**An Empirical Study on Impact of Code Smells on Software Maintainbility**

***Submitted By***

**Pavan Kumar Gaddam**

**Sai Teja Singamaneni**

**Siva Chaitanya Jonnalagadda**

**Swamy Gorla**

**Yamini Dondla**

|  |
| --- |
|  |

# ABSTRACT

One important quality trait that influences the long-term flexibility and success of software systems is software maintainability. It's well accepted that code smells, which point to possible design errors or bad coding techniques, make a system harder to maintain. In this study, we examine ten open-source Java projects to objectively assess the effect of code smells on program maintainability. Of these projects, five have a high amount of code smells, whereas the other five are comparatively free of them. Code smells are found via tools like JDeodorant, and metrics like Lack of Cohesion in Methods (LCOM), Weighted Methods per Class (WMC), and Coupling Between Objects (CBO) are used to evaluate maintainability.

According to our research, there is a noticeable link between code smells and less maintainable software. Compared to their cleaner equivalents, projects with code smells exhibit higher coupling, greater complexity, and worse cohesion, making them more difficult to maintain and develop. These results emphasize the significance of timely reworking to improve maintainability and routine code quality assessments. This study highlights the importance of following clean coding techniques and offers software developers practical insights.

**Keywords**

Code Smells, Software Maintainability, Coupling Between Objects, Weighted Methods per Class, Lack of Cohesion in Methods, Refactoring, Java Projects

# INTRODUCTION

One essential feature that has a direct impact on the long-term viability and adaptability of software systems is software maintainability. High maintainability guarantees that software may be effectively comprehended, altered, and expanded, saving time and money during development. However, code smells—which are signs of more serious structural problems in the codebase—are frequently introduced by bad coding methods and subpar design. Although they are not direct problems, these smells make software maintenance more difficult and are closely linked to technical debt [1]. According to empirical research, code smells can have a detrimental effect on important maintainability indicators, so it's critical to proactively fix them to guarantee software quality over the long run [2].

Through an analysis of ten open-source Java projects, this study seeks to scientifically assess the impact of code smells on program maintainability. Using well-known maintainability measures like Lack of Cohesion in Methods (LCOM), Weighted Methods per Class (WMC), and Coupling Between Objects (CBO), five projects with serious code smells and five that are comparatively clean are compared.

The format of this report is as follows: The methodology for detecting and evaluating code smells and extraction of object-oriented metrics is explained in Section 2, along with the metrics and tools utilized. The study's findings are presented and discussed in Section 3, emphasizing important trends and findings. Section 4 lists potential threats to validity and the steps taken to mitigate them. Section The paper is concluded in Section 5 with a summary of the findings, their implications, and possible directions for future study.

1. **METHODOLOGY AND TOOLS USED**

This study uses three important metrics—Lack of Cohesion in Methods (LCOM), Weighted Methods per Class (WMC), and Coupling Between Objects (CBO)—as part of the Goal-Question-Metric (GQM) strategy to assess how code smells affect program maintainability. The following steps are part of the methodology:

**Research Goal:** Using the CK tool for metric extraction, 10 open-source Java projects' object-oriented metrics—Lack of Cohesion in Methods (LCOM), Weighted Methods per Class (WMC), and Coupling Between Objects (CBO)—will be analyzed to experimentally assess the effect of code smells on software maintainability.

**Research Questions:**

We will proceed with the futher research by keeping in mind to answer the following questions.

**RQ1**: How do the LCOM, WMC, and CBO metrics differ between Java projects with code smells and those without?

This question assesses directly how the presence or absence of code smells affects maintainability-related metrics. This question helps determine whether projects with code smells show measurable declines in maintainability by comparing the two groups' cohesiveness (LCOM), complexity (WMC), and coupling (CBO). It discusses the focus of our research, which is the connection between quantifiable quality metrics and code smells.

**RQ2**: How does the density of code smells in Java projects influence the maintainability metrics (LCOM, WMC, CBO) across varying project sizes and complexities?

This question broadens the focus by investigating the relationship between maintainability metrics and the quantity or density of code smells (number of smells per project size) in projects of various sizes and complexity. It offers a detailed viewpoint on whether code smells significantly damage larger or more complex projects, providing information on whether code smells are more dangerous in particular situations. By linking the distribution of code smells to the variation in maintainability, this helps address a more general question in our research and allows for more focused interventions.

**Metrics Used:**

**Coupling Between Objects (CBO)**: CBO quantifies how much a class depends on other classes. Strong interdependencies inside the class are indicated by a high CBO score, which may make the software more complex.

**Maintainability Significance of CBO**: High coupling complicates the software's comprehension, testing, and modification. Maintainability suffers when modifications are made to a strongly connected class since there is a greater chance of unforeseen ripple effects.

**Weighted Methods per Class (WMC):** WMC determines the total complexity of all methods in a class. The difficulty of each method is frequently determined using cyclomatic complexity or, in the absence of a complexity metric, is simply tallied as 1.

**Maintainability Significance Of WMC**: A class with a high WMC value has too many or too complicated methods, which makes it more difficult to test, debug, and change. In general, lower WMC scores indicate that a class is easier to maintain and has greater cohesiveness.

**Lack of Cohesion in Methods (LCOM)**: LCOM gauges how closely connected methods are to one another inside a class. Usually, it evaluates how many methods access distinct sets of instance variables. Lower cohesiveness is indicated by higher values.

**Maintainability Significance of LCOM**: A class with low coherence is likely completing several unrelated tasks, which makes it challenging to comprehend, test, and maintain. A cohesive class adheres to good design standards by concentrating on a single duty.

**Research Methodology:**

We have followed the below steps to proceed with our research.

**Phase 1: Setting critera for Project Selection and Repository Cloning**

To choose projects that are appropriate for examining how code smells affect software modularity, we used the following criteria in this study:

**i. Project Size Requirement:** For the chosen projects, we have a minimum size requirement of 5,000 lines of code (LoC). This criterion guarantees that the programs are big enough and sophisticated enough to possibly display several kinds of code smells. Examining how code smells might impact the overall structure and maintainability of software is made easier by the fact that a larger codebase is more likely to experience maintainbility difficulties.

**ii.Project Age Requirement:** We stipulated that the chosen projects must be at least six years old to capture the long-term consequences of code smells on software maintenance. This guarantees that the projects have experienced several cycles of development and maintenance, increasing the likelihood that they may display code smells that have changed over time. Understanding the long-term effects of inadequate modularity and the persistence of code smells in real-world systems can be gained by examining earlier projects.

**iii.** **Involvement of Developers**

We stated that the codebase for the projects must be contributed to by a minimum of three developers. Because a larger development team typically provides a range of coding styles, design decisions, and potential disputes, this criterion is crucial. These elements impact program modularity and the manifestation of code smells. Because different coding styles and design approaches may result in code smells, having numerous contributors guarantees that the project represents a collaborative atmosphere**.**

Based on this ceiteria we have filtered out the projects and have chosen 10 projects from the GitHub.In this phase we have cloned each repository locally. In this selection criteria we also made sure to include projects and without code smells. For this we have looked at various factors like Number of Open Issues, commits before finalizing the selection.

Please refer to the Table 1 for more detailed information of each selected project.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Project Name** | **Description** | **Size (LoC)** | **Age** | **Contributors** |
| zhisheng17/flink-learning | A comprehensive Flink learning repository covering various concepts, examples, and use cases [4]. | 43,490 | 6+ years | 6 |
| topjohnwu/libsu | A complete solution for apps using root permissions [5]. | 23,643 | 6+ years | 8 |
| ixrjog/opscloud4 | A cloud operations and maintenance platform [6] | 18,455 | 6+ years | 3 |
| tianshiyeben/wgcloud | Linux system monitoring and operational tools, offering advanced monitoring and visualization [7] | 46,024 | 6+ years | 3 |
| iluwatar/java-design-patterns | A repository implementing design patterns in Java [8] | 28,281 | 6+ years | 268 |
| alibaba/druid | A database connection pool monitoring tool [9] | 65,997 | 6+ years | 169 |
| alibaba/arthas | A Java diagnostic tool by Alibaba [10] | 28,983 | 6+ years | 150 |
| NLPchina/elasticsearch-sql | Extends Elasticsearch to support SQL queries [11]. | 55,020 | 6+ years | 31 |
| wxiaoqi/Spring-Cloud-Platform | A platform for building cloud-based Spring applications [12]. | 11,000 | 6+ years | 4 |
| ityouknow/spring-boot-examples | Examples showcasing various Spring Boot features [13]. | 28,603 | 6+ years | 8 |

***Table 1: Detailed information of the selected projects***

**Phase 2: Tool Selection and Metrics Calculation**

We have opted to run CK Metrics for each project to understand different aspects of maintainbility and used Jdeodrant to detect the code smells.

**4.1 CK Metrics Suite Analysis (CKMAT):** Six important object-oriented metrics that are crucial for assessing and comprehending software system architecture make up the CK Metrics Suite [3]. Some of these metrics are Number of Children (NOC), which counts the number of subclasses; Coupling Between Objects (CBO), which counts the degree of inter-class dependencies; Response For a Class (RFC), which counts the total number of methods that a class can call; Weighted Methods per Class (WMC), which determines the complexity of a class based on its methods; Depth of Inheritance Tree (DIT), which evaluates the depth of inheritance of a class; and Lack of Cohesion in Methods (LCOM), which counts the cohesion of methods within a class. When taken as a whole, these metrics offer insightful information on software complexity, maintainability, and the possibility of enhancing modularity through an examination of the system's architecture and relationships.

**4.2 Code Smell Detection Using JDeodorant:**

To find different kinds of code smells, we used Jdeodorant [14], an Eclipse plugin that is frequently used for static code analysis, in this study. The program is excellent at identifying problems that impair code quality and make it more difficult to maintain, like Feature Envy, Long Method, God Class, Duplicated Code, and Type Checking. Software modularity can be seriously harmed by these code smells, which usually result in problems like tightly linked code, poor cohesion, and needless complexity. JDeodorant helps developers enhance software structure, maintainability, and overall performance by offering insightful analysis of these problems and refactoring recommendations. JDeodorant is essential to maintaining the software's scalability and ease of maintenance over time by detecting and fixing these code smells.

**Phase 3: Analysis and Interpretation of Metrics**

In this from the extracted metrics we have analyzed all the metrics and have drawn conclusion from the data by effectively answering our research questions. We have filtered out the unnessary information and focussed on important metrics that affect maintainbility and which can show correlation to the code smells.

**3 RESULTS AND DISCUSSIONS**

From the Jdeodrant tool whave found the below code smells

|  |  |
| --- | --- |
| **Project Name** | **Identified Code Smells** |
| zhisheng17/flink-learning | No Code Smells |
| topjohnwu/libsu | No Code Smells |
| ixrjog/opscloud4 | No Code Smells |
| tianshiyeben/wgcloud | No Code Smells |
| iluwatar/java-design-patterns | No Code Smells |
| alibaba/druid | God Smells, Long Method |
| alibaba/arthas | God Smells, Long Method |
| NLPchina/elasticsearch-sql | LongMethod, Duplicated Code, God Class |
| wxiaoqi/Spring-Cloud-Platform | LongMethod, God Class and  Feature Envy |
| ityouknow/spring-boot-examples | Long Method |

***Table 2: Code Smells across different projects***

We have used a script that calculates weighted averages for the metrics CBO, WMC, and LCOM to respond to RQ1, considering each project's complexity by utilizing WMC as a weight. This makes it possible to compare projects more accurately by considering their size and number of methods. For every metric, the script also computes class-level statistics like the median, mode, and 75th percentile. To determine typical values, the most common occurrences, and the upper range of the metrics for both projects with and without code smells, these statistics offer insights into the central tendency and distribution of the metrics.

Bar charts that contrast metrics between projects with and without code smells are used to display the results. While the class-level statistics provide a more in-depth understanding of the metrics' distribution, the weighted averages provide a normalized view of the metrics. When taken as a whole, these computations and graphics successfully show how the two groups' CBO, WMC, and LCOM differ from one another, offering important new information about the effects of code smells on various software measures.

**RQ1** **Results**: How do the LCOM, WMC, and CBO metrics differ between Java projects with code smells and those without?

***Figure 1: Comparison of CBO for projects with and without code smells.***

A graph showing the number of bars

Description automatically generated with medium confidence

***Figure 2: Comparison of WMC for projects with and without code smells.***

A graph with different colored bars

Description automatically generated

A graph with blue and orange bars

Description automatically generated

***Figure 3: Comparison of LCOM for projects with and without code smells.***

From the Figure 1 we can observe that Projects such as elastic-search-sql project has higher CBO and the least value of CBO is displayed by java design patterns project. From this trend we have observed that Projects which have code smells have higher CBO and clearly the least CBO valued project is the one which is not having any code smells.

If we observe Figure 1, we can also conclude that Classes having code smells have higher CBO Values compared to the projects which are not having code smells.

As we see Figure 2, we have observed that WMC values of projects having code smells is higher than WMC values of projects which are not having code smells. Same as CBO the elastic-search-sql project has higher WMC and java-design-patterns have lower WMC.The distribution of WMC is uniform among the projects which are not having code smells.

If we loot at the Figure 3, elasticsearch-sql project and druid projects has higher lcom and these are the projects that are having code smells. And the java-design pattern project which is not having any code smell has lower values of LCOM.

Code smells have an apparent impact on software maintainability, according to the findings from the figures and the examination of the CBO, WMC, and LCOM metrics. According to the study, Java projects with code smells typically have higher LCOM (Lack of Cohesion of Methods), WMC (Weighted Methods per Class), and CBO (Coupling Between Objects) scores. Projects with more intricate interactions between classes and methods, such as `elasticsearch-sql`, exhibit higher CBO and WMC, which can have a detrimental effect on maintainability. However, code-smell-free projects, like `java-design-patterns`, have lower values for these metrics, indicating fewer coupling and complexity as well as improved maintainability.

Projects with code smells have higher values of these measures, which indicates a decline in software quality and makes maintenance more difficult. Classes with higher CBO values are more interconnected, which makes it harder to comprehend and change the system. Likewise, classes with higher WMC values are thought to have more methods, which makes testing and maintenance more difficult. Poor cohesion between methods inside a class is indicated by higher LCOM values, which can result in a lack of modularity and make software more challenging to maintain over time. In line with the general knowledge of the connection between code quality and software maintainability, this empirical study shows that code smells have a detrimental effect on the maintainability of Java projects.

**RQ2 Results**: How does the density of code smells in Java projects influence the maintainability metrics (LCOM, WMC, CBO) across different projects

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Project Name** | **Density of Code Smells** | **Aggregated CBO** | **Aggregated WMC** | **Aggregated LCOM** |
| zhisheng17/flink-learning | 0 | 2480 | 2307 | 1926 |
| topjohnwu/libsu | 0 | 493 | 1338 | 3553 |
| ixrjog/opscloud4 | 0 | 21152 | 13249 | 19255 |
| tianshiyeben/wgcloud | 0 | 713 | 1496 | 4076 |
| iluwatar/java-design-patterns | 0 | 568 | 628 | 471 |
| alibaba/druid | 6 | 29171 | 75052 | 1327627 |
| alibaba/arthas | 6 | 21152 | 13249 | 19255 |
| NLPchina/elasticsearch-sql | 15 | 2480 | 2307 | 1926 |
| wxiaoqi/Spring-Cloud-Platform | 4 | 865 | 1450 | 10204 |
| ityouknow/spring-boot-examples | 1 | 1451 | 975 | 1545 |

***Table 3: Density of Code Smells across 10 projects***

**Code Smells vs. Coupling Between Objects (CBO):** There is a discernible pattern in the Coupling Between Objects (CBO) of projects with different concentrations of code smells. Projects without code smells, for example, show a broad range of CBO values, including 2480 for `zhisheng17/flink-learning`and 21152 for `ixrjog/opscloud4`, suggesting that coupling is influenced by variables other than code smells. The CBO values of`alibaba/druid` and `alibaba/arthas`, for example, are significantly high at 29171 and 21152, respectively, for projects with six code smells. In a similar vein, CBO typically grows as code smell density does. Projects with lesser density, such one code smell (`ityouknow/spring-boot-examples`) or four code smells (`wxiaoqi/Spring-Cloud-Platform`), have CBO values of 1451 and 865, respectively, whilst a project with 15 code smells (`NLPchina/elasticsearch-sql`) has a CBO of 2480 (see Fiure 4).Based on the interdependencies between modules, this connection implies that code smells lead to greater coupling between classes, which makes projects more difficult to manage and restructure.

***Figure 4: Density of Code Smells vs CBO***

**Code Smells vs. Weighted Methods per Class (WMC):** There is a moderate to strong positive link between code smell density and Weighted Methods per Class (WMC). The WMC values of code-smell-free projects vary; for instance, ixrjog/opscloud4 has a WMC of 13249, whereas zhisheng17/flink-learning has a WMC of 2307(see Figure 5). The WMC values for projects with six code smells, such Alibaba/druid and Alibaba/arthas, are remarkably high, at 75052 and 13249, respectively. Additionally, WMC tends to rise in tandem with the quantity of code smells. Ityouknow/spring-boot-examples with one code smell has a WMC of 975, wxiaoqi/Spring-Cloud-Platform with four code smells has a WMC of 1450, and NLPchina/elasticsearch-sql with fifteen code smells has a WMC of 2307.According to this trend, classes become more complex as the density of code smells increases. This is frequently the result of poorly designed code, such as redundant logic or too complicated methods. As classes get more complex and difficult to comprehend and alter, elevated WMC values are a sign of decreased maintainability.

***Figure 4: Density of Code Smells vs WMC***

**Code Smells vs. Lack of Cohesion in Methods (LCOM):**

An examination of the relationship between code smell density and Lack of Cohesion in Methods (LCOM) shows a strong positive association. There is significant variation in LCOM values across code-smell-free projects; for example, topjohnwu/libsu has an LCOM of 3553, whereas iluwatar/java-design-patterns has a far lower LCOM of 471. On the other hand, projects with six code smells, such alibaba/druid and alibaba/arthas, exhibit significantly high LCOM scores of 19255 and 1327627, respectively. Moreover, the LCOM grows in proportion to the density of code smells. Ityouknow/spring-boot-examples with one code smell has an LCOM of 1545, wxiaoqi/Spring-Cloud-Platform with four code smells has an LCOM of 10204, and NLPchina/elasticsearch-sql (see Figure 5) with fifteen code smells has an LCOM of 1926. These results imply that a lack of cohesiveness among methods within classes is linked to a larger density of code smells, leading to fragmented and inconsistent class structures. Maintainability is hampered by such fragmentation since it makes the codebase more challenging to comprehend, explore, and reuse efficiently.

***Figure 5: Density of Code Smells vs LCOM***

The thorough examination of the supplied data clearly shows that, across various Java projects, the density of code smells negatively impacts three important maintainability metrics: Lack of Cohesion in Methods (LCOM), Weighted Methods per Class (WMC), and Coupling Between Objects (CBO). Higher CBO is linked to a higher density of code smells, indicating tighter coupling between classes and making maintenance and refactoring more difficult because of increasing interdependencies. In a similar vein, a high WMC brought on by multiple code smells indicates a more complex class, which makes the code more difficult to understand and alter. Furthermore, a higher density of code smells is associated with higher LCOM, which suggests that class methods are less cohesive and result in poorly organized, disconnected classes. Together, these results show that projects with higher densities of code smells typically have more closely related classes, more complex methods, and less coherent method structures—all of which significantly reduce the software's maintainability.

**4 THREATS TO VALIDITY**

**4.1 Internal Validity**

The degree to which the study's findings may be ascribed to the modification of the independent variables as opposed to other confounding variables is known as internal validity. In this research:

* **Confounding variables**: Unconsidered elements that may affect the projects' maintainability include developer experience, workplace culture, and development tools.
* **Risks Associated with Tool Limitations**: The internal validity may be impacted by the limits of the tools utilized (CK Metrics for metric extraction and JDeodorant for code smell detection), as these techniques might not fully capture the range of problems influencing software maintainability.
* **Selection Bias**: The availability of open-source projects that satisfied specific requirements was necessary for the investigation. Due to the possibility of bias, this selective sampling could distort the results and lower internal validity.

**4.2 External Validity**

The degree to which study results can be extrapolated to different contexts, demographics, or eras is known as external validity. In this research:

* **Project Representativeness**: The study may not be entirely representative of the larger collection of Java projects or other programming languages because it was based on a particular group of open-source Java projects that satisfied requirements.
* **Tool Generalizability**: It's possible that not all software projects or development environments can use the JDeodorant and CK Metrics tools employed in this study. The study's findings may therefore be applicable to open-source Java projects, but they might not be for other kinds of projects or programming languages.

**4.3 Construct Validity**

The extent to which the instruments and metrics truly measure the idea they are supposed to measure is known as construct validity. In this research:

* **Code Smells and Software Maintainability**: While CK Metrics and JDeodorant are suitable tools for assessing software metrics and detecting code smells, respectively, they might not fully capture all facets of maintainability. Metrics like WMC (Weighted Methods per Class) and CBO (Coupling Between Objects), for instance, offer some information on class structure but miss other crucial aspects of maintainability, including developer familiarity or ease of refactoring.
* **Interpretation of Metrics**: Although the study depends on software metrics, it's possible that these don't accurately reflect the software's general maintainability. The study's construct validity may be limited because it did not examine variables that could affect maintainability, such as developer expertise, team dynamics, or even the application of contemporary refactoring techniques.

**4.4 Conclusion Validity**

The degree to which the data support the study's conclusions is known as conclusion validity. In this research.

* **Tool Sensitivity**: The findings about the influence of code smells on maintainability may be partial or biased due to the limits of the tools employed to identify code smells and extract metrics.
* Interpretation of Results: Although the study indicates that specific metrics and code smells are related to maintainability, the conclusions are less definite because other factors (such as developer talent or team dynamics) were not taken into consideration. Furthermore, the validity of the conclusions may be further undermined if the chosen projects were not entirely typical of Java projects overall.

**5 CONCLUSIONS AND FUTURE RESEARCH**

By analyzing ten open-source projects using recognized object-oriented metrics, this study offers a thorough investigation of how code smells affect the maintainability of Java projects. The results unequivocally show that code smells significantly impair software systems' capacity to be maintained. Projects with code smells, like those found in druid, arthas, and elasticsearch-sql, typically have higher scores on maintainability metrics like Lack of Cohesion in Methods (LCOM), Weighted Methods per Class (WMC), and Coupling Between Objects (CBO). These findings suggest that because of their greater complexity, greater interdependencies across classes, and less cohesiveness within classes, such initiatives are more difficult to sustain, alter, and expand.

These results make it clear that detecting and fixing code smells in a timely manner is essential to preserving software quality. Refactoring and proactive code quality evaluations can greatly improve software maintainability and lower technical debt. To guarantee scalability and long-term adaptability, the study's findings highlight the need of implementing sound coding methods, utilizing code smell detection tools, and routinely restructuring software.

**FUTURE RESEARCH DIRECTIONS**

These results make it clear that detecting and fixing code smells in a timely manner is essential to preserving software quality. Refactoring and proactive code quality evaluations can greatly improve software maintainability and lower technical debt. To guarantee scalability and long-term adaptability, the study's findings highlight the need of implementing sound coding methods, utilizing code smell detection tools, and routinely restructuring software.

Even though this study sheds light on how code smells affect maintainability, there are still several areas that need study:

* **Broader Study with More Projects**: A more thorough understanding of the connection between code smells and maintainability may be possible by broadening the study's scope to include a greater and more varied collection of open-source projects that span several programming languages and fields. This would enable cross-linguistic and cross-environmental comparisons and assist confirm the generalizability of our findings.
* **Longitudinal Analysis**: Researchers can monitor the evolution of code smells and the effects of their removal or mitigation on software maintainability by carrying out a longitudinal analysis on the same set of projects over time. This could help us comprehend how code smells affect a project's lifespan over the long run.
* **Automated Code Smell Detection Integration**: Further studies may examine the direct integration of code smell detection programs, such as JDeodorant, into pipelines for continuous delivery (CD) and continuous integration (CI). Teams may identify problems early and keep codebases cleaner and easier to manage by automating the identification and correction of code smells during the development process.
* **Refactoring Impact Study:** Although this study noted the detrimental consequences of code smells, more research might concentrate on determining how well various refactoring strategies can lessen their effects. For developers aiming to enhance their codebases, comparative research on the effects of different refactoring techniques on maintainability measures would be helpful.
* **Correlation with Other Software Quality Attributes**: More research might investigate the effects of code smells on other aspects of software quality, like testability, security, and performance. A more comprehensive grasp of the trade-offs and factors developers must consider whiledevelopingsoftware would result from a more thorough investigation of the relationship between code quality and various aspects of software quality.
* **User-Centric Evaluation**: To determine how developers' experiences with maintaining codebases with varying degrees of code smells correspond with the metrics examined in this work, a user-centric evaluation could be carried out. To close the gap between theoretical analysis and real-world experience, qualitative research comprising developer interviews and surveys may offer more context for the quantitative findings.

The discipline of software maintainability can advance by following these research avenues, providing developers with improved tools, techniques, and insights to create high-caliber, long-lasting software systems.

# 6 REFERENCES

1. Fowler, M., Beck, K., Brant, J., Opdyke, W., & Roberts, D. (1999). Refactoring: Improving the design of existing code. Addison-Wesley.
2. Lanza, M., & Marinescu, R. (2007). Object-oriented metrics in practice. Springer Science & Business Media.
3. Aniche, M. (n.d.). CK: A metrics suite for object-oriented software [GitHub repository]. GitHub. <https://github.com/mauricioaniche/ck>
4. Zhisheng17. (n.d.). Flink-learning. GitHub. Retrieved December 13, 2024, from <https://github.com/zhisheng17/flink-learning>
5. Topjohnwu. (n.d.). Libsu. GitHub. Retrieved December 13, 2024, from <https://github.com/topjohnwu/libsu>
6. Ixrjog. (n.d.). Opscloud4. GitHub. Retrieved December 13, 2024, from <https://github.com/ixrjog/opscloud4>
7. Tianshiyeben. (n.d.). Wgcloud. GitHub. Retrieved December 13, 2024, from <https://github.com/tianshiyeben/wgcloud>
8. Iluwatar. (n.d.). Java-design-patterns. GitHub. Retrieved December 13, 2024, from <https://github.com/iluwatar/java-design-patterns>
9. Alibaba. (n.d.). Druid. GitHub. Retrieved December 13, 2024, from <https://github.com/alibaba/druid>
10. Alibaba. (n.d.). Arthas. GitHub. Retrieved December 13, 2024, from <https://github.com/alibaba/arthas>
11. NLPchina. (n.d.). Elasticsearch-sql. GitHub. Retrieved December 13, 2024, from <https://github.com/NLPchina/elasticsearch-sql>
12. Wxiaoqi. (n.d.). Spring-Cloud-Platform. GitHub. Retrieved December 13, 2024, from <https://github.com/wxiaoqi/Spring-Cloud-Platform>
13. Ityouknow. (n.d.). Spring-boot-examples. GitHub. Retrieved December 13, 2024, from <https://github.com/ityouknow/spring-boot-examples>
14. Tsantalis, N. (n.d.). \*JDeodorant\*. GitHub. Retrieved December 13, 2024, from [https://github.com/tsantalis/JDeodorant](https://github.com/tsantalis/JDeodorant)