that tools combining dependency analysis, code smell detection, and automated recommendations significantly improve maintainability and reduce technical debt.

III. PROBLEM STATEMENT

The continuous accumulation of technical debt increases maintenance cost and reduces development efficiency in large-scale software systems. As projects grow across multiple developers and technologies, identifying deep structural issues becomes difficult for developers. Manual code reviews are subjective and often fail to detect complex problems such as high module coupling, circular dependencies, and duplicate code.
Existing static analysis tools can detect basic issues but lack the ability to understand the overall architecture of the system and provide actionable solutions. They generate alerts without offering safe and effective refactoring methods. Therefore, there is a need for an intelligent system that can analyze code as a structured graph, detect hidden issues, and provide automated, context - aware refactoring recommendations

IV. OBJECTIVE

The objective of this project is to develop an intelligent system for automatic code analysis and improvement of large-scale software. It aims to detect code smells, map dependencies, and identify structural issues such as circular dependencies. The system provides AI-based refactoring suggestions, calculates maintainability scores, and generates reports to improve code quality and reduce technical debt. It also offers interactive visualizations to help developers better understand code structure. Additionally, the system is designed to learn from user feedback and continuously enhance its recommendations.

V. SYSTEM ARCHITECTURE

The AI-Powered Intelligent Code Dependency Mapping and Automated Refactoring Recommendation System uses a three-tier architecture consisting of Presentation, Application, and Data Persistence layers.

A. System Architecture Diagram:

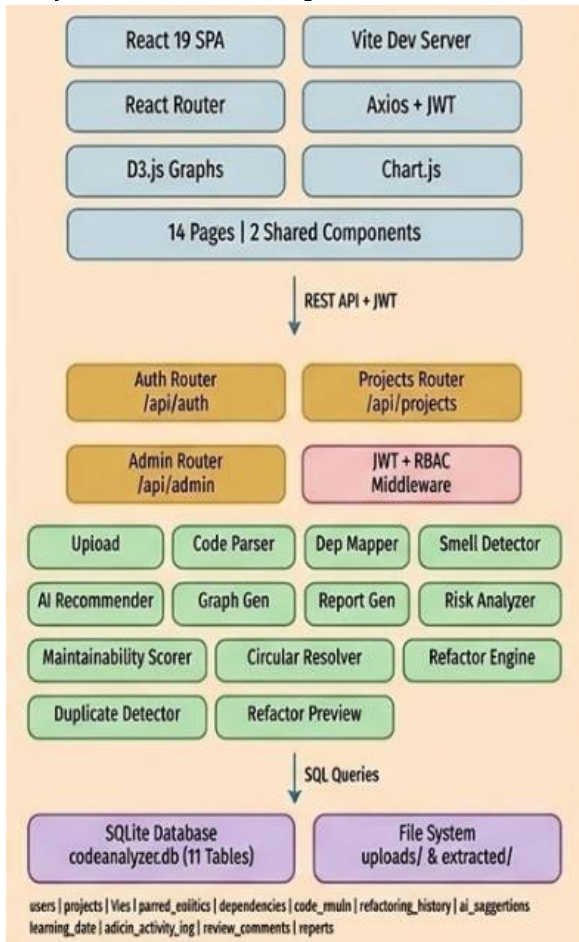


Figure.1-System architecture

The Presentation Layer is built with React and Vite, providing a user interface to upload project ZIP files and view analysis results. The frontend communicates with the backend through REST APIs using Axios and JWT authentication. It also uses D3.js to visualize module dependencies through interactive graphs.

The Application Layer is developed using FastAPI, which processes all requests and manages the code analysis pipeline. It includes routers such as Auth Router for authentication, Projects Router for code analysis, and Admin Router for system management. The analysis pipeline contains modules like Code Parser, Dependency Mapper, Smell Detector, Refactor Engine, and AI Recommender.

The Data Persistence Layer stores uploaded files in the file system and analysis data in an SQLite database.

Finally, the system generates reports and maintainability scores to help developers improve code quality and software maintainability.

B. Data Flow Diagram (DFD):

The Data Flow Diagram (DFD) of the system, showing how data moves through different stages of processing.

The process begins when the user uploads a ZIP file, which acts as the input data to the system. This file is passed to the Upload & Extract Module, where the ZIP is validated and its contents are extracted into individual source code files. The extracted data is then sent to the Code Parser, which analyzes the code and converts it into structured information such as classes, functions, and imports.

Next, the processed data flows to the Dependency Mapper, which builds relationships between different modules and identifies how files are interconnected. This structured data is further passed to the Code Smell Detection Module, where the system identifies issues like long functions, large classes, and deep nesting. Based on these detected problems, the Refactor Engine generates improved versions of the code.

The refined data is then processed by the AI Recommender, which enhances the suggestions using machine learning techniques to provide more intelligent and context-aware recommendations. After all analysis is completed, the data flows to the Report Generator, where results such as code quality metrics, graphs, and suggestions are compiled into structured reports.

Finally, the output data is delivered to the Frontend UI, where the user can view the analysis results, visualizations, and recommendations. Overall, the diagram clearly shows how raw input data is transformed step-by-step into meaningful insights through the system

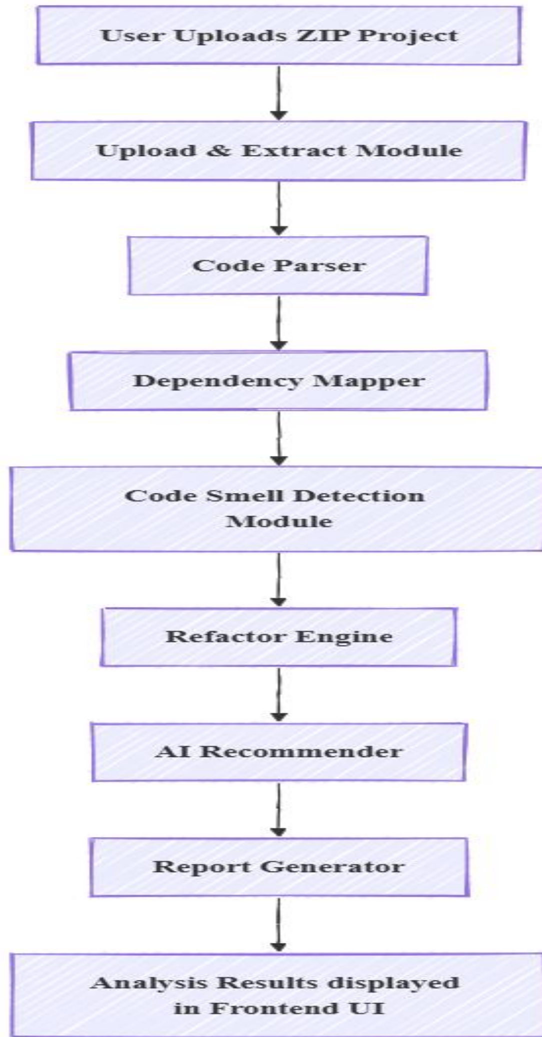


Figure 2-Dataflow diagram

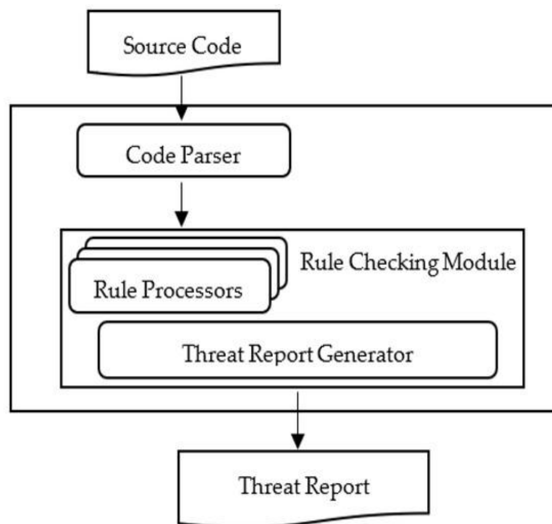


Figure.3- Code Parser Module

VI. METHODOLOGY

The implementation of the system adheres to a highly synchronous, five-stage pipelined methodology designed to progressively translate raw source code into intelligent, actionable telemetry.

STAGE 1: INGESTION & PARSING

Ingestion & Lexical Parsing (The Abstract Syntax Tree) The primary analytical phase avoids rudimentary text-matching by constructing formal Abstract Syntax Trees (AST). Upon the ingestion of a source archive, the multi-language Code Parser engine scans the extracted files. For native Python targets, the system invokes the internal ast compiler to traverse formal nodes, isolating global classes, encapsulated functions, argument signatures, and exact file-line boundaries. For heterogeneous, non-native environments (e.g., Java, JavaScript), the system relies on a matrix of cascading Regular Expressions designed to safely identify class-method bounds without requiring localized compiler environments.

STAGE 2: DEPENDENCY GRAPH

Topological Graph Construction (Directed Cyclic Graphing) Post-extraction, the flat metadata arrays are transmuted into a relational topology. The Dependency Mapper algorithm iterates through all recorded import statements and method invocations, charting them as discrete "Nodes" connected by directional "Edges." Utilizing the Python NetworkX library, the system plots a massive Directed Cyclic Graph. This topological structure provides the foundation for executing Depth-First Search (DFS) algorithms, which are mathematically necessary to unearth deep architectural flaws—such as obfuscated circular dependency loops that inherently corrupt modularity.

STAGE 3: CODE SMELL DETECTION

Hybrid Heuristic Pattern Recognition (Smell Detection) With the structural and topological blueprints established, the nodes undergo rigorous severity-weighted heuristic scanning. The Smell Detector evaluates each component against a rigid set of quantitative thresholds. Functions exceeding predefined line lengths (Long Methods), classes commanding an excessive number of methods (God Classes), and blocks exhibiting Deep Nesting paths are

flagged and mapped directly to their parent node. Concurrently, a secondary Duplicate Detector scans the codebase utilizing cryptographic SHA-256 line-item hashing. By stripping out ambient whitespace and neutralizing variable names, the algorithm reliably isolates severe copy-paste logic fragmentation across entirely decoupled modules.

STAGE 4: REFACTORING ENGINE

Generative Restructuring (The Refactor Engine) Diverging from traditional static analyzers that halt at the diagnostic phase, the methodology proceeds to generative remediation. By intersecting the flagged node's topological risk against its known anti-pattern, the system automatically selects applied refactoring templates derived from Martin Fowler's baseline strategies. For a flagged "Long Method," the engine synthetically abstracts isolated blocks spanning internal variables into a standalone static method construct. A specialized Python difflib processor then calculates the absolute byte-difference between the original execution path and the generated code, compiling a secure, readable Unified Visual Diff for developer verification.

STAGE 5: AI-BASED EVALUATION

Algorithmic Intelligence & Predictive Risk Scoring The final methodological stage utilizes supervised Machine Learning to grade the structural integrity of the project and predict the viability of the generated code adjustments. The Maintainability Scorer aggregates the raw frequency of anti-patterns and network coupling density. These aggregated metrics are funneled into a predictive Random Forest classifier (implemented via Scikit-Learn), returning a continuous Project Maintainability Grade (0.00 – 100.00). Furthermore, this predictive engine is strictly tied to a persistent Bayesian Decision Tree matrix—the Learning Engine—which silently records whether human developers "Accept" or "Reject" the proposed visual diffs. This feedback loop adjusts the heuristic weighting dynamically, guaranteeing that the system’s refactoring behavior converges precisely with the architectural tolerance of the deployment team.

VII. SYSTEM IMPLEMENTATION & RESULT

The implementation of the proposed system is carried out through a structured integration of backend processing, graph-based analysis, machine learning, and frontend visualization components, ensuring efficient and scalable code analysis.

A. BACKEND ANALYTICS COMPILATION

The core backend is developed using Python 3.10, leveraging its advanced data handling capabilities. An asynchronous FastAPI server, deployed using Uvicorn, manages API requests and ensures high performance during heavy computations. When a user uploads a ZIP file, the Upload Module securely extracts the contents using Python’s built-in libraries. The extracted files are indexed and passed to the Code Parser, which utilizes the AST (Abstract Syntax Tree) module to convert raw source code into structured, analyzable data such as classes, functions, and their relationships.

B. GRAPH THEORY EXECUTION

The system implements dependency analysis using graph theory concepts with the help of the NetworkX library. Parsed code elements are represented as nodes, while relationships such as function calls and imports are represented as edges in a directed graph (DiGraph). Advanced algorithms like Depth-First Search (DFS) are used to detect complex patterns such as circular dependencies. This graph-based approach enables accurate identification of architectural issues that are difficult to detect using traditional methods.

C. MACHINE LEARNING MODEL TRAINING & DEPLOYMENT

The intelligent recommendation system is built using Scikit-Learn. A Random Forest Classifier is trained on datasets containing code metrics and maintainability scores. During execution, real-time metrics extracted from the analyzed code—such as coupling and code smell counts—are normalized and passed to the trained model. The system then predicts a Maintainability Score and Refactoring Risk Probability, helping developers make informed decisions about code improvements.

D. FRONTEND INTERFACE INITIALIZATION

The frontend is developed using React 19 and Vite, providing a fast and interactive user interface. The frontend communicates with the backend through RESTful APIs and displays results dynamically. Dependency graphs generated in the backend are

converted into JSON format and visualized using D3.js, which renders interactive graphs with node relationships. This allows users to explore code structure visually, making the analysis more intuitive and user-friendly.

Overall, the system integrates backend processing, graph theory, machine learning, and interactive visualization to provide a comprehensive and intelligent code analysis platform.

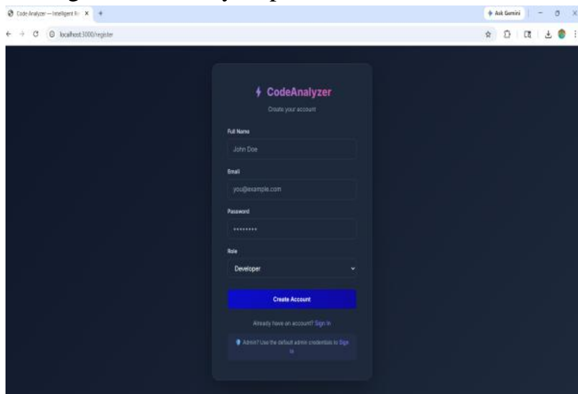


Figure:4 -Registration

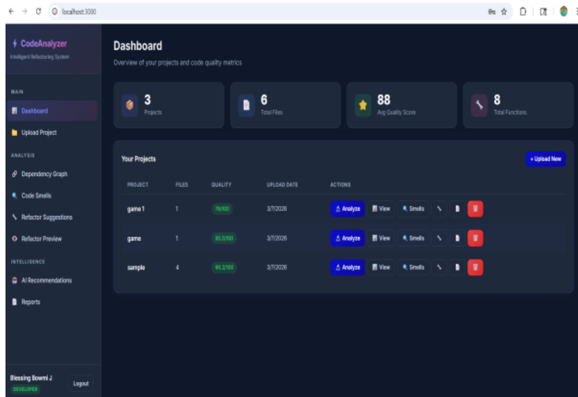


Figure:5 -Developer Mode

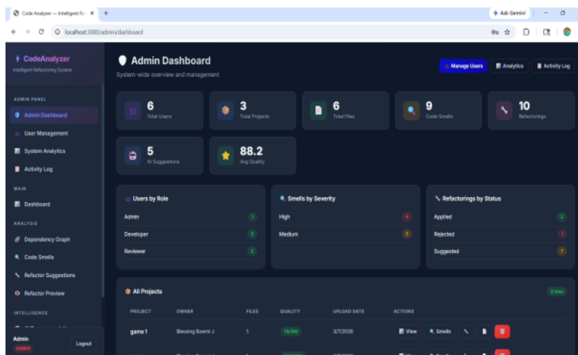


Figure: 6 -Admin Dashboard

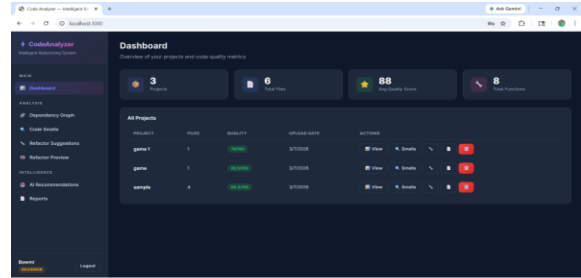
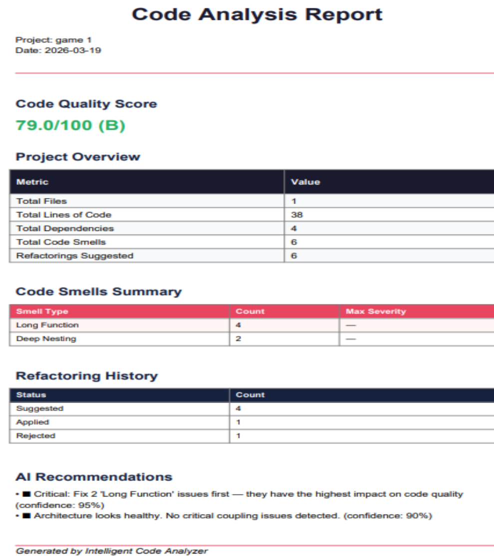


Figure:7 -Reviewer Dashboard



Code Analysis Report

Project: game 1
Date: 2026-03-19

Code Quality Score

79.0/100 (B)

Project Overview

Metric	Value
Total Files	1
Total Lines of Code	38
Total Dependencies	4
Total Code Smells	6
Refactorings Suggested	6

Code Smells Summary

Smell Type	Count	Max Severity
Long Function	4	—
Deep Nesting	2	—

Refactoring History

Status	Count
Suggested	4
Applied	1
Rejected	1

AI Recommendations

- ■ Critical: Fix 2 'Long Function' issues first — they have the highest impact on code quality (confidence: 95%)
- ■ Architecture looks healthy. No critical coupling issues detected. (confidence: 90%)

Generated by Intelligent Code Analyzer

Figure:8 Code Analysis Report

VIII. CONCLUSION

In conclusion, the research demonstrates how static code analysis can be transformed from a simple diagnostic tool into an intelligent system for improving software quality. The proposed Automated Refactoring Recommendation System combines code parsing, dependency graph analysis, and machine learning to detect code smells, circular dependencies, and structural issues. It provides automated refactoring suggestions to help developers improve code maintainability. The refactor preview feature allows users to review changes before applying them, increasing confidence in the recommendations. Additionally, the learning component adapts to developer feedback and improves suggestions over time. Overall, the system helps reduce technical debt and supports better maintenance of large-scale software systems.

IX. FUTURE SCOPE

Future improvements of the system include deep IDE integration by developing native plugins for tools like VSCode and IntelliJ, enabling real-time refactoring recommendations while developers write code. The system can also be enhanced by replacing static heuristic-based suggestions with advanced Large Language Models (LLMs) that can better understand code context and generate more intelligent refactoring solutions. Another future enhancement is the integration with CI/CD pipelines such as GitHub Actions or Jenkins, allowing the system to automatically analyze new commits and generate refactoring pull requests. Additionally, the system can be extended to support broader programming languages by integrating native AST frameworks such as Clang for C++ and JDT for Java, enabling deeper analysis similar to the current Python-based implementation.

REFERENCES

[1] Bharadwaj, R., & Parker, I. (2023). Double-edged sword of LLMs: mitigating security risks of AI-generated code. In Disruptive Technologies in Information Sciences VII (Vol. 12542, pp. 141–146). SPIE.

[2] Yadav, P. S., Rao, R. S., Mishra, A., & Gupta, M. (2024). Machine learning-based methods for code smell detection: A survey. Applied Sciences, 14(14), 6149.

[3] Khleel, N. A. A., & Nehéz, K. (2024). Improving accuracy of code smells detection using machine learning with data balancing techniques. Journal of Supercomputing.

[4] Nyirongo, B., Jiang, Y., Jiang, H., & Liu, H. (2024). A survey of deep learning-based software refactoring.

[5] Ho, A., Bui, A. M. T., Nguyen, P. T., Di Salle, A., & Le, B. (2025). EnseSmells: Deep ensemble and programming language models for automated code smell detection.

[6] Pantiuchina, J., Lanza, M., & Bavota, G. (2020). Why developers reject automated refactoring. IEEE Transactions on Software Engineering, 47(11), 2419–2434.

[7] Pantiuchina, J., Lanza, M., & Bavota, G. (2020). "Why developers (reject) automated refactoring". IEEE Transactions on Software Engineering, 47(11), 2419-2434.

[8] Sharma, T., & Spinellis, D. (2018). "A survey on software smells". Journal of Systems and Software, 138, 158-173.