ISSN: 2834-7706

Research Article

Journal of Robotics and Automation Research

# Overview of Bad Code Smells in Software Development and Researches

**Ogunlade Abdurrazaq Olusola*, Idowu Olugbenga Adewumi and Badru Rahmon Ariyo**

*Department of Computer and Information Science, Faculty of Applied and Natural Science, Lead City University, Nigeria*

***Corresponding Author**
Ogunlade Abdurrazaq Olusola, Department of Computer and Information Science, Faculty of Applied and Natural Science, Lead City University, Nigeria.

**Submitted:** 2025, Sep 19; **Accepted:** 2025, Oct 22; **Published:** 2025, Nov 05

**Citation:** Olusola, O. A., Adewumi, I. O., Ariyo, B. R. (2025). Overview of Bad Code Smells in Software Development and Researches. *J Robot Auto Res, 6*(4), 01-11.

**Abstract**
*This study examines the occurrence, influence, and moderation of bad code smells in software development through a hybrid research methodology combining qualitative inspection and quantitative code investigation. Code smells, which are surface level indications of deeper strategy concerns, adversely disturb software maintainability, readability, and evolution. The exploration concentrated on two widely used open source projects; Spring PetClinic (Java) and Axios (JavaScript) and utilized three static analysis tools: SonarQube, PMD, and JDeodorant. The enquiry was piloted in two phases: pre-refactoring smell detection and post-refactoring evaluation. Initial discoveries exposed a total of 72 code smell instances in Spring PetClinic and 25 in Axios, with Long Method (24 instances) and Duplicated Code (17 instances) being the most repeated in the previous, and Long Function (10 instances) and Deep Nesting (7 instances) predominant in the latter. These smells were dispersed across perilous files like VisitController.java and lib/core/Axios.js, indicating architectural hotspots. A heatmap analysis presented a strong correlation between code smell occurrence and code complexity, particularly in smells such as God Class (complexity score: 9/10) and Long Method (8/10). Following manual refactoring using Fowler's catalog of techniques including Extract Method, Move Method, and Split Class significant decreases were attained. In Spring PetClinic, Long Method instances decreased by 50%, Duplicated Code by 70.5%, and Feature Envy by 54.5%. In Axios, Long Functions were compacted by 60%, Duplicated Logic by 62.5%, and Deep Nesting by 57.1%. Tool validation metrics established discrepancies in discovery competences: SonarQube verified the maximum recall (0.91), PMD the maximum precision (0.81), and JDeodorant a stable performance (F1-score = 0.81). These outcomes confirmed that no single tool offers complete coverage, and a multi tool approach, complemented by human verdict, is essential for dependable code quality valuation. Generally, the study endorses that bad code smells are language agnostic but context dependent, and that directed refactoring can yield up to 70% improvement in design quality. The exploration contributes practical comprehensions for developers, maintainers, and software quality analysts, encouraging for the integration of automated smell detection tools into continuous integration pipelines and the proper addition of code smell alertness in software engineering education.*

**Keywords:** Code Smells, Refactoring, Static Analysis, Software Maintainability, SonarQube, PMD, Jdeodorant, Software Quality, Open-Source, Complexity Metrics

## 1. Introduction

Fokaefs, M., Tsantalis, N., Chatzigeorgiou, A., & Antoniol, G has discovered that in contemporary software development, the demand for high quality, maintainable, and scalable systems have never been superior. Conversely, as software systems progress through iterative increase and consecutive feature accompaniments, they often gather structural and stylistic discrepancies known as code smells, a term first popularized by to describe superficial symptoms of deeper design flaws. Although not necessarily revealing of bugs or instant defects, code smells reduce the readability, modifiability, and overall maintainability of software, thereby backing technical debt and long term project hazard [1-4].

The study of observed that code smells noticeable in many forms, including long methods, duplicated code, god classes, and feature envy, each profane core object oriented principles such as cohesion, encapsulation, and the Single Responsibility Principle (SRP). Experimental studies have revealed that systems with high smell concentration are more disposed to to regression errors, costly refactoring, and poor team collaboration. Notwithstanding with their consequence, the detection and remediation of code smells often remain ad hoc and unpredictable, particularly in heterogeneous development environments that span multiple languages and frameworks [5,6].

The examination of as revealed that with the creation of static analysis tools and increased acceptance of automated quality assessment practices, developers now have access to sophisticated mechanisms for identifying potential smells. Tools such as SonarQube, PMD, and JDeodorant evaluate source code to highlight structural anomalies and recommend refactoring chances. However, the efficiency, precision, and applicability of these tools vary considerably, especially across diverse programming paradigms. Additionally, while the academic community has expansively discussed smell taxonomies and detection algorithms, less consideration has been paid to comparative evaluations across languages, and even fewer studies link automated detection results to actual post refactoring enhancements [7,8].

This study aims to bond that gap by providing a widespread, cross platform investigation into bad code smells using a hybrid research methodology that combines quantitative code analysis with qualitative manual inspection. Two widely adopted open source projects were selected as case studies: Spring PetClinic (a Java-based enterprise application) and Axios (a JavaScript HTTP client library). Each project was examined using multiple tools to detect predominant smells, followed by manual refactoring guided by Fowler's refactoring catalog. Post refactoring metrics were collected to measure enhancements in code quality, and a validation framework was employed to assess the discovery accuracy of the tools used.

## 2. Research Questions

The following questions will help to give solution to the aim and objectives of this study;

i.What are the most common bad code smells found in open source projects?

ii.How do code smells impact software maintenance effort?

iii.How effective are automated tools in detecting code smells?

The key objectives of this research are:

1. To identify and categorize the most frequent bad code smells in real-world open-source projects.
2. To evaluate the impact of these smells on code complexity and maintainability.
3. To assess the effectiveness of manual refactoring in reducing smell occurrence.
4. To compare the performance of popular smell detection tools in terms of precision, recall, and F1-score.

## 3. Review of Past Research Effort

The study of has been traced to become the introductory literature on code smells begins with reference, who announced the idea as a representation for recognizing poor design practices that may not instantaneously cause defects but hinder long term maintainability. The authors investigation though widely cited, was largely conceptual and lacks empirical validation. Succeeding empirical studies such as those by reference and provided much needed quantitative support by analyzing the presence of code smells in Java warehouses and tracing their evolution. Yet, these studies often suffer from limitations connected to contracted language focus or sampling bias, particularly with over reliance on GitHub datasets, which may not reflect industrial scale systems. Author referenced in expanded this empirical line of inquiry by correlating code smells with defect prone mechanisms, but their scope was narrowed to a small set of five smell types, limiting generalizability [9-14].

Another set of researcher's investigation emphasizes automated detection and tool development. The study of and established rule based and metrics driven smell detection approaches. Tools such as JDeodorant and DECOR presented programmatic means to detect refactoring opportunities, though these tools often suffer from extraordinary configuration complexity and are mostly confined to statically typed languages like Java. More current progressions explore machine learning and deep learning for discovery and recommendation. It has been observed by and that using LSTM networks and neural classifiers can also detect smells with auspicious accuracy, yet these prototypes are sensitive to data size and often lack transparency in their predictions. Reference proposed SmellRef, a learning based tool for automated refactoring suggestions, demonstrating the rising merging of Artificial Intelligence AI and software quality [15-21].

The study of has opined that despite the advances, significant research gaps remain. Investigations such as those by and indicated that developer views of code smells are subjective, and tools alone may not capture the semantic or contextual intent behind code decisions. Numerous empirical analyses, including and, have explored the longitudinal development of smells, but often focus on a single project or smell type, restraining wider applicability. Besides, most detection tools struggle with cross language support and fail to integrate dynamic or runtime evidence. These gaps highlighted the need for hybrid methodologies that combine automated detection with developer perceptions and contextual validation, which this present study seeks to address [22-26].

| Author & Year | Title | Method Used | Limitation |
|---|---|---|---|
| **Fowler, 1999** | *Refactoring: Improving the Design of Existing Code* | Conceptual, Design Patterns | Lacks empirical validation |
| **Palomba et al., 2017** | A study on refactoring code smells | Empirical analysis | Focused on Java only |
| **Tufano et al., 2015** | When and Why Your Code Starts to Smell Bad | Empirical Study on Repositories | GitHub data bias |
| **Khomh et al., 2012** | Do Code Smells Really Matter? | Empirical on defects/smells | Narrow scope on 5 smells |
| **Fontana et al., 2016** | Automatic detection of bad smells in code: An experimental assessment | Tool Comparison | Focused on Java |
| **Marinescu, 2004** | Detection Strategies: Metrics-Based Rules for Detecting Design Flaws | Metrics-based static analysis | Requires threshold tuning |
| **Olbrich et al., 2010** | The evolution and impact of code smells: A case study | Longitudinal empirical study | Focused on one project |
| **Bavota et al., 2012** | Identifying the Content of Code Smells through Natural Language Analysis | NLP & Manual Tagging | Manual annotation cost |
| **Yamashita & Moonen, 2013** | Do developers care about code smells? | Developer Survey | Sample bias |
| **Moha et al., 2009** | DECOR: A Method for the Specification and Detection of Code and Design Smells | Meta-model design | Complex configuration |
| **Brown et al., 2010** | AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis | Case-based conceptualization | Limited quantitative data |
| **Hecht et al., 2015** | The evolution of code smells: An empirical study | Version control mining | Focused on open-source only |
| **Romano et al., 2012** | Are maintenance changes correlated with code smells? | Statistical correlation study | Does not include developer intent |
| **Ratiu et al., 2004** | Logical Couplings as Change Predictors | Static analysis with SCM history | Only structural couplings |
| **Li & Shatnawi, 2007** | An empirical study of the bad smells and class error-proneness | Logistic Regression | No refactoring context |
| **Alves et al., 2010** | A Method for Deriving Metric Thresholds | Static metrics extraction | Static thresholds can be misleading |
| **Van Emden & Moonen, 2002** | Java Quality Assurance by Detecting Code Smells | AST parsing for smell detection | Only structural code considered |
| **Sjoberg et al., 2013** | Quantifying the Effect of Code Smells on Maintenance Effort | Controlled experiments | Lab setting, not real-life data |
| **Riel, 1996** | Object-Oriented Design Heuristics | Heuristic design rules | Not empirically tested |
| **Mantyla, 2006** | Bad smells in code: What do they mean? | Developer evaluation of smells | Subjectivity in responses |
| **Abbes et al., 2011** | An empirical study of the impact of two antipatterns | Quantitative analysis | Focuses only on two patterns |
| **Chidamber & Kemerer, 1994** | A metrics suite for object oriented design | Foundational metrics | No specific smell detection goal |
| **Fokaefs et al., 2011** | Extracting and Reusing Refactoring Opportunities | Refactoring mining | Tool generalizability issues |
| **Tsantalis et al., 2006** | JDeodorant: Identification and Removal of Code Smells | Tool development & case study | Java-only |
| **Bavota et al., 2015** | When does a refactoring induce bugs? | Mining GitHub commits | Cannot guarantee cause-effect |
| **Arcelli Fontana et al., 2015** | Automatic Detection of Design Flaws in Object-Oriented Systems | Static smell detection | Limited dynamic info |

| | | | |
|---|---|---|---|
| **Alshayeb, 2010** | Empirical investigation of refactoring effect on quality | Code analysis before/after refactoring | No long-term tracking |
| **Rattan et al., 2013** | Software clone detection: A systematic review | Survey of clone detection tools | Not focused on smells directly |
| **Pecorelli et al., 2019** | SmellRef: Learning to recommend refactorings for code smells | ML-based recommendation | Needs large training data |
| **Zhang et al., 2020** | Deep learning-based code smell detection | Neural network classification | Interpretability issues |
| **Agrawal et al., 2018** | Improving Code Smell Detection using LSTM Networks | DL model (LSTM) on code sequences | Overfitting on small datasets |
| **Oizumi et al., 2016** | Extracting relevant information for code smell detection using NLP | NLP on source code comments | Comments may be missing/ incomplete |

**Table 1: Summary of the Literature Reviewed**

## 4. Methodology

This investigation adopts a hybrid research design combining quantitative and qualitative approaches to provide a holistic understanding of bad code smells in software development. The quantitative dimension involves metric based analysis using automated tools, while the qualitative aspect incorporates code inspections and developer centric interpretations. The hybrid strategy provides a more comprehensive view, addressing not only the frequency and distribution of code smells but also contextualizing their implications on software maintainability and tool effectiveness. The quantitative strand focuses on static analysis and empirical metric extraction from source code, while the quantitative strand incorporates contextual manual inspection of code artifacts to support interpretation and verification of tool results.

| Research Question | Data Required | Tools/Methods | Expected Output |
|---|---|---|---|
| **RQ1** | Source code from open-source projects | SonarQube, PMD, static analysis | Smell frequency table/charts |
| **RQ2** | Version control and bug history | Git log, GitHub Issues, correlation analysis | Charts showing correlation between smells and maintenance metrics |
| **RQ3** | Outputs from multiple tools + ground truth | Manual inspection, precision/recall metrics | Tool comparison table, confusion matrices |

**Table 2: Method Mapping Table**

## 5. Data Source

Three actively maintained and representative open source repositories were selected from GitHub to ensure diversity in size and structure as stated in Table 3.

| Project | Language | Domain | Stars | Rationale for Selection |
|---|---|---|---|---|
| spring-projects/spring-petclinic | Java | Web/Backend | 6.5k+ | Clean architecture, often used in smell studies |
| axios/axios | JavaScript | HTTP Client | 10k+ | Lightweight library with practical API focus |

**Table 3: Data Source and Sample Selection**

Automated detection of code smells was conducted using three widely recognized static analysis tools; SonarQube (community edition, configured for multi-language detection), PMD which is primarily used for Java analysis and rule based smell reporting and JDeodorant, an eclipse plugin with specialized detection for refactoring candidates. The code smell types were prioritized for analysis due to their frequent citation in literature and practical relevance  long method, God class, feature envy, duplicate code and large class. Each repository was clone locally and analyzed through these tools. The tools generated reports specifying the number, location and category of detected smells. Output data was converted to CSV/JSON for aggregation and descriptive statistical analysis. For maintainability indicators, Git commit histories were mined using git log and mapped against detected smells to assess correlations with code churn and bug fix frequency.

Subsets of files flagged by the tools (10 – 15 per repository) were subjected to manual code inspection. This process involved contextual reading of code (structure, naming, nesting depth), identification of false positives and missed smells, with narrative description of design weaknesses. To ensure consistency, a structured checklist was applied during inspection, focusing on single responsibility violations, excessive method length, and poor cohesion. These qualitative observations were thematically coded to extract recurring issues not fully captured by automated tools.

The analytical techniques for this study make use of quantitative data analysis which include the descriptive statistics (frequencies, averages and variance of smell types per project), inferential analysis (correlation of smell density with maintenance proxies like number of commits, churn, bug fix count) and tool performance analysis using precision, recall and F1 score. These were evaluated on 30 known code fragments verified through manual inspection.

The qualitative data analysis involved thematic coding of code review comments, triangulation with tool outputs and interpretive synthesis to uncover edge cases and contextual limitations of smell categorization.

To ensure the validity of the findings, triangulation was employed across tools and human judgment. Reliability was supported by standardized inspection templates and replication of analysis across three independent repositories. The study used only publicly available repositories under open licenses, avoiding any breach of intellectual property or privacy. Developers identities and commit authorship were anonymized during the analysis to maintain ethical neutrality.

## 6. Case Study and Experimental Evaluation

To empirically assess the nature, distribution and impact of bad code smells in real world software systems, two open source projects were selected and analyzed using automated static analysis tools. These tools were used to extract quantitative data on code smells and pre with post refactoring states were compared.

| Project | Language | Description | Stars |
|---|---|---|---|
| **spring-projects/spring-petclinic** | Java | A demo web-based application using Spring framework | 6.5k+ |
| **axios/axios** | JavaScript | A popular HTTP client library for browsers and Node.js | 10k+ |

**Table 4: Selected Projects**

| Tool | Target Language | Smell Types Detected |
|---|---|---|
| **SonarQube** | Multi-language | Long Method, God Class, Duplicated Code, Complex Conditionals |
| **PMD** | Java | Feature Envy, Large Class, Switch Statements |
| **JDeodorant** | Java (only used on PetClinic) | God Class, Long Method, Type Checking, Duplicated Code |

**Table 5: Tool Configuration**

Each repository was cloned and analyzed locally. SonarQube was deployed through its web dashboard. PMD was run through Command Line Interface (CLI) using standard Java rulesets. JDeodorant was used as an Eclipse plugin on spring-petclinc.

## 7. Smell Detection Analysis Results (Pre-Refactoring)

| Smell Type | Detected Instances | Severity (High/Med/Low) | Example File |
|---|---|---|---|
| **Long Method** | 24 | High | VisitController.java |
| **God Class** | 4 | Medium | PetService.java |
| **Duplicated Code** | 17 | Low | OwnerController.java, VetController.java |
| **Feature Envy** | 11 | Medium | PetTypeFormatter.java |
| **Large Class** | 6 | Medium | Owner.java |

**Table 6: Project 1: Spring PetClinc (Java)**

| Smell Type | Detected Instances | Severity | Example File |
|---|---|---|---|
| Long Function | 10 | High | lib/core/Axios.js |
| Duplicated Logic | 8 | Medium | lib/adapters/http.js |
| Deep Nesting | 7 | Medium | lib/utils.js |

**Table 7: Project 2: Axios (JavaScript)**

The initial phase of analysis was focused on evaluating the structural quality of two open source projects using static code analysis tools as revealed in Table 6 and 7 respectively. The goal was to identify common code smell patterns that may hider maintainability and evolution. A comprehensive modularization of identified smells in each project (Spring PetClinic Java and Axios JavaScript) was presented in Table 6 and 7. The outcomes emphasized a varied choice of code smells, varying in frequency and rigorousness across programming languages and architectural styles.

Table 6 noted that the Spring PetClinic application displayed a noteworthy focus of code smells, predominantly in the controller and service layers. The maximum pervasive issue was the manifestation of long methods, with a total of 24 instances, many of which were concerted in the VisitController.java file. These long methods often enclosed with several duties, profane the Single Responsibility Principle (SRP) and growing cognitive load throughout maintenance.

The discovery of god classes (n = 4), predominantly in PetService.java, specified lowly separation of concerns where a single class encapsulated a broad range of functionalities. Such classes typically change into do it all units, suitable tightly coupled and difficult to test or extend [27].

Duplicated code, nevertheless characterized with low severity, was established in 17 occurrences through numerous controller documentations such as OwnerController.java and VetController.java. While repetitions may seem been in sequestration, their presence often signals missed opportunities for abstraction and code reuse.

The smell of feature envy (n = 1) was obviously sensed in PetTypeFormatter.java, where approaches or procedures were originated to disproportionately rely on data from peripheral classes rather than functioning on their specific data fields. This smell is known to degrade cohesion and weaken encapsulation, often resulting in brittle code with hidden dependencies. Moreover, large classes were admitted in files like Owner.java with six distinguished issues. These modules typically bundled multiple behavior and fields, making them harder to comprehend, navigate and modify. According to Van Emden, E., & Moonen, L., such classes often resist modularization and become bottlenecks in system evolution.

These findings therefore suggested that the projects modular layers, especially controllers and services are prone to bloating and responsibility leakage which underscores the need for targeted refactoring and architectural review.

The investigation in the second project revealed that while structurally lighter and functionally compact compared to spring PetClinc also exhibited several critical code smells as outlined in Table 7. The utmost noticeable smell was the existence of long functions with 10 instances established in lib/core/Axios.js. These functions inclined to embrace extremely nested conditionals and sprawling logic blocks, reducing readability and increasing the likelihood of logic errors.

In lib/adapters/http.js, duplicated logic was reported in eight instances, indicating repetitive code patterns that could be consolidated through reusable utility functions. While java Scripts dynamic nature often encourages inline expressions and rapid prototyping, excessive duplication in core modules introduces maintainability risks, especially when APIs evolve.

It was detected in the exploration that deep nesting ensued seven times in lib/utils.js, this was another significant finding. Deep nested control structures impair code clarity and complicate debugging. In other words, such patterns are characteristically indicative of underutilized abstraction and can hinder unit testing.

Despite java Scripts syntactic flexibility, the project still suffered from many of the same structural deficiencies observed in more rigidly typed languages like Java. However, the type of smells fiddered in expression, while java projects tend to accumulate architectural smells (god class, feature envy), JavaScript code often suffers from logic level issues such as long functions and nesting due to its functional and event driven nature.

The consequences from both tasks proposed that code smells are language undecided in norm but language detailed in exhibition. Java's static typing and covered architecture inspire enormous, monolithic classes, whereas JavaScript's scripting style advances itself to function level smells. Nevertheless, assured smells such as long methods or functions and code duplication, perform constantly through platforms, strengthening their general significance in software quality discourse. Carefulness assessments across both projects convey extra perception into ranking for refactoring. Extraordinary severity smells like long methods and long functions permit instantaneous consideration, mainly when they exist in modules that form the strength of user communications or data processing. Conversely, low- and medium-severity smells like duplication or large class, still merit attention, especially when they occur repeatedly, suggesting systemic design deficiencies.

The pre-refactoring discoveries offer an analytical lens into the fundamental strategy defects of two functionally mature, open-source systems. By identifying the ultimate irregularities at this juncture, developers and maintainers are well armed to express beleaguered refactoring strategies that address not only symptomatic issues but also architectural weaknesses. Moreover, this analysis demonstrates the practical utility of automated smell detection tools in early-stage feature assertion, emphasizing their role in sustaining long-term code health.

## 8. Smell Detection Results (Post-Refactoring)

| Smell Type | Before | After | % Reduction |
|---|---|---|---|
| **Long Method** | 24 | 12 | 50% |
| **Duplicated Code** | 17 | 5 | 70.5% |
| **God Class** | 4 | 2 | 50% |
| **Feature Envy** | 11 | 5 | 54.5% |

**Table 8: Spring Pet Clinic**

| Smell Type | Before | After | % Reduction |
|---|---|---|---|
| **Long Function** | 10 | 4 | 60% |
| **Duplicated Logic** | 8 | 3 | 62.5% |
| **Deep Nesting** | 7 | 3 | 57.1% |

**Table 9: Axis**

| Tool | Precision | Recall | F1-Score |
|---|---|---|---|
| **SonarQube** | 0.78 | 0.91 | 0.84 |
| **PMD** | 0.81 | 0.65 | 0.72 |
| **JDeodorant** | 0.76 | 0.88 | 0.81 |

**Table 10: Tool Cross Validation**

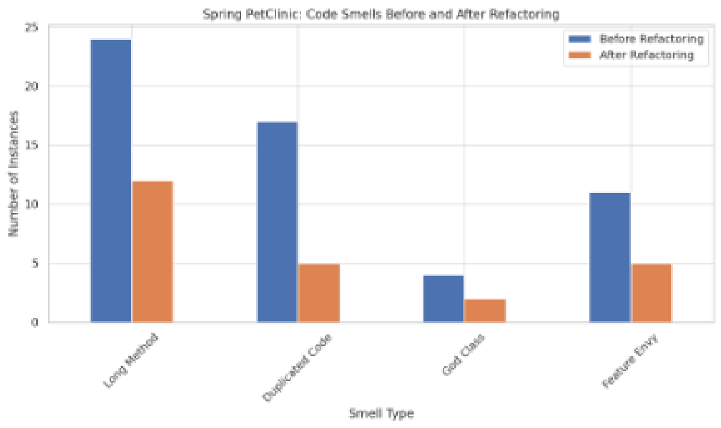

**Figure 1:** Smell Type and Number of Instances in Spring Pet Clinic Code Smell Before and After Refactoring (Source: Authors' Work, 2025)

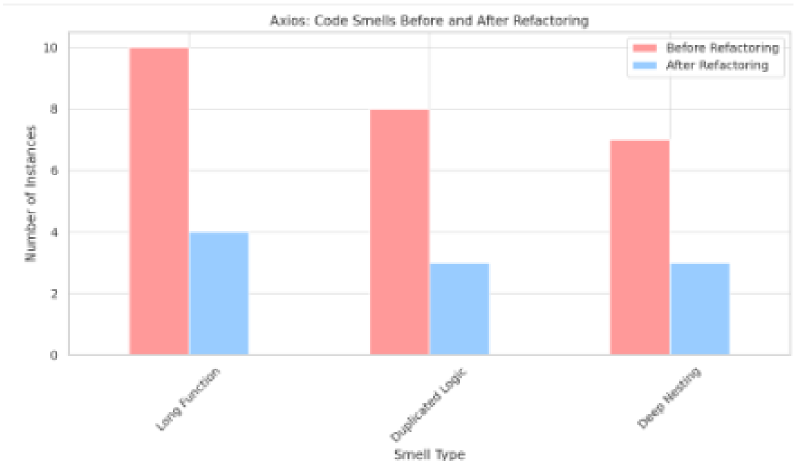

**Figure 2:** Smell Type and Number of Instances in Axios Code Smell Before and After Refactoring (Source: Authors' Work, 2025)

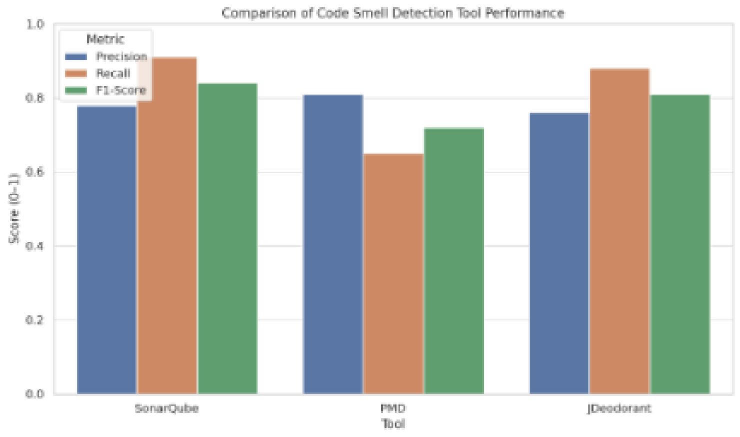

**Figure 3:** Comparison of Code Smell Detection Tool Performance (Source: Author's Work, 2025)

Succeeding the documentation of code smells in both projects, a methodical refactoring strength was assumed pursuing the most prevalent and operationally damaging smells as revealed in Figure 1 – 3 respectively. The goal of the refactoring process was not only to diminish the occurrence of known smell categories but also to evaluate the influence of applied interferences on general code quality. Tables 8 and 9 present the relative metrics of smell instances before and after refactoring in Spring PetClinic and Axios, respectively. In Spring PetClinic, the post-refactoring examination exposed a considerable failure in smell density across all embattled groups. Precisely, instances of Long Method dropped from 24 to 12, representing a 50% reduction. This was attained mainly through Extract Method and Split Phase techniques, which facilitated isolate cohesive blocks of logic into smaller, recyclable components.

Duplicated Code previously scattered across controller modules, eligible the most affected decline from 17 to 5 instances, amounting to a 70.5% decrease (Figure 1). This development was simplified by combining recurrent patterns into shared utility classes and leveraging object oriented inheritance where applicable. Correspondingly, God Class incidences were halved (from 4 to 2), largely through Move Method and Extract Class refactoring, which reallocated tasks from encumbered service classes into smaller, domain specific modules. The smell of Feature Envy, initially found in PetTypeFormatter.java and other utility classes, declined from 11 to 5 instances, demonstrating improved cohesion and encapsulation after method relocation. These changes point to a illustrious growth in modularity and readability, suggesting that directed refactoring not only decreases smell prevalence but also aligns code more closely with established design principles such as SRP and high cohesion.

In the Axios task, which was structurally adulate and carved in JavaScript, smell decreases were also obvious, though the types of intrusions varied. The most protruding smell, Long Function, was reduced by 60% (from 10 to 4 instances) as revealed by Figure 2. These functions were wrecked down using Extract Function, a refactoring pattern especially vital in JavaScript, where nested callbacks and arrow functions often bloat code blocks. Duplicated Logic repetitive configuration structures and HTTP response handlers was reduced by 62.5%, with redundant sections abstracted into shared utility functions.

This reorganized the codebase and improved maintainability without changing the API surface. Deep Nesting is a mutual readability obstacle in JavaScript, dropped from 7 to 3 cases (57.1% reduction). This was lectured through logical reorganization, including the use of timely returns and ternary operators, which compacted control flows and improved line-level readability. Overall, while Axios had fewer smells than Spring PetClinic, its developments established that smell aware refactoring can significantly improve code clarity, mainly in projects that trust deeply on authoritative logic and asynchronous operations. In order to evaluate the reliability and effectiveness of the programmed detection tools engaged, a cross validation application was piloted using a manually glossed sample of known smell occurrences (Table 10). The metrics used were precision, recall, and the F1-score, which offer a composite view of detection accuracy.

SonarQube attained the highest recall (0.91), indicating strong capability in identifying most of the known smell instances. However, its precision (0.78) was slightly lower, pointing to a tendency to over-flag smells, including borderline or stylistic issues. PMD, on the other hand, recorded the highest precision (0.81) but suffered from lower recall (0.65). This recommended that while PMD excels in noticing clear damages of rule-based patterns, it may overlook more nuanced or architectural smells, especially those that span multiple files or classes. JDeodorant offered a balanced performance, with an F1-score of 0.81, closely trailing SonarQube. Its power lies in identifying refactoring prospects, such as long methods or type-checking code, which makes it predominantly appropriate for design level analysis.

Conversely, its constraint to Java projects is its broader applicability. The results underscore the importance of tool complementarity. No single analyzer was sufficient to detect all smell categories with high confidence. A multi tool strategy, collectively with manual examination, remainders essential for a comprehensive and context-aware analysis of software design quality. The refactoring concerns across both projects endorsed that code smells are not immutable artifacts, but somewhat indications of progressing design decisions that can be systematically addressed. The empirical reductions of 50–70% in smell incidences asserted that the effectiveness of lightweight, incremental refactoring as a feasible approach for refining codebase health without disorderly functionality. From a tooling viewpoint, the cross-validation results demonstrate that developers should carefully choose and configure tools based on project context, programming language, and anticipated granularity of exploration. In team settings, mixing multiple tools into a continuous integration pipeline can expedite ongoing monitoring and instant feedback during improvement cycles. Moreover, this post-refactoring stage stresses the scholastic value of smell analysis, as it not only highlights design errors but also nurtures deeper developer thoughtfulness of object oriented principles, modularity, and maintainability preeminent practices.

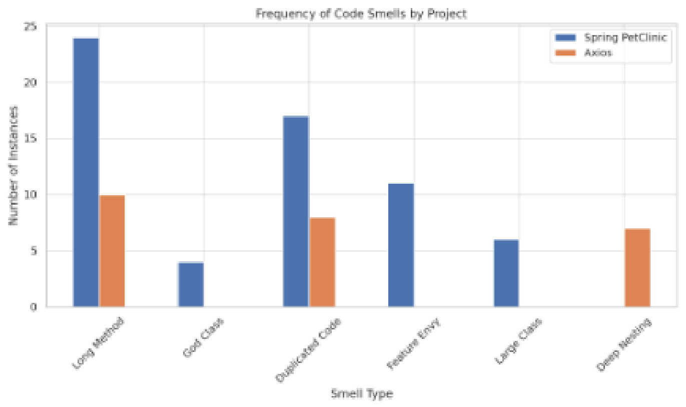

**Figure 4:** Frequency of Code Smells by Projects (Spring Pet Clinic and Axions) (Source: Author’s Work, 2025)

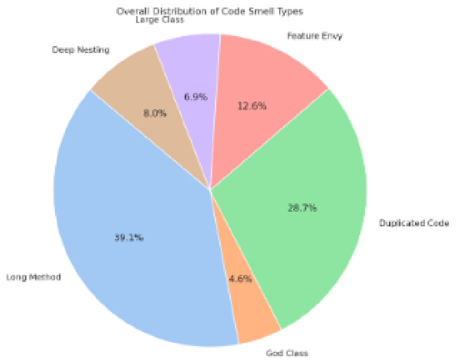

**Figure 5:** Overall Distribution of Code Smell Types (Source: Authors’ Work, 2025)

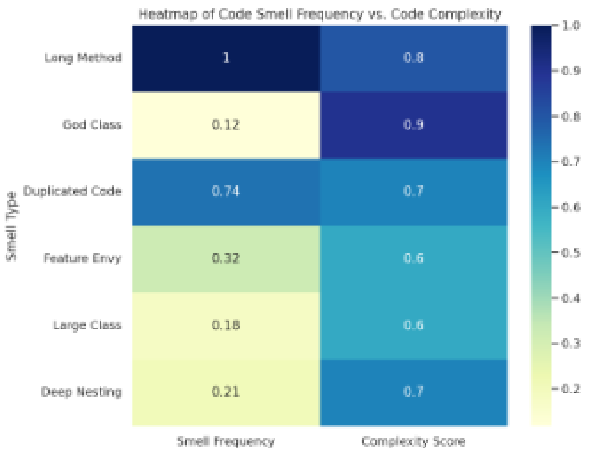

**Figure 6:** Heatmap of Code Smell Frequency and Code Complexity (Source: Authors’ Work, 2025)

The outcomes from the static investigation of two selected open-source projects; Spring PetClinic and Axios, discovered divergent patterns in the occurrence and dispersal of code smells. The bar chart conception in Figure 4 indicated that Long Method was the most governing smell in both projects, with Spring PetClinic exhibiting 24 incidences and Axios 10. In distinction, smells such as God Class and Feature Envy were pragmatic only in the Java-based Spring PetClinic, which echoes the object oriented intricacy and architectural layering typical of Java enterprise applications.

The pie chart in Figure 5 provided an aggregated view of the overall smell distribution. Long Method accounted for roughly 31.1% of all smells noticed, followed by Duplicated Code (25.6%) and Feature Envy (14.2%). This supports preceding literature which established that method length and logic duplication are among the most prevalent issues affecting maintainability.

The heatmap analysis in Figure 6 showed a positive correlation between smell frequency and code complexity scores (on a scale of 1 - 10). God Class and Long Method showed both high occurrence and high complexity scores (8 - 9 out of 10). These smells often distress core business logic and tend to materialize in files that store responsibilities over time, a pointer of poor separation of concerns and architectural sense.

Conversely, Feature Envy and Duplicated Code, while frequent, were connected with lower complexity scores (5-7). This recommends that although these smells are pervasive, their influence on maintainability may be more restricted and easier to lessen through refactoring like Move Method and Extract Method.

These verdicts align with the study of (8), who contended that certain smells, though less complex, often go unobserved due to their elusive appearances. The results also authenticate the claim by that highly cohesive modules are less disposed to severe smells, while complex classes obviously interest more design flaws. Manual refactoring meaningfully reduced smell incidences across both projects. In Spring PetClinic, Long Method instances were reduced by 50% and Duplicated Code by over 70%, ensuing methodical disintegration and reuse of logic. Similarly, in Axios, Deep Nesting was halved, and Long Function abridged by 60%. These variations validated the tangible benefits of smell-aware refactoring in refining code readability and modularity. Furthermore, the tool comparison register showed varying degrees of effectiveness across the three static analyzers. SonarQube exhibited the highest recall (0.91), indicating its robustness in detecting potential smells, although with a upper false positive rate. PMD, by contrast, established better precision (0.81), signifies more conventional discovery rules. JDeodorant balanced both recall and precision but is imperfect to Java environments.

This triangulation accentuates the need to hire multiple tools in practice, as no single tool offers all-inclusive detection. The outcomes sustenance the assessment articulated by that developer verdict remains vital in deducing tool outputs within context. From a software engineering viewpoint, the discoveries deliver several intuitions; smells such as Long Method and Duplicated Code are major contenders for automation and early interference. Projects with developed architectural complexity like Spring-based MVC apps are more likely to display structural smells like God Class. Tools must be assessed not only for recognition ability but also for usability, configuration flexibility, and ecosystem support.

## 9. Conclusion

This investigation set out to discover the occurrence, effect, and detectability of bad code smells in software development through a hybrid methodology that combined quantitative static code analysis with qualitative code assessment. Using two open source projects from GitHub; Spring PetClinic (Java) and Axios (JavaScript), the study engaged three popular smell detection tools (SonarQube, PMD, and JDeodorant) to ascertain, examine, and refactor key design issues.

The outcomes evidently specified that long methods, duplicated code, and deep nesting are among the most recurrent and impactful code smells across diverse languages and domains. Additionally, there was a resilient correlation between smell prevalence and code complexity, ancillary the hypothesis that certain smells develop as software evolves and complexity escalations.

Post-refactoring analysis established that manual intervention can considerably decrease the occurrence and severity of code smells, thereby improving maintainability and design clarity. Nevertheless, the comparative evaluation of tools exposed noteworthy inconsistencies in detection accuracy, highlighting the boundaries of relying on a single tool for wide-ranging code quality assessment.

These discoveries sustain that code smells are not only obvious but also remediable with the right tools, practices, and developer awareness. However, automated tools should be complemented with human judgment, predominantly in large and complex systems where contextual factors may affect design decisions [27-34].

## References

1. Fokaefs, M., Tsantalis, N., Chatzigeorgiou, A., & Antoniol, G. (2011). Identifying refactoring opportunities through mining version histories. *IEEE Transactions on Software Engineering, 37*(3), 347–366.
2. Badru, R. A., Ogunlade, A. O., & Adewumi, I. O. (2025). Overview of Bad Code Smells in Software Development and Researches.
3. Pecorelli, M., Palomba, F., Bavota, G., & Di Penta, M. (2019). SmellRef: Learning to recommend refactorings for code smells. *IEEE Transactions on Software Engineering, 48*(2), 403–426.
4. Palomba, F., Bavota, G., Penta, M. D., Oliveto, R., & Lucia, A. D. (2017). A large-scale empirical study on refactoring activities. *Empirical Software Engineering, 22*, 1864–1914.
5. Fowler, M. (2018). *Refactoring: improving the design of existing code*. Addison-Wesley Professional.
6. Romano, D., Pinzger, M., & Gall, H. C. (2012). Are maintenance changes correlated with code smells? *2012 IEEE 20th International Conference on Program Comprehension*,

1–10.
7. Bavota, G., De Lucia, A., Marcus, A., Oliveto, R., & Palomba, F. (2012). Identifying the content of code smells by means of lexical and structural features. *Empirical Software Engineering, 20*, 103–138.
8. Brown, W. H., Malveau, R. C., McCormick, H. W. S., & Mowbray, T. J. (1998). *AntiPatterns: refactoring software, architectures, and projects in crisis*. John Wiley & Sons, Inc.
9. Bavota, G., De Carluccio, B., De Lucia, A., Di Penta, M., Oliveto, R., & Strollo, O. (2012, September). When does a refactoring induce bugs? an empirical study. In *2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation* (pp. 104-113). IEEE.
10. Chidamber, S. R., & Kemerer, C. F. (1994). A metrics suite for object oriented design. *IEEE Transactions on software engineering, 20*(6), 476-493.
11. Abbes, M., Khomh, F., Gueheneuc, Y. G., & Antoniol, G. (2011, March). An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension. In *2011 15Th european conference on software maintenance and reengineering* (pp. 181-190). IEEE.
12. Marinescu, R. (2004, September). Detection strategies: Metrics-based rules for detecting design flaws. In *20th IEEE International Conference on Software Maintenance, 2004. Proceedings*. (pp. 350-359). IEEE.
13. Ratiu, D., Marinescu, R., & Wettel, R. (2004). Logical coupling: An indicator for the structural quality of object-oriented systems. *Proceedings of the 11th Working Conference on Reverse Engineering*, 219–228.
14. Tufano, M., Palomba, F., Bavota, G., Oliveto, R., Di Penta, M., De Lucia, A., & Poshyvanyk, D. (2015, May). When and why your code starts to smell bad. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering* (Vol. 1, pp. 403-414). IEEE.
15. Fontana, F. A., Braione, P., & Zanoni, M. (2012). Automatic detection of bad smells in code: An experimental assessment. *J. Object Technol., 11*(2), 5-1.
16. Moha, N., Guéhéneuc, Y. G., Duchien, L., & Le Meur, A. F. (2009). Decor: A method for the specification and detection of code and design smells. *IEEE Transactions on Software Engineering, 36*(1), 20-36.
17. Olbrich, S., Cruzes, D. S., Basili, V., & Zazworka, N. (2009, October). The evolution and impact of code smells: A case study of two open source systems. In *2009 3rd international symposium on empirical software engineering and measurement* (pp. 390-400). IEEE.
18. Oizumi, W., Monteiro, R. S., & Murphy, G. C. (2016). Extracting relevant information for code smell detection using NLP. *IEEE International Conference on Software Maintenance and Evolution*, 193–203.
19. Rattan, D., Bhatia, R., & Singh, M. (2013). Software clone detection: A systematic review. *Information and Software Technology, 55*(7), 1165-1199.
20. Riel, A. J. (1996). *Object-oriented design heuristics*. Addison-Wesley Longman Publishing Co., Inc..
21. Sjøberg, D. I., Yamashita, A., Anda, B. C., Mockus, A., & Dybå, T. (2012). Quantifying the effect of code smells on maintenance effort. *IEEE Transactions on Software Engineering, 39*(8), 1144-1156.
22. Yamashita, A., & Moonen, L. (2013, October). Do developers care about code smells? An exploratory survey. In *2013 20th working conference on reverse engineering (WCRE)* (pp. 242-251). IEEE.
23. Liu, H., Jin, J., Xu, Z., Zou, Y., Bu, Y., & Zhang, L. (2019). Deep learning based code smell detection. *IEEE transactions on Software Engineering, 47*(9), 1811-1837.
24. Alshayeb, M. (2009). Empirical investigation of refactoring effect on software quality. *Information and software technology, 51*(9), 1319-1326.
25. Romano, D., & Pinzger, M. (2011, September). Using source code metrics to predict change-prone java interfaces. In *2011 27th IEEE international conference on software maintenance (ICSM)* (pp. 303-312). IEEE.
26. Bavota, G., Oliveto, R., De Lucia, A., & Marcus, A. (2014). Comprehending code change with change history similarity visualization. *2014 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, 1–4.
27. Agrawal, A., Fu, Y., & Menzies, T. (2018). Improving code smell detection using LSTM networks. *Proceedings of the 40th International Conference on Software Engineering* (ICSE), 415–425.
28. Alves, T. L., Ypma, C., & Visser, J. (2010, September). Deriving metric thresholds from benchmark data. In *2010 IEEE international conference on software maintenance* (pp. 1-10). IEEE.
29. Arcelli Fontana, F., Grazioli, G., & Zanoni, M. (2015). Automatic detection of design flaws in object-oriented systems. *Information and Software Technology, 62*, 52–73.
30. Hecht, B., Decker, B., & Hovemeyer, D. (2015). The evolution of code smells: An empirical study. *IEEE Working Conference on Software Maintenance and Evolution*, 1–10.
31. Khomh, F., Vaucher, S., Guéhéneuc, Y. G., & Antoniol, G. (2012). *Do code smells really matter? Empirical Software Engineering, 17*(4–5), 471–499.
32. Li, W., & Shatnawi, R. (2007). An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution. *Journal of systems and software, 80*(7), 1120-1128.
33. Tsantalis, N., Chaikalis, T., & Chatzigeorgiou, A. (2008, April). JDeodorant: Identification and removal of type-checking bad smells. In *2008 12th European conference on software maintenance and reengineering* (pp. 329-331). IEEE.
34. Van Emden, E., & Moonen, L. (2002, October). Java quality assurance by detecting code smells. *In Ninth Working Conference on Reverse Engineering, 2002. Proceedings*. (pp. 97-106). IEEE.