Journal of Software Engineering Research and Development, 2026, 14:13, doi: 10.5753/jserd.2026.6765

Exploring the Relationship Between SOLID Principles and Code Smells: The Software Developer Perspective

Ivandeclei Mendes [**Federal University of Minas Gerais** | ***ivandeclei@ufmg.br***]
Mateus Dutra [**Federal University of Minas Gerais** | ***mateusmdutra@ufmg.br***]
Khalid Usman [**Federal University of Juiz De Fora** | ***khalid.usman@estudante.ufjf.br***]
Eduardo Figueiredo [**Federal University of Minas Gerais** | ***figueiredo@dcc.ufmg.br***]

Abstract

Software engineering principles aim to deliver high-quality products efficiently. However, as development progresses and new features are added, software quality can deteriorate without proactive measures. Refactoring helps maintain software quality by systematically restructuring code through small incremental changes without altering its behavior. A crucial step in the refactoring process is identifying code smells, which are design or structural flaws that increase software complexity, hinder maintenance, and contribute to duplication and rework. This study examines developers' knowledge of code smells and their perceptions of using SOLID principles to avoid them. To explore this relationship, we conducted a survey with 71 professional developers. Our results show no correlation between knowledge of code smells and the developer's academic background or experience level. Additionally, participants widely recognized the SOLID principles as an effective way of avoiding code smells. These findings reinforce the importance of studying code smells, understanding their refactoring techniques, and applying the SOLID principles for maintaining high-quality software products.

Keywords: *Software Engineering, Refactoring, Code Smell, SOLID Principles, Code Maintenance, Software Design, Survey*

1 Introduction

Software systems evolution and maintenance are critical activities in software engineering and focus on the continuous modification of software to adapt to evolving requirements, improve functionality, and ensure user satisfaction (Aljedaani and Javed, 2020; Pathiraja, 2024). However, due to these necessary changes, the software quality tends to degrade over time, leading to increased complexity and maintenance challenges (de Oliveira and de Almeida, 2016; Pathiraja, 2024). This degradation often manifests through code smells – symptoms of poor design or implementation choices that indicate potential issues in the source code (Fowler, 1999). As software systems evolve, unplanned modifications, rushed implementations, and lack of proper refactoring contribute to the accumulation of code smells (Riquet et al., 2024), making the software systems harder to maintain and extend.

Refactoring involves a series of small, incremental steps aimed at improving code readability without altering its behavior or functionality (Fowler, 1999). It helps preserve the functionality of software code while making it easier to develop, debug, and read, thus accelerating delivery (Fowler, 1999). Refactoring can take various forms, including Preparatory Refactoring (performed before adding new features), Comprehension Refactoring (enhancing code readability), and Garbage Collection Refactoring (removing duplication) (Fowler, 1999). As software systems grow, issues like duplicated code and large classes emerge, requiring software engineering techniques and tools to optimize both the code and the overall system design.

The SOLID principles consist of five well known design guidelines in software development: Single Responsibility, Open-Closed, Liskov Substitution, Interface Segregation, and Dependency Inversion (Martin, 2017). These principles aim to provide a structured approach for organizing code, often associated with high modularity, low coupling, and extensibility (Martin, 2017). While their effectiveness may depend on the specific context and implementation, these principles are generally intended to improve software maintainability and adaptability. By applying these principles, software tend to become more resilient to change, which helps reduce the risk of code smells and the accumulation of technical debt (Martin, 2017). However, the relationship between the SOLID principles and the occurrence of code smells remains unclear. That is, we lack empirical evidence on whether SOLID principles effectively prevent or mitigate code smells.

To address this problem, this paper investigates the developers' perspectives on using SOLID principles to avoid code smells. To achieve this, we conducted a survey with 71 professional software developers to gather insights into their awareness of code smells and their views on using SOLID principles to avoid them. Our key findings indicate that the ability to recognize code smells is independent of practitioners' experience levels or academic backgrounds. Furthermore, our results suggest that SOLID principles are regarded as effective strategies for avoiding or refactoring code smells.

We also highlight the practical implications of our research for developers, such as the dominant role of the Single Responsibility Principle in common code smells. These results have implications for software engineering education since they uncover an underexplored relationship for maintaining high quality software products. Finally, our findings not only provide valuable guidance for software practitioners aiming

to improve existing software but also equip researchers with a deeper understanding of the relationship between these two important software engineering concepts.

Building on these insights, our study makes the following contributions.

- We provide empirical evidence of the relationship between SOLID principles and code smells from the perspective of professional developers.
- We identify a substantial knowledge gap regarding code smells, even among professional with several years of experience. This results has direct implications for software engineering education and training.
- We release a replication package (Replication package, 2025) that includes the survey instrumentation and an anonymous dataset, enabling transparency, reproducibility, and further extension of our research.

The remainder of this paper is organized as follows. Section 2 presents background information on SOLID principles and code smells, and outlines an example to motivate our research work. Section 3 describes the research methods used in this study. Section 4 presents the results of the survey. Section 5 discusses these results in light of the two research questions. Section 7 reviews related work on SOLID principles and code smells. Finally, Section 8 concludes the paper and outlines directions for future work.

2 Background

This section presents background information on the SOLID principles and code smells, and introduces the motivation for establishing a relationship between them.

2.1 SOLID Principles

Object-oriented design principles, such as SOLID, emerged to improve software development by making code more understandable, flexible, and maintainable (Cabral et al., 2024). These principles help reduce complexity and facilitate code reuse, which allows the development of more robust and understandable systems (Oktafiani and Hendradjaya, 2018). The SOLID principles also support systems that are easy to test, reducing coupling, and improving quality in the code development cycle (Chebanyuk and Markov, 2016). SOLID is an acronym to aggregate five well-known design principles, described as follows.

Single Responsibility Principle (SRP) states that a software module should have responsibility toward a single actor or stakeholder (Martin, 2017). Rather than limiting a module to one task, the principle emphasizes that a module should have only one reason to change, promoting cohesion and maintainability.

Open/Closed Principle establishes that software entities should be open for extension but closed for modification (Martin, 2017). This principle allows new functionality to be introduced without altering existing code, reducing the risk of introducing defects and improving system extensibility.

Liskov Substitution Principle (LSP) states that objects of a subclass should be replaceable with objects of their superclass without affecting program correctness (Martin, 2017). This principle ensures behavioral consistency in inheritance hierarchies and supports reliable polymorphism.

Interface Segregation Principle (ISP) recommends that clients should not be forced to depend on interfaces they do not use (Martin, 2017). Instead of large, general-purpose interfaces, smaller and more specific interfaces should be designed to improve modularity and maintainability.

Dependency Inversion Principle (DIP) states that high-level modules should depend on abstractions rather than concrete implementations (Martin, 2017). By reducing dependency on volatile components, this principle promotes flexibility, testability, and easier system evolution.

2.2 Code Smells

Code smells are symptoms of poor design and programming choices that indicate potential issues in the codebase (Fowler, 1999). Code smells may increase the complexity of software in terms of maintenance and adding new features (Hu et al., 2023; Liu et al., 2024; Palomba et al., 2018; Santana et al., 2024b). They can result in rework, code duplication, and other problems (Santos et al., 2023), highlighting the need for refactoring. It is important to emphasize that developers' intuition, acquired through years of experience, should determine whether a potential problem requires attention and modification (Fowler, 1999; Paiva et al., 2017).

Table 2 (in the appendix) presents the definitions of the main code smells discussed in this paper. We chose them because they are the most common in software engineering literature and may be related to the SOLID principles. For instance, Large Class and Long Method hurt the Single Responsibility Principle. In fact, while both SOLID principles and code smells have been extensively discussed in isolation, little practical research has examined their potential cooperation.

Prior surveys (Yamashita and Moonen, 2013) investigated general developer awareness of code smells, while others (Fernandes et al., 2016; Paiva et al., 2017; Palomba, 2015) proposed automated detection techniques. Similarly, some empirical studies (Källén et al., 2014; Yanakiev et al., 2024) assessed the impact of applying the SOLID principles in refactoring contexts. However, to the best of our knowledge, previous studies did not explicitly investigate how developers perceive the relationship between these two constructs. This gap motivated our research, which aims to capture the practitioner's perspective by surveying professional software developers. By bridging these two research strands, our work contributes an empirical dataset that links developers' understanding of SOLID principles with their views and experience on reducing code smells.

2.3 Motivation Example

Understanding how developers address code smells during software evolution is an important challenge in software engineering. In practice, developers often rely on design principles to guide refactoring decisions, yet the relationship be-

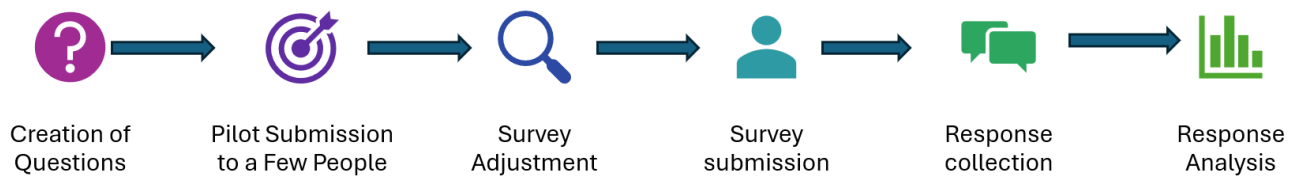

Figure 1. Overview of the Research Study

```java
public class OrderService {
    public void process(Order order) { ... }
    public void save(Order order) { ... }
    public void sendEmail(Order order) { ... }
}
```

Listing 1. Before refactoring

tween specific code smells and the principles used to mitigate them remains insufficiently documented. Establishing this connection can help clarify how design knowledge is applied into real-world maintenance activities.

To illustrate this relationship, consider a common scenario in software evolution in which a class gradually accumulates multiple, unrelated responsibilities. As new features are added, the same class becomes responsible for handling different concerns. Over time, this design becomes increasingly difficult to maintain. For example, changes to the database schema or to the notification mechanism may require modifications to this central class, increasing coupling and raising the risk of introducing defects. This progressive accumulation of responsibilities often leads to the emergence of a Large Class (also known as a God Class), one of the most frequently recognized code smells reported by developers in our survey and studied in the literature (Fernandes et al., 2016; Santana et al., 2024a).

Listing 1 presents a simplified Java example in which the `OrderService` class may violate the SRP. The class combines multiple concerns by handling business logic (order processing), data persistence (database operations), and external communication (email notifications).

Listing 2 illustrates a refactoring guided by the SRP. By separating responsibilities into distinct classes, the design becomes more modular and easier to evolve. Each component now encapsulates a single concern, reducing coupling and improving maintainability.

This example reflects the cognitive mapping observed in our survey: 44 developers explicitly identified SRP as the most appropriate principle for addressing Large Classes. Such evidence suggests that practitioners frequently rely on SOLID principles as conceptual guidelines for mitigating code smells.

Examining these associations is important from both practical and academic perspectives. In practice, identifying which design principles are commonly used to mitigate specific code smells can help developers make more informed refactoring decisions and improve the maintainability of evolving software systems. From an academic perspective, this knowledge can support the development of more intelligent code analysis tools capable of recommending refactor-

```java
public class OrderProcessor {
    public void process(Order order) { ... }
}

public class OrderRepository {
    public void save(Order order) { ... }
}

public class NotificationService {
    public void sendEmail(Order order) { ... }
}
```

Listing 2. After refactoring

ings grounded in design principles, as well as contribute to improving how software design practices are taught.

3 Research Method

In this section, we outline the research method used to conduct the survey, including the development process of the questionnaire (Section 3.1), our goal with the questions (Section 3.2), the pilot study conducted, and the data analysis approach to validate and refine the survey (Section 3.3). To guide our investigation, we defined the following research questions.

Research Questions

RQ1: How familiar are developers with the main code smells that can occur in a codebase?
This question evaluates developers' level of knowledge with the code smells considered in this study, investigating their prior knowledge and recognition ability regarding commonly occurring code smells analyzed in the survey.

RQ2: What are developers' perceptions regarding the use of SOLID principles to mitigate code smells?
This question examines practitioners' perceptions of applying SOLID design principles as refactoring strategies for addressing existing code smells, exploring how developers associate specific SOLID principles with distinct code quality issues.

3.1 Research Development

Figure 1 shows an overview of the research process used in this study. We developed the questionnaire using Google Forms, featuring various questions about the relationship of

SOLID principles and code smells. All questions were optional, allowing participants to respond freely and without obligation. Most of the questions were closed-ended, but there were also open-ended questions, allowing participants to describe, in more detail, other code smells not covered in the survey.

To recruit participants, we disseminated the survey through our professional networks, particularly LinkedIn, where the authors have established contacts with software engineers. Participation was voluntary, and we applied filters to ensure that only participants with professional or academic experience in software development were included in the final dataset. We distinguished between professional experience (measured in years of employment as a software developer or software engineer) and academic experience (measured in years of exposure to programming through formal education). To broadly assess the participants' experience levels, we also included demographic questions, which allowed us to classify participants and exclude invalid responses.

In total, we collected 71 valid responses. This sample size is comparable to, and in many cases larger than, other empirical studies in software engineering. For instance, Yamashita and Moonen (2013) conducted their study with a similar sample size (i.e., 85 developers). Abdelkader et al. (2025) collected 30 valid responses from a initial pool of participants after applying selection criteria, demonstrating that focused samples in the 30-70 range are valuable for specialized domains. Similarly, Ünlü et al. (2022) collected 67 total responses, with 45 participants (67%) having relevant experience and being included in the final analysis across 9 countries. These studies demonstrate that quality-driven samples in the range of 30-70 participants are not only acceptable but common and valuable for exploratory research in specialized software engineering domains. Furthermore, small-to-medium sample surveys are standard practice in exploratory studies, where the focus is on uncovering initial patterns and hypotheses rather than providing statistically generalizable results. To reduce threats to reliability, the survey was piloted with three developers, which allowed us to refine questions and ensure clarity.

3.2 Survey Questions

Table 1 presents the questions asked to developers who participated in the survey, organized into three groups. The first group contains demographic questions with the aim of collecting information about the profile of the participants. The second group contains questions about knowledge of SOLID principles. Finally, the third group includes questions about code smells and their relationship with the application of the SOLID principles.

After the demographic questions, the survey included questions to assess participants' knowledge of the SOLID principles S01 to S04 in Table 1. We aimed with these questions to determine which principles developers were most familiar with and which they perceived as easy or difficult to implement in practice.

To evaluate developers' understanding of code smells and their perception of the relationship between code smells and the application of SOLID principles, we asked a range of questions (C01 to C24 in Table 1). These included basic questions such as, "Are you familiar with the concept of code smells?". Additionally, for the main code smells identified by Fowler (1999), we asked direct questions to determine which principle developers would use to mitigate a specific code smell.

3.3 Pilot Study and Data Analysis

We sent the survey invitation to five developers as a pilot study to receive feedback before its actual execution. Three of these five pilot participants completed the survey and provided feedback. As a result, we made improvements, including enhancing the survey instructions and removing some redundant questions from the initial version.

We analyzed the data quantitatively. For each question, we examined the number of responses corresponding to each principle and plotted them in histograms. We analyze these charts in Section 4 to explore the relationship between code smells and the SOLID principles.

4 Research Results

This section presents the results gathered from the survey. First, we provide demographic information about the participants, including their academic background and years of practical experience (Section 4.1). Next, we explore their understanding of SOLID principles (Section 4.2) and code smells (Section 4.3). Finally, we analyze their perspectives on using SOLID principles to mitigate smells (Section 4.4).

4.1 Demographic Information

Figure 2 illustrates the educational backgrounds of the participants, who came from a variety of academic and professional experiences. Their academic qualifications ranged from incomplete undergraduate studies to doctoral degrees. While some participants were still in the process of completing their undergraduate studies, their responses were included to assess whether academic background had an influence on the survey results. Figure 3 shows that only five participants had limited practical experience; i.e., less than 1 year of experience in software development. On the other hand, 54 participants (76%) in our survey have more than four years of experience in software development in various programming languages, such as .Net, PHP, Java, and JavaScript, among others, as seen in Figure 4.

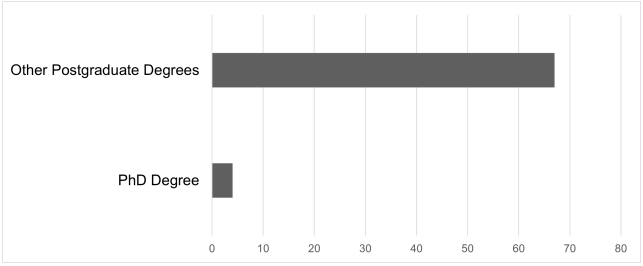

Figure 2. Level of Education

Table 1. Survey Questions

ID	Question
	Demographic Questions
D01	What is your nationality?
D02	What is your level of education?
D03	How many years of professional experience do you have as a software developer?
D04	What is your main programming language?
	Questions About SOLID Principles
S01	Do you know the SOLID software design principles?
S02	Which SOLID principles do you know? (Select all that apply)
S03	How often do you apply the SOLID principles when writing code?
S04	On a scale from very easy to very difficult, how do you rate the difficulty of applying each SOLID principle?
	Questions About Code Smells
C01	Do you know the concept of code smells?
C02	Which of the following code smells do you recognize? (Select all that apply)
C03	Which methods do you use to identify code smells in your code? (Select all that apply)
C04	Which SOLID principle would you apply to improve the design when handling the *Duplicated Code* code smell?
C05	Which SOLID principle would you apply to improve the design when handling the *Long Method* code smell?
C06	Which SOLID principle would you apply to improve the design when handling the *Large Class* code smell?
C07	Which SOLID principle would you apply to improve the design when handling the *Divergent Change* code smell?
C08	Which SOLID principle would you apply to improve the design when handling the *Shotgun Surgery* code smell?
C09	Which SOLID principle would you apply to improve the design when handling the *Data Clumps* code smell?
C10	Which SOLID principle would you apply to improve the design when handling the *Primitive Obsession* code smell?
C11	Which SOLID principle would you apply to improve the design when handling the *Repeated Switches* code smell?
C12	Which SOLID principle would you apply to improve the design when handling the *Temporary Field* code smell?
C13	Which SOLID principle would you apply to improve the design when handling the *Refused Bequest* code smell?
C14	Which SOLID principle would you apply to improve the design when handling the *Alternative Classes with Different Interfaces* code smell?
C15	Which SOLID principle would you apply to improve the design when handling the *Global Data* code smell?
C16	Which SOLID principle would you apply to improve the design when handling the *Lazy Element* code smell?
C17	Which SOLID principle would you apply to improve the design when handling the *Speculative Generality* code smell?
C18	Which SOLID principle would you apply to improve the design when handling the *Feature Envy* code smell?
C19	Which SOLID principle would you apply to improve the design when handling the *Inappropriate Intimacy* code smell?
C20	Which SOLID principle would you apply to improve the design when handling the *Message Chains* code smell?
C21	Which SOLID principle would you apply to improve the design when handling the *Middleman* code smell?
C22	Which SOLID principle would you apply to improve the design when handling the *Mutable Data* code smell?
C23	Which SOLID principle would you apply to improve the design when handling the *Loops* code smell?
C24	Which SOLID principle would you apply to improve the design when handling the *Data Classes* code smell?

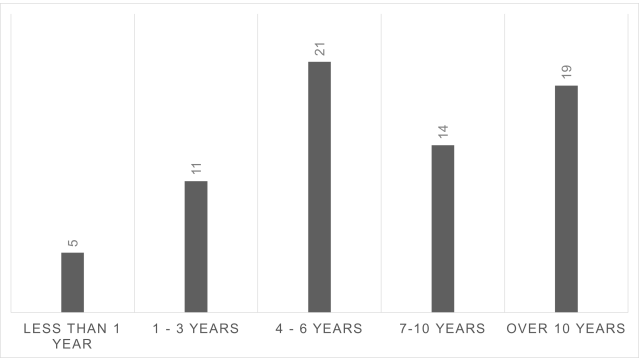

Figure 3. Years of Participants' Experience

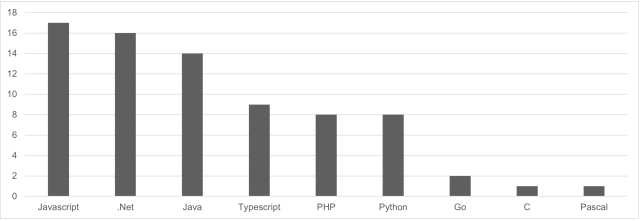

Figure 4. Programming Languages Used by Developers

4.2 Knowledge of the SOLID Principles

The survey answers showed that 68 participants claimed to be know with the SOLID principles and generally consider the principles when writing code. However, only fourteen participants think about all five principles on a daily basis. An important observation is the level of difficulty reported by participants for each principle. The this information can be helpful for improving the education strategies of these principles.

Figure 5 shows how developers perceive the difficulty of applying SOLID principles in code development. The difficulty levels presented are very easy, easy, moderate, hard, and very hard to apply. For each principle, participants could choose a level of difficulty. It is possible to see that, with the exception of the Single Responsibility Principle, the other principles are often considered to be of moderate difficulty. The principle identified as the most difficult was the Liskov Substitution Principle.

4.3 Code Smells

There are various types of code smells. However, most papers (Amorim et al., 2025; Santana et al., 2024b; Di Nucci et al., 2018) usually focus on addressing some smells defined by Fowler (1999). This paper covers a high number of

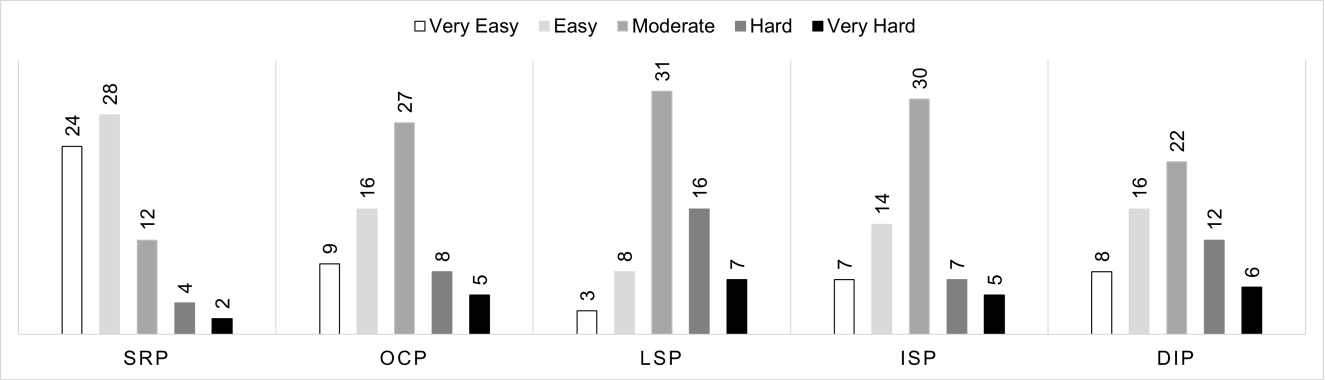

Figure 5. Difficulty levels of applying each SOLID principle

code smells and aims at investigating which SOLID principle could be useful in refactoring and resolving code smells, based on the developers' perspectives.

Figure 6 presents the code smells in decreasing order of familiarity among developers, highlighting the number of participants who claimed to know each of them. Our results showed that {68 participants claimed to understand the concept of code smells. However, among the smells mentioned in the survey, only a few participants were familiar with all of them. No other code smells besides those presented in Figure 6 were mentioned by the participants. The most recognized code smells were: Duplicated Code, Long Function, Large Class and Global Data.

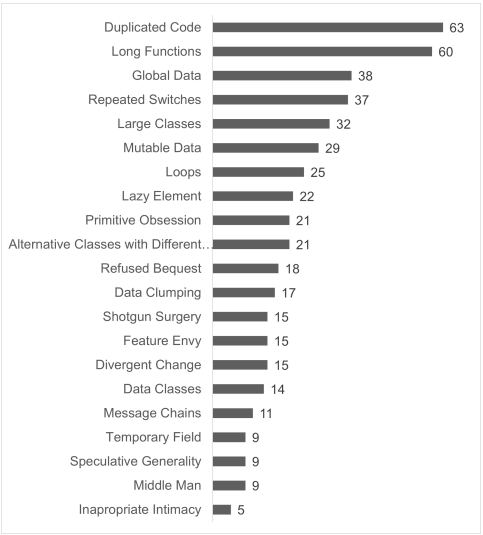

Figure 6. Code smells known by developers

4.4 Relationship between Code Smells and SOLID Principles

Our empirical survey investigates the perceived efficacy of SOLID principles in the remediation of common software code smells. Figure 7 illustrates how developers map specific principles to distinct categories of code smells. Overall, this figure reveals a structured and nuanced understanding among practitioners, where each SOLID principle is mostly associated with two or three specific code smells, providing insights into the practical relationships of these foundational design concepts.

Single Responsibility Principle (SRP). The survey data indicate that SRP is predominantly viewed as the primary mitigating strategy for code smells related to excessive component size and functional overload, as shown in Figure 7. Long Functions was mentioned by 48 developers, followed by Large Classes mentioned by 44 developers, and Duplicated Code cited by 40 developers. The association of SRP to these smells suggests that developers perceive SRP as a fundamental tool for enforcing modularity. By constraining a module to a single well-defined responsibility, this principle naturally limits component size and complexity. In turn, SRP reduces the likelihood of code duplication by fostering the creation of reusable and single-purpose modules.

Open/Closed Principle (OCP). The OCP principle, which mandates that software entities be extensible without modification, was primarily correlated with smells arising from structural rigidity and complex conditional logic. The most pronounced association was with Repeated Switches with 17 developers mentioning it, which is shown in Figure 7. This result aligns with the canonical OCP application of replacing conditionals with polymorphism. Furthermore, 8 developers identified OCP as a remedial factor for Divergent Change and, to a lesser degree, the speculative generality smell cited by 6 developers. These results underscore the recognized value of OCP in designing software systems that are resilient to change and can evolve without incurring significant maintenance overhead.

Liskov Substitution Principle (LSP). Figure 7 shows that the Liskov Substitution Principle (LSP) was consistently identified as the key to resolving violations within inheritance hierarchies. Its strongest perceived application was in addressing Refused Bequest a code smell where a subclass invalidates the contract of its superclass as reported by 11 developers. This correspondence is theoretically sound as LSP's core tenet of behavioral subtyping directly enforces semantic integrity in inheritance. A secondary association was established with Primitive Obsession with 6 developers. This association indicates that practitioners may view the creation of robust substitutable type hierarchies guided by LSP as a sophisticated alternative to the overuse of primitive data types.

Interface Segregation Principle (ISP). Figure 7 reveals that ISP exhibited a highly specialized application, being almost exclusively correlated with a single code smell. A significant majority of developers 30 cited ISP as the definitive solution for Alternative Classes with Different Interfaces.

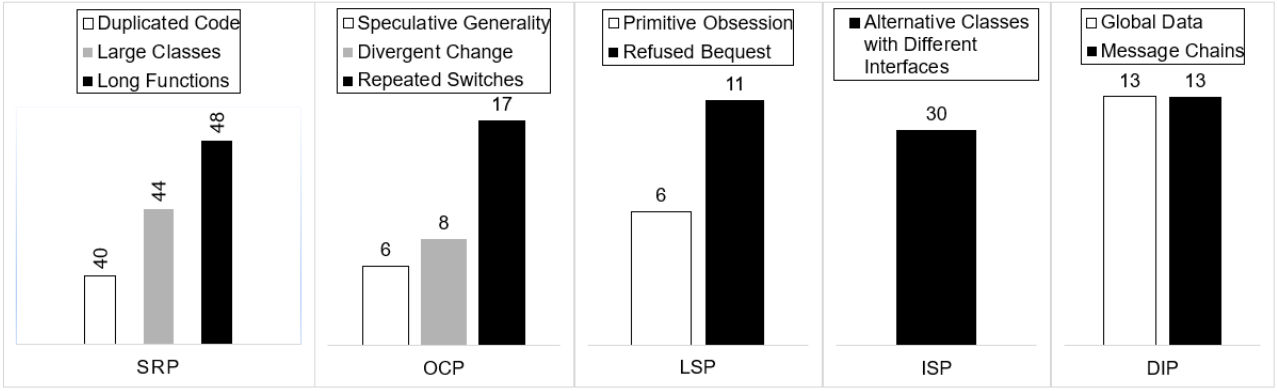

Figure 7. Code smells grouped by SOLID principles as perceived by developers

This strong, focused association highlights the principle's role in preventing cohesive-lacking, "fat" interfaces. By promoting the design of fine-grained, client-specific interfaces, ISP directly remedies situations where disparate classes implement similar functionality through non-uniform contracts, thereby enforcing architectural consistency and improving software systems interoperability.

Dependency Inversion Principle (DIP). Finally, DIP was primarily associated with the management of improper coupling and transitive dependencies. This principle was cited as a solution for both Message Chains mentioned by 13 developers and the Global Data pointed out by 13 developers, as shown in Figure 7. This perception aligns with the fundamental goal of DIP to decouple high-level policy from low-level implementation by mediating dependencies through abstractions. This practice inherently disrupts deep, fragile dependency chains and insulates components from reliance on globally scoped state. Thus, it contributes to a more modular, flexible, and testable software architecture.

In summary, our findings substantiate a highly differentiated cognitive mapping between the SOLID principles and specific code smell remediation strategies among software developers. A clear dichotomy emerges wherein Single Responsibility Principle is perceived as a fundamental mechanism for controlling modular complexity, while other principles, such as LSP and ISP, are precisely targeted at rectifying distinct software problems within inheritance and interface contracts. This evidence points toward a non-monolithic context-aware application of design principles, deployed as a nuanced set of interventions to enhance specific facets of software architecture and maintainability.

5 Discussions

In this section, we discuss our findings in the context of our two research questions. Our research reveals a structured and nuanced understanding among practitioners regarding the relationship between design principles and code quality. While both SOLID and code smells are often discussed in isolation, this study provides empirical evidence of their practical cooperation in refactoring.

RQ1: *How familiar are developers with the main code smells that can occur in a codebase?*

The survey results indicate that while the general concept of code smells is universally understood, specific knowledge is highly fragmented and exhibits no correlation with professional seniority. To evaluate this, we analyzed responses from a sample where 78% of participants possessed over four years of professional experience, and a significant portion had exceeded ten years in the field. Despite this high level of seniority, the data confirms a "knowledge-experience paradox" wherein years of practice do not equate to a comprehensive understanding of structural flaws. Only the most basic smells achieved near-universal recognition, specifically Duplicated Code, known by 63 participants, and Long Functions, known by 60.

As the complexity of the code smells increases, the recognition rates drop significantly, regardless of the developer's career stage. For instance, structural smells such as Inappropriate Intimacy and Middle Man were identified by only 5 and 9 participants, respectively. This sharp decline among a predominantly senior cohort provides strong evidence that code smell recognition is a specialized skill that does not naturally "accrue" through passive professional exposure or formal academic background. These findings highlight a substantial knowledge gap in the industry, suggesting that seniority alone is an insufficient proxy for the ability to diagnose technical debt.

RQ2: *What are the developers perceptions on the use SOLID principles to mitigate code smells?*

Our findings substantiate a highly differentiated cognitive mapping process where developers treat SOLID principles as a targeted set of interventions rather than a monolithic block. These findings indicate a highly differentiated cognitive mapping in which developers perceive SOLID principles not as a unified framework but as a set of targeted interventions applied to specific design problems. Among these principles, Figure 8 shows that SRP emerges as the primary strategy for mitigating the most common code smells. It is most frequently associated with Long Function, Large Class, and Duplicated Code, suggesting that developers regard SRP as a fundamental mechanism for enforcing modularity and controlling software complexity.

Beyond SRP, the remaining SOLID principles are perceived as more specialized mechanisms for addressing dis-

tinct structural issues. OCP is primarily associated with replacing rigid conditional logic, particularly in cases involving Repeated Switches. LSP is linked to resolving inheritance-related design violations, most notably Refused Bequest. ISP is viewed as an effective strategy for addressing Alternative Classes with Different Interfaces, helping prevent the emergence of non-cohesive interfaces. Finally, DIP is commonly associated with managing improper coupling, particularly in situations involving Message Chains and Global Data.

Our results suggest that when developers consistently understand these principles, they perceive them as effective tools for both preventing and mitigating a wide array of code smells. However, the identified knowledge gaps particularly regarding less common smells hinder the full realization of these benefits. This gap points to a need for a shift in software engineering education: moving from teaching SOLID as abstract theory to presenting it as a diagnostic and corrective framework for specific, detectable code smells.

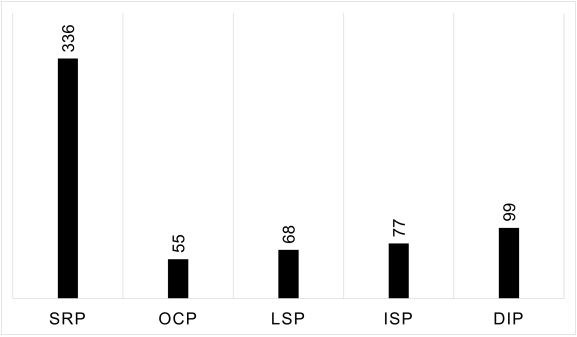

Figure 8. Number of times a SOLID principle was cited for all code smells.

6 Threats to Validity

This section aims to address the threats to validity of the research, by considering four typical categories of threats: construct, internal, external, and conclusion (Wohlin, 2000).

Construct validity refers to the accuracy of the survey instrument in measuring the intended concept (Wohlin, 2000). Although the questions were designed to be clear and precise, there is a possibility that some participants misunderstood the terminology related to SOLID principles and code smells. Furthermore, although most developers claimed to understand these concepts, where some participants reported that they were familiar with several concepts, but several code smells were unfamiliar to them. Reliance on self-reported data introduces a risk of response bias, where participants may overestimate or underestimate their familiarity with code smells or the SOLID principles.

Another potential threat to construct validity is the length of the questionnaire. The relatively large number of questions may have led to respondent fatigue, potentially affecting the attention and care with which subsequent questions were answered. This fatigue effect may have introduced random or less thoughtful responses, thus influencing the reliability of the collected data. The mitigation strategy was not to require participants to answer all questions, allowing them to skip items when necessary. This approach could reduce the pressure on respondents and help maintain the quality of responses throughout the research.

Internal validity refers to the extent to which the study accurately represents causal relationships (Wohlin, 2000). To ensure reliable responses, we primarily targeted developers with at least three years of experience, and given participation of less experienced professionals was minimal. However, some developers with more than four years of experience still showed a lack of knowledge of certain code smells. This suggests that there may be knowledge gaps or differences in professional focus, potentially biasing the results. However, we believe this does not compromise the analysis of how SOLID principles are applied in addressing code smells, given the experience levels of most respondents.

External validity refers to the generalizability of the results beyond the study sample (Wohlin, 2000). Given the predominantly quantitative nature of the research, it is expected that repeating the study with different participants using the same questionnaire would yield similar results. However, variations in participant backgrounds, such as companies worked for and project type, may affect knowledge with specific code smells. Furthermore, with a sample size of 71 respondents from different countries (hidden for the anonymous review process), the findings may not be fully representative of the broader developer community. Future studies should consider expanding the sample size and including developers from diverse backgrounds to increase the generalizability of the results.

Conclusion validity refers to the reliability of inferences drawn from the collected data (Wohlin, 2000). Due to the limited sample size, some findings may not be statistically significant, particularly for less familiar code smells. Additionally, self-reported responses may not always accurately reflect actual coding practices, potentially creating a gap between theoretical knowledge and real-world application in organizations. There is also the possibility of social desirability bias, where participants provide responses they believe are expected rather than those that reflect their true experiences. Although the survey was anonymous and allowed for responses to be revoked, we cannot be certain that all responses were entirely accurate. These factors emphasize the need for further empirical validation, such as analyzing real-world codebases to confirm the research findings.

By acknowledging these threats to validity, we acknowledge the limitations of the study while reinforcing its methodological rigor. Future research can address these limitations by incorporating larger and more diverse samples, as well as by supplementing survey data with observational or experimental approaches.

7 Related Work

Several studies address the detection of code smells and the application of the SOLID principles as strategies to identify and mitigate issues related to technical debt in software products. These two concepts are directly connected, as code smells help identify problematic areas in the code, while the SOLID principles provide guidelines to improve the quality and maintainability of the software. The following sections

discuss some studies related to these two aspects.

7.1 Code Smells

Palomba (2015) proposed a textual analysis approach called TACO (Textual Analysis for Code smell detection) to identify Code smells in source code. TACO was instantiated to detect the Long Method smell. Although, TACO can be extended to other types of Code smells as well. The study evaluated three open-source Java projects: Apache Cassandra, Apache Xerces, and Eclipse Core. The results showed that TACO could detect between 50% and 77% of Long Method instances with a precision ranging from 63% to 67%. Additionally, TACO identified code smells that were not detected by approaches based solely on structural information.

Yamashita and Moonen (2013) conducted a survey with developers to undertand their knowledge about code smells and if they care about them. The survey had 85 responses from professional developers. They found that 32% of respondents did not know about code smells. Most developers mentioned the Large Class, Long Method, and the antipattern Accidental Complexity. The study revealed that developers claimed for better support to detect and refactor the codebase following the software evolution.

Silva et al. (2024) evaluated the effectiveness of ChatGPT in detecting code smells in Java projects, using two types of prompts: generic and specific. The study used the MLCQ dataset (Madeyski and Lewowski, 2020) with 14,739 instances of code smells and showed that specific prompts were more effective, particularly in detecting critical ”smells”. ChatGPT performed better in detecting Data Class and Long Method, but encountered difficulties with Blob and Feature Envy. The study suggests that ChatGPT could be a useful tool for identifying design issues in code, especially with detailed prompts.

Santana et al. (2024b) investigated how the agglomeration of code smells affects code stability. The study analyzed two years of commit history from 30 open source Java systems on GitHub, using four measures: number of commits, modified lines of code, rate of modified classes, and proportion of changes. The results showed that classes with two or more types of code smells change more frequently and intensely, increasing maintenance and evolution costs. The study identified nine types of code smells at different levels of code and observed that heterogeneous agglomerations were more harmful than homogeneous agglomerations and isolated classes, indicating greater instability. Furthermore, it was found that classes deleted over time tend to have many lines added before they become unmaintainable, highlighting the importance of prioritizing the refactoring. In terms of practical implications, the insights from the study can help developers and maintainers make informed decisions about refactoring strategies, prioritizing efforts to improve code quality. The research also contributes to the understanding of the impact of code smell clusters on code stability, encouraging discussions on best practices for managing these clusters, and the availability of the collected data and the tools used promotes the replication and extension of the study, increasing the validity of the results presented.

7.2 SOLID Principles

Yanakiev et al. (2024) conducted an analysis on the use of SOLID principles to refactor a legacy ASML codebase. The repository comprised 14 components, with more than 3,000 implementation files and 1,000 interfaces. Initially, they defined metrics to monitor the quality of the components. Additionally, they performed an impact analysis and subsequently established a prioritization based on the effect of each selected quality metric. Through a literature review, they identified Dependency Injection as the most suitable SOLID principle for the refactoring. The study identified four illegal dependencies across three components, involving nine interfaces and over 350 illegally used symbols. The refactoring process successfully removed 50% of the illegal dependencies and reduced illegally used exported symbols by over 85%, encapsulating them to prevent future violations. The findings emphasize the challenges of applying design principles to large-scale codebases and highlight the value of strategic refactoring, impact analysis, and design patterns in managing technical debt and improving maintainability.

Källén et al. (2014) refactored an academic application called HAParaNDA using SOLID principles and assessed the impact of each refactoring on code quality. They used software quality metrics, such as LOC (Lines of Code), AMS (Average Method Size), and others to measure the effects. A quantitative and qualitative analysis of the refactored codebase revealed that procedurally written code became more object-oriented, enhancing maintainability. However, RFC (Response for a Class) and LCOM (Lack of Cohesion in Methods) increased, and application performance declined due to a rise in polymorphic calls. The findings highlight that while applying SOLID principles improves maintainability, it can come at the cost of other factors, such as performance.

8 Conclusions and Future Work

This research investigated the application of the SOLID principles in mitigating code smells and highlighted the need for a broader dissemination of this topic. Many developers who participated in the survey had not been familiar with various code smells, which made it difficult to identify and propose solutions to them.

Despite this, it became clear that the SOLID principles were perceived as effective tools for addressing various code smells. This result suggested that, as long as developers consistently understood each of these principles, it was possible to prevent and mitigate a large number of code smells, there by promoting software quality. However, it was necessary to spread awareness about code smells, as the research demonstrated that knowledge of code smells was independent of academic background or years of experience.

As future work, we propose further empirical studies that directly involve the developer community. These studies will aim to understand the challenges developers face in acquiring knowledge about the SOLID principles and code smells, considering that many developers are unfamiliar with several of these concepts when presented directly. To address this gap, it is essential to promote the dissemination of this

knowledge through scientific papers and playful approaches, particularly in undergraduate courses.

Additionally, it is planned to conduct experiments with professionals to determine whether the correlation observed in this research remains consistent when applying the SOLID principles to solve code smells in a controlled environment. These experiments could provide practical evidence of the principles effectiveness and help develop more efficient strategies for their dissemination and application.

Acknowledgements

This research was partially supported by Brazilian funding agencies: CNPq (Grant 406089/2025-6), CAPES, and FAPEMIG (Grant APQ-01488-24).

References

Abdelkader, H., Abdelrazek, M., Rani, P., Vasa, R., and Schneider, J.-G. (2025). Ensuring robustness in ml-enabled software systems: A user survey. *ArXiv*, abs/2510.18292.

Aljedaani, W. and Javed, Y. (2020). Empirical study of software test suite evolution. In *2020 6th Conference on Data Science and Machine Learning Applications (CDMA)*, pages 87–93.

Amorim, L., da Costa, I. M., Alves, L., and Figueiredo, E. (2025). Bad smell detection using google gemini. In *IEEE 49th Annual Computers, Software, and Applications Conference (COMPSAC)*, pages 1637–1642.

Cabral, R., Kalinowski, M., Baldassarre, M. T., Villamizar, H., Escovedo, T., and Lopes, H. (2024). Investigating the impact of solid design principles on machine learning code understanding. In *Proceedings of the IEEE/ACM 3rd International Conference on AI Engineering - Software Engineering for AI*, page 7–17. Association for Computing Machinery.

Chebanyuk, E. and Markov, K. (2016). An approach to class diagrams verification according to solid design principles. In *2016 4th International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*, pages 435–441.

de Oliveira, R. P. and de Almeida, E. S. (2016). Evaluating lehman's laws of software evolution for software product lines. *IEEE Software*, 33(3):90–93.

Di Nucci, D., Palomba, F., Tamburri, D. A., Serebrenik, A., and De Lucia, A. (2018). Detecting code smells using machine learning techniques: Are we there yet? In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 612–621.

Fernandes, E., Oliveira, J., Vale, G., Paiva, T., and Figueiredo, E. (2016). A review-based comparative study of bad smell detection tools. In *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering*, EASE.

Fowler, M. (1999). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley.

Hu, W., Liu, L., Yang, P., Zou, K., Li, J., Lin, G., and Xiang, J. (2023). Revisiting "code smell severity classification using machine learning techniques". In *2023 IEEE 47th Annual Computers, Software, and Applications Conference (COMPSAC)*, pages 840–849.

Källén, M., Holmgren, S., and þóra Hvannberg, E. (2014). Impact of code refactoring using object-oriented methodology on a scientific computing application. In *IEEE 14th International Working Conference on Source Code Analysis and Manipulation*, pages 125–134.

Liu, L., Lin, G., Zhu, L., Yang, Z., Song, P., Wang, X., and Hu, W. (2024). Revisiting code smell severity prioritization using learning to rank techniques. *Expert Systems with Applications*, 249:123483.

Madeyski, L. and Lewowski, T. (2020). Mlcq: Industry-relevant code smell data set. In *Proceedings of the 24th International Conference on Evaluation and Assessment in Software Engineering*, EASE '20, page 342–347. Association for Computing Machinery.

Martin, R. C. (2017). *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Prentice Hall Press, 1st edition.

Oktafiani, I. and Hendradjaya, B. (2018). Software metrics proposal for conformity checking of class diagram to solid design principles. In *5th International Conference on Data and Software Engineering (ICoDSE)*, pages 1–6.

Paiva, T., Damasceno, A., Figueiredo, E., and Sant'Anna, C. (2017). On the evaluation of code smells and detection tools. *Journal of Software Engineering Research and Development*, 5(1):7.

Palomba, F. (2015). Textual analysis for code smell detection. In *Proceedings of the 37th International Conference on Software Engineering - Volume 2*, ICSE '15, page 769–771. IEEE Press.

Palomba, F., Bavota, G., Di Penta, M., Fasano, F., Oliveto, R., and De Lucia, A. (2018). On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, page 482. Association for Computing Machinery.

Pathiraja, A. L. C. D. (2024). Software quality assurance practices towards waterfall and agile information systems projects of boi registered software companies in sri lanka. In *2024 9th International Conference on Information Technology Research (ICITR)*, pages 1–4, Colombo, Sri Lanka.

Replication package (2025). Survey response dataset repository. Available at: https://github.com/Ivandeclei/replication-package-solid-code-smell/tree/main.

Riquet, N., Devroey, X., and Vanderose, B. (2024). Debt stories: Capturing social and technical debt in the industry. In *2024 IEEE/ACM International Conference on Technical Debt (TechDebt)*, pages 40–44.

Santana, A., Figueiredo, E., Alves Pereira, J., and Garcia, A. (2024a). An exploratory evaluation of code smell agglomerations. *Software Quality Journal*, 32(4):1375–1412.

Santana, A., Figueiredo, E., and Pereira, J. A. (2024b). Unraveling the impact of code smell agglomerations on code stability. In *2024 IEEE International Conference on Soft-*

ware Maintenance and Evolution (ICSME), pages 461–473.

Santos, G., Santana, A., Vale, G., and Figueiredo, E. (2023). Yet another model! a study on model's similarities for defect and code smells. In *International Conference on Fundamental Approaches to Software Engineering*, pages 282–305. Springer.

Silva, L. L., Silva, J. R. d., Montandon, J. E., Andrade, M., and Valente, M. T. (2024). Detecting code smells using chatgpt: Initial insights. In *Proceedings of the 18th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '24, page 400–406. Association for Computing Machinery.

Sobrinho, E. V. d. P. and Maia, M. d. A. (2021). On the interplay of smells large class, complex class and duplicate code. In *Brazilian Symposium on Software Engineering*, pages 64–73.

Wohlin, C. (2000). *Experimentation in Software Engineering: An Introduction*. The sKluwer ISECS.

Yamashita, A. and Moonen, L. (2013). Do developers care about code smells? an exploratory survey. In *2013 20th Working Conference on Reverse Engineering (WCRE)*, pages 242–251.

Yanakiev, I., Lazar, B.-M., and Capiluppi, A. (2024). Applying solid principles for the refactoring of legacy code: An experience report. *Journal of Systems and Software*, 220.

Ünlü, H., Bilgin, B., and Demirörs, O. (2022). A survey on organizational choices for microservice-based software architectures. *Turkish Journal of Electrical Engineering and Computer Sciences*, 30(4):1187–1203.

A Appendix

Table 2. Examples of Code Smells and Their Definitions

DUCO	Duplicated Code	It occurs when identical or similar code segments appear in multiple places, signaling a potential problem that may require refactoring. (Fowler, 1999).
LOFU	Long Function	It refers to a method or function that is overly long and complex, consolidating logic from many areas. It is one of the most common code smells, making comprehension, maintenance, and evolution difficult. Often, this smell is seen in early versions of software (Fowler, 1999).
LACL	Large Class	It happens when a class takes on too many functionalities and responsibilities, leading to bloated, hard-to-maintain code. It is often coupled with duplicated code and requires frequent updates, increasing the risk of compatibility-breaking changes (Sobrinho and Maia, 2021).
DICH	Divergent Change	It is identified when a module is modified in different ways for various reasons, usually during the addition of new features. This smell arises when changes need to be applied across multiple locations in the codebase (Fowler, 1999).
SHSU	Shotgun Surgery	It is similar to Divergent Change but in the opposite direction. It states that modifying one part of the code requires small changes in other classes or methods. This can lead to forgotten modifications and subsequent issues (Fowler, 1999).
DACL	Data Clumps	It occurs when related properties appear together in various places. This suggests the properties should be encapsulated in a dedicated class, which makes the data a logical unit (Fowler, 1999).
PROB	Primitive Obsession	It refers to the use of primitive types where a specialized type should be used instead, such as representing a phone number as a string. This indicates the code could benefit from more appropriate data types (Fowler, 1999).
RESW	Repeated Switches	It happens when switch statements are duplicated throughout the code. Polymorphism can often replace switch statements, though modern switch instructions may be more complex than before (Fowler, 1999).
TEFI	Temporary Field	It occurs when a variable, which should be scoped within a method, is declared at the class level. This increases accessibility unnecessarily and makes the code harder to maintain (Fowler, 1999).
REBE	Refused Bequest	It happens when subclasses inherit unnecessary data from their parent class, indicating that the inheritance relationship is poorly applied. This results in confusing, difficult-to-maintain code (Fowler, 1999).
ALCL	Alternative Class w/ Different Interfaces	It is seen when multiple classes with similar purposes have different interfaces, which reduces cohesion and increases complexity, making the code harder to read and understand (Fowler, 1999).
GLDA	Global Data	It occurs when variables are declared globally and accessed throughout the software systems, making dependencies implicit and increasing coupling. (Fowler, 1999).
LAEL	Lazy Element	It happens when a class, method, or variable is present in the code but is rarely used or does not justify its existence.(Fowler, 1999).
SPGE	Speculative Generality	It occurs when parts of the code are designed for future use but are never actually needed. (Fowler, 1999).
FEEN	Feature Envy	It happens when a method in one class is more interested in the data of another class rather than its own. This indicates that functionality might be misplaced and should be refactored (Fowler, 1999).
ININ	Inappropriate Intimacy	It occurs when a class relies too much on the internal details of another class, violating encapsulation and increasing coupling. (Fowler, 1999).
MUDA	Mutable Data	It refers to data that is modified in multiple locations, making the behavior of the program unpredictable. (Fowler, 1999).
LOOP	Loops	It happens when loops are used excessively instead of more readable constructs, such as higher-order functions or declarative approaches. (Fowler, 1999).
MIMA	Middle Man	It occurs when a class exists solely to delegate work to another class without adding any real value, introducing unnecessary indirection and complexity (Fowler, 1999).