Minimizing Maintenance Cost through Prioritizing Refactoring of Code Smells

Md. Masudur Rahman

A Thesis
Submitted to the Institute of Information Technology (IIT)
University of Dhaka

For fulfillment of degree requirement of

DOCTOR OF PHILOSOPHY (Ph.D.)

Institute of Information Technology (IIT)
UNIVERSITY OF DHAKA
DHAKA, BANGLADESH

Minimizing Maintenance Cost through Prioritizing Refactoring of Code Smells

By

Md. Masudur Rahman

Institute of Information Technology (IIT)

University of Dhaka

Supervised by

Dr. Kazi Muheymin-Us-Sakib

Professor

&

Dr. Md. Mahbubul Alam Joarder

Professor

Institute of Information Technology (IIT)

University of Dhaka

Institute of Information Technology (IIT)

UNIVERSITY OF DHAKA

AUTHOR'S DECLARATION

I hereby declare that the thesis entitled *"Minimizing Maintenance Cost through Prioritizing Refactoring of Code Smells"* is the result of my original research carried out under the supervision of Dr. Kazi Muheymin-Us-Sakib and Dr. Md. Mahbubul Alam Joarder. To the best of my knowledge, this work has not been submitted, either in whole or in part, for any other academic degree or qualification at this or any other institution. All sources of information used in this thesis have been duly acknowledged and referenced. The data, analyses, and conclusions presented herein are based on my own investigations unless explicitly stated otherwise. This work is submitted in fulfillment of the requirements for the award of the degree of Doctor of Philosophy (Ph.D.) in Institute of Information Technology (IIT), University of Dhaka.

Author: Md. Masudur Rahman

Registration No. - 138

Session: 2020 - 2021

Email: bit0413@iit.du.ac.bd

Institute of Information Technology (IIT)

University of Dhaka

CERTIFICATE FROM THE SUPERVISORS

This is to certify that the thesis entitled *"Minimizing Maintenance Cost through Prioritizing Refactoring of Code Smells"* is a research work carried out by **Md. Masudur Rahman** under our guidance and supervision for the requirement of the degree of Doctor of Philosophy (Ph.D.) to be awarded by the University of Dhaka. We also certify that the candidate has committed sufficient time during his work under our supervision for his research work.

To the best of our knowledge, the thesis satisfies the following:

a. Embodies the work done by the candidate himself only.

b. Duly satisfies the requirements of the degree of Doctor of Philosophy (Ph.D.) from the University of Dhaka.

c. Satisfies the desired standard with respect to the quality of the content, depth of language, and format of presentation befitting the current work.

d. Does not contain any part which forms the basis for the award of any degree, diploma, or certification, and/or similar thesis other than the sole purpose of degree requirement of Doctor of Philosophy (Ph.D.) at the University of Dhaka for which the current work has been submitted.

We congratulate the candidate on his research contribution and wish him continued success in his future academic career.

Supervisor:
Dr. Kazi Muheymin-Us-Sakib
Professor
Institute of Information Technology (IIT)
University of Dhaka

Co-supervisor:
Dr. Md. Mahbubul Alam Joarder
Professor
Institute of Information Technology (IIT)
University of Dhaka

To my dear parents,
Md. Ismail Mollah & Mahmuda Khanam,

whose constant encouragement and support have shaped who I am today and made this journey possible

ABSTRACT

Code smells are indicators of poor design practices that increase complexity, reduce comprehensibility, and hinder maintainability. While they do not directly affect system functionality, their long-term presence leads to higher maintenance costs and degraded software quality. Refactoring is the primary strategy to remove or minimize code smells. However, addressing all smell types is impractical due to time, budget, and resource constraints. Thus, identifying and prioritizing the most impactful code smells is essential for efficient maintenance and sustainable evolution of software systems.

Code smell prioritization in existing literature is largely system-specific and often requires significant developer intervention. In other words, prioritization is typically applied to individual systems and tends to vary across different contexts. To address this research gap, the research conducts a large-scale empirical investigation aimed at establishing a generalized prioritization of code smells based on their impact on software quality and maintainability. Thirteen common smell types were examined across 35 open-source Java projects, analyzing their relation-

ships with 25 internal software quality metrics such as size, complexity, coupling, etc. as well as two maintainability metrics such as change-proneness and fault-proneness. The study also incorporated perception-based insights from developers to compare subjective judgments with metric-driven analysis, uncovering notable discrepancies between these two. Finally, the research identified those code smells that exert the most detrimental effect on program comprehensibility, which is a key factor in reducing long-term maintenance costs.

The results demonstrate that code smells vary in their degree of impact. High-priority smells include *Anti Singleton, Long Parameter List, Class Data Should Be Private,* and *Blob*. Moderate-priority smells consist of *Long Method, Complex Class, Large Class, Refused Parent Bequest,* and *Spaghetti Code*. Finally, low-priority smells are *Speculative Generality, Many Field Attributes But Not Complex, Base Class Should Be Abstract,* and *Lazy Class*. Alignment between developers' perceptions and system analysis was observed for 61.54% of smell types, while 38.46% diverged, highlighting the need for developers to refine prioritization strategies to minimize maintenance costs. Finally, smells such as *Long Method, Spaghetti Code, Refused Parent Bequest,* and *Anti Singleton* were found to significantly degrade comprehensibility, with a correlation coefficient of -0.56 indicating that higher impact scores reduce comprehensibility.

To summarize, the contributions of this research include an empirically derived prioritization of code smells, a comparative analysis of perception-based and metric-driven prioritization, and a synthesized dataset of 74,253 smelly files across 13 types. Collectively, these findings provide practical guidance for developers to focus refactoring on the most impactful smells, improve software quality, maintainability, and reduce long-term costs, while also supporting researchers in developing innovative refactoring tools and advancing future work in code smell management.

ACKNOWLEDGMENTS

I express my gratitude to Allah for giving me the opportunity and granting me the ability to complete my thesis work properly.

The completion of this thesis would not have been possible without the invaluable guidance of my supervisors and the unwavering support of my family. First and foremost, I would like to express my deepest gratitude to my principal supervisor, Dr. Kazi Muheymin-Us-Sakib, for his exceptional mentorship, advice, and encouragement. Dr. Sakib was always available to address my uncertainties, and I am truly thankful for his continuous support and insightful feedback on both this thesis and several research papers.

I would like to sincerely thank my supervisor, Professor Md. Mahbubul Alam Joarder, for his invaluable guidance. His positivity, support, and insightful discussions have greatly contributed to the smooth progression of my research.

I would like to express my appreciation to the faculty and thesis committee members of the Institute of Information Technology, University of Dhaka, for their

valuable feedback, which has significantly enhanced the quality of my thesis.

I am deeply grateful to my friends, particularly Mr. Abdus Satter and Mr. Rashed Rabby Riyadh. I owe a special debt of gratitude to Mr. Satter for his unwavering support from the very beginning of my PhD journey, both academically and personally. Thank you, Mr. Satter, for being such a reliable friend and confidant. I am also thankful to Mr. Riyadh for his valuable assistance during the early stages of my PhD. I could always rely on both of them for open and comfortable discussions on any issue. I would also like to extend my thanks to Mr. Toukir Ahammed for his valuable comments, which played a significant role in advancing my work.

I gratefully acknowledge the funding support provided by the Information and Communication Technology (ICT) Division, Ministry of Posts, Telecommunications and Information Technology, Dhaka, Bangladesh, through Grant/Fellowship Number: 56.00.0000.052.33.005.21-2, dated 18-01-2022.

Last but certainly not least, I would like to take this opportunity to express my deepest gratitude to my parents, Mr. Md. Ismail Mollah and Mrs. Mahmuda Khanam. Throughout my life, they have consistently prioritized my well-being, education, and happiness above their own, and for that, I am eternally grateful. Their unconditional love and unwavering support have been the foundation of my achievements. I am also profoundly thankful to my wife, Humayra Khanam Anny. She has been my constant source of strength, especially during the most challenging times of my journey. Her encouragement pushed me forward when I needed it most, and her patience, love, and understanding have been invaluable. Thank you, Anny, for standing by me with such grace and resilience.

PUBLICATIONS AND SUBMITTED PAPERS

Journal Publications:

1. Md. Masudur Rahman, Toukir Ahammed, Md. Mahbubul Alam Joarder and Kazi Sakib (2025). "Do Internal Software Metrics Have Relationship with Fault-proneness and Change-proneness?" *Programming and Computer Software, Springer.* (**Quartile: Q4, Impact Factor: 0.5**). *Corresponds to Chapter 4.*

2. Md. Masudur Rahman, Abdus Satter, Md. Mahbubul Alam Joarder and Kazi Sakib (2024). "Software metric based impact analysis of code smells – a large scale empirical study." *Software: Practice and Experience, Wiley.* (**Quartile: Q2, Impact Factor: 2.7**). *Corresponds to Chapter 5.*

3. Md. Masudur Rahman, Toukir Ahammed, Md. Mahbubul Alam Joarder and Kazi Sakib (2025). "Code Smell Prioritization based on the Impact on Software Maintainability." *Results in Engineering, Elsevier.* (**Quartile: Q1, Impact Factor: 7.9**). [Submitted]. *Corresponds to Chapter 6 and 7.*

4. Md. Masudur Rahman, Md. Mahbubul Alam Joarder and Kazi Sakib (2025). "Understanding the relationship of code smells with program comprehensibility – an empirical study." *Software: Practice and Experience, Wiley.* (**Quartile: Q2, Impact Factor: 2.7**). [Submitted]. *Corresponds to Chapter 7 and 8.*

Conference Publications:

1. Md. Masudur Rahman, Abdus Satter, Md. Mahbubul Alam Joarder and Kazi Sakib (2022). "An empirical study on the occurrences of code smells in open source and industrial projects." *In Proceedings of the* 16^{th} *ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pp. 289-294. (**Core Rank: A**). *Corresponds to Chapter 4.*

2. Md. Masudur Rahman, Toukir Ahammed, Md. Mahbubul Alam Joarder and Kazi Sakib (2022). "Does Code Smell Frequency Have a Relationship with Fault-proneness?." *In Proceedings of the* 27^{th} *ACM/IEEE International Conference on Evaluation and Assessment in Software Engineering, (EASE)*, pp. 261-262. (**Core Rank: A**). *Corresponds to Chapter 6.*

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABBREVIATIONS

ACS	Average Comprehensibility Score
ACP	Average Change-Proneness
AFP	Average Fault-Proneness
AIS	Average Impact Score
API	Application Programming Interface
AS	Anti Singleton
ASF	Average Smell Frequency
BCSBA	Base Class Should Be Abstract
BLOB	Blob
CC	Complex Class
CDSBP	Class Data Should Be Private
CM	Code Metrics
CQ	Code Quality
CP	Change-Proneness
CPU	Central Processing Unit
FP	Fault-Proneness

GC	God Class
IDS	Inverse of Developer Score
IEC	International Electrotechnical Commission
ISF	Inverse Smell Frequency
ISO	International Organization for Standardization
LC	Large Class
LPL	Long Parameter List
LM	Long Method
LOC	Lines of Code
LzC	Lazy Class
MFNBC	Many Field Attributes But Not Complex
OO	Object-Oriented
OOP	Object-Oriented Programming
OSS	Open Source System
RPB	Refused Parent Bequest
SC	Spaghetti Code
SDLC	Software Development Life Cycle
SF	Smell Frequency
SG	Speculative Generality
SLR	Systematic Literature Review
SRP	Single Responsibility Principle
TCP	Total Change-Proneness
TFP	Total Fault-Proneness
TSF	Total Smell Frequency

CHAPTER 1

INTRODUCTION

Code smell is an important design issue that makes software code more complex, leading to decreased software quality and long-term maintainability. It is necessary to detect and remove code smells from the software systems to improve their quality and maintainability. With various types of code smells such as *Long Method, God Class, and Long Parameter List* – not all have the same impact, as each affects software design and maintainability differently. Moreover, it is not possible for software developers to work with all the types of code smell, as it requires a huge time which results in increasing maintenance cost. Therefore, prioritizing code smells presents a reduced number of the smells to users, making the codebase easier to work with. As a result, this leads to improved code quality and better long-term maintainability. So, refactoring code smells based on their priority levels helps to achieve this by improving coding design. The thesis is

about prioritization of the code smell types that helps in minimizing maintenance cost. This chapter provides an introduction to the work presented in this thesis. Section 1.1 presents the overview of the thesis work. In Section 1.2, the motivation of the research is described. Section 1.3 describes the state-of-the-art approaches about the research. Research questions and contribution are presented in Section 1.4 and Section 1.5 respectively. Finally, the overall organization of this thesis is presented in Section 1.6.

1.1 Introduction

Code smells refer to poor coding practices that increase complexity and make code harder for developers to understand and maintain [1]. These smells may be introduced either intentionally or unintentionally as a result of their inefficient coding practices. Moreover, a software system is usually developed and deployed within strict time frames, requiring developers to meet tight deadlines. Such constraints can lead to the introduction of code smells, which indicate poor code quality [2]. Martin Fowler, known as the 'father of code smells', identified 22 types of code smells such as *Long Method, God Class, Long Parameter List,* etc. [1]. Actually, code smells do not hamper programs performance or accuracy, but decrease program comprehensibility. Therefore, it is quite difficult and time consuming to adapt change requirements [3] in the maintenance phase for the software system containing the code smells. So, code smells should be removed or reduced from the system to improve code quality and maintainability.

Refactoring is a very important technique for developers to remove code smells and reduce maintenance cost by enhancing code quality and maintainability. Refactoring is the process of improving the internal structure of the code without affecting its external behavior[1]. It not only enhances various aspects of software quality but also increases productivity and maintainability [4, 5, 6]. There are a

number of refactoring operations such as *Extract Method, Extract Class, Replace Parameters with Object,* etc. mentioned in the literature [1]. Although refactoring does not change features or functionalities of a software system, it is a significant technique for developers to improve code quality in the maintenance phase. Refactoring makes a software system easier to understand by improving design and removing code smells.

An example of code smell and its corresponding refactored version are shown in Figure 1.1, where a method named *updateUser()* takes too many parameters (greater than four) to perform some tasks (top portion of the figure). Although this method works correctly, it is not easy to understand the method at a glance, because of the long list of parameters. This practice is known as *Long Parameter List* code smell [1]. Since *updateUser()* contains *Long Parameter List* smell, it is a 'smelly method' and therefore, it should be refactored. In Figure 1.1 (bottom portion), a refactored version of the code is presented through *Replace Parameters with Object* refactoring technique [1], in which, *updateUser()* is smell-free as well as comprehensible.

*** Before Refactoring ***

```
// Smelly Method
public void updateUser(String name, int age, String address, String email, String phone)
{
    // Code for updating a user
}
```

*** After Refactoring ***

```
//Smell-free Method
public void updateUser(User user)
{
    // Code for updating a user
}

class User{
    String name;
    int age;
    String address;
    String email;
    String phone;
}
```

Figure 1.1: Code Smell and Refactoring

Software quality is a key measure of a system's maintainability [7]. According to the ISO/IEC 9126 standard (International Organization for Standardization [8]), software quality is defined as a set of attributes or metrics that describe and assess the quality of a software system. These attributes are grouped into two categories [7]: (i) *internal quality attributes*, which are measurable directly from the source code (e.g., lines of code), and (ii) *external quality attributes*, which are evaluated indirectly through the system's performance (e.g., maintainability). It is noted that in this thesis, we refer the *internal quality attributes* as *software quality metrics* or only *software metrics*, and *external quality attributes* as *maintainability metrics*. Assessing both internal and external attributes provides developers with a comprehensive understanding of the software's overall quality and maintainability. However, the presence of code smells has a negative impact on software quality and maintainability [1, 9, 10]. To improve software quality and maintainability, code smells are significant concerns for the developers and these should be removed consciously.

Program comprehensibility is a key aspect in software maintenance phase and its maintainability [11]. Program comprehension refers to the process of understanding an existing software system in order to plan, design, code, and test modifications [3]. It is a significant activity that consumes approximately more than 50% of the total effort expended throughout the life cycle of a software system [11, 3, 12]. A clear understanding of the system is essential for effective maintenance, reducing the risk of introducing defects and lowering overall software costs. Conversely, The presence of code smells is indicative of design flaws that hinder program understanding, complicate the maintenance process, and increase maintenance costs.

Code smells have negative impact on the software quality and its maintainability [9]. Software quality can be measured by software metrics such as size,

complexity, coupling, etc. [13, 14] and maintainability can be measured by fault-proneness and change-proneness [10]. So, code smells are important design factors that affect both the quality and maintainability metrics of a software system. Minimizing these metrics through removing important or impacted code smells is one of the goals of developers in the maintenance phase of the system. Therefore, it is necessary to identify the relationship between code smells and these software metrics so that the metrics impacted by a particular code smell can be recognized. It is very difficult for the developers to remove or work with all the code smells at a time from a developed system, as it is a time consuming task [15]. It is also not wise to refactor solely based on known or frequently occurring code smells, as their impact on software quality and maintainability can vary significantly. Not all code smells have the same level of severity, and some may have minimal or negligible effects on system performance, maintainability, or readability. So, it is essential to identify such code smells having higher impact on the software metrics to be refactored. It will help developers to refactor not only the known smells by definition but also the impactful ones which ultimately improve the design quality and maintainability of the system.

Code smell prioritization is an important process that involves determining the order in which different types of code smells should be refactored to enhance the maintainability and overall quality of a software system. Given the large number of code smells, each with differing levels of impact on the system, it is important to evaluate their relative significance. Distinguishing between high, moderate, and low-priority smell types helps in effectively prioritizing refactoring efforts. Not all of the code smells carry the same weight; some may cause more significant issues, such as making the codebase harder to maintain or comprehend, while others may have a more negligible effect. Prioritizing the code smells helps developers focus their efforts on the most critical areas of the codebase, ensuring that the most harmful or disruptive smells are addressed first. Thus, smell prioritization helps

in reducing maintenance cost through improving quality of the source code.

Code smells are problems resulting from poor design practices and refer to design situations that adversely affect the software maintenance. It is also known as anti-patterns [16], anomalies [1], design flaws [17], or bad smells [18]. Previous studies have investigated the relationship between code smells and the two significant maintainability metrics – change-proneness and fault-proneness [19, 10]. Very few researches have been found about their impact on internal software metrics such as size, complexity, coupling, etc. [9]. Moreover, some works have focused on trying to reduce the number of code smells by prioritizing or filtering them [20]. For instance, subjective factors such as developers' perceived criticality [21] and developers' relevance [22] have been used to prioritize code smells. Sae-Lim et al. [23] proposed a technique to prioritize code smells by considering developers' current coding context.

We have identified a couple of research gaps in the existing literature. Firstly, code smell prioritization has been performed in the literature which is mostly system (or project) specific and requires developer's intervention. That is, prioritization of the smells is possible for a developed system, and it varies from project to project. In other words, there is a lack of generic smell prioritization approaches that are system-independent. The system-independent smell prioritization helps new developers to recognize and work with the prioritized smell types in advance. Secondly, there exists no work about the analysis of relationship between code smells and program comprehensibility which is useful to reduce maintenance cost.

1.2 Motivation of the Research

In recent years, code smells have become a well-recognized concept for identifying patterns or characteristics in software design that may hinder future development and system maintenance [24]. In object-oriented systems, code smells indicate de-

sign issues that make the software harder to maintain, tightly coupled, and more complex [1]. While code smells do not directly impact system performance or accuracy, they degrade code quality and maintainability. To facilitate maintenance, code smells should be eliminated or minimized, that results in easing the workload of software engineers.

Code smells have negative impact on the quality of a software system that affect its maintainability [9]. Software quality can be measured by internal software quality metrics such as size, complexity, coupling, etc. [13, 14]. So, code smells are important design factors that affect the quality metrics of a software system. Minimizing these metrics through removing important or impacted code smells is one of the goals of developers in the maintenance phase of the system. Therefore, it is necessary to identify the relationship between code smells and software quality metrics so that the metrics impacted by a particular smell can be recognized to refactor and improve the software quality.

Software maintenance is a crucial phase of the *Software Development Life Cycle (SDLC)*, accounting for approximately 67% of the total software cost [3, 25]. However, these costs continue to rise and can sometimes reach up to 90% of the total costs [26, 27, 27]. Moreover, approximately 50% of the total maintenance effort spends in this phase to improve its maintainability [28]. Code smells have a significant impact on two important maintainability metrics – change-proneness and fault-proneness, which directly influence developers' activities [19]. Change-proneness refers to the likelihood that a software system will be modified to meet evolving requirements. On the other hand, fault-proneness refers to the likelihood that a software system will contain defects or bugs. It is stated that 90% of the total software costs is required in the maintenance and evolution of large-scale systems [29]. Code smells can make code more difficult to change and increase the likelihood of introducing new faults by making code harder to understand and

maintain [10, 19]. Consequently, this can increase the time and effort required for making changes and resolving faults, leading to cost overruns and slowing down the development and maintenance processes. It is essential to identify the relationship between code smells and these maintainability metrics so that impactful code smells can be refactored to improve its maintainability.

Indeed, with the vast number of code smells present, identifying and refactoring these manually is a challenging task for software developers. Attempting to address all the code smells at once, especially in a developed system, can be extremely time-consuming and impractical. Furthermore, focusing only on commonly known or frequent code smell types may not be the most effective approach, as not all smell types have the same impact on software quality and maintainability. Some smell types may have a lower impact on these factors, and hence do not require immediate attention. To minimize maintenance cost, developers need to identify and prioritize code smells that have the most significant impact on software quality and maintainability. By focusing on these high impactful smells, developers can address not only the familiar ones but also those that, if left untreated, could hinder the system's performance and longevity. This prioritization helps to improve the quality and maintainability of the software more effectively. Additionally, prioritizing code smells plays a critical role in reducing maintenance costs and supporting software engineers in their efforts to maintain a cleaner and more sustainable codebase. This approach allows developers to maximize their refactoring efforts and achieve higher software quality with fewer resources.

Usually, developers introduce code smells due to less awareness of design and program comprehensibility [1]. Therefore, it is necessary to enlighten them about the existence of code smells, especially the impactful ones. In this study, a code smell is considered highly impactful if it strongly related with the software quality and/or maintainability metrics, and less impactful if the relation is weak. If a de-

veloper intends to reuse code from an existing system and is aware of the presence of impactful code smells, they can refactor these smells (if applicable) before integrating the code into their own system[2]. This proactive approach enhances the quality and maintainability of the system in the long term. Moreover, providing developers with a short list of impactful code smells, as opposed to all possible smells or only the well-known ones, helps them to focus and prioritize effectively during development and maintenance.

The importance of program comprehension becomes even more evident during maintenance, which typically involves modifying parts of the system to fix bugs, improve functionality, or adapt to new requirements. Without a complete and accurate understanding of the system's architecture, components, and interactions, any changes made are likely to introduce new bugs, degrade system quality, or cause reliability issues [3]. On the other hand, code smells refer to design flaws in a program that make it harder to comprehend and maintain [1]. These smells are generally recognized as patterns in the program that reduce comprehensibility and increase the likelihood of errors [30]. If these issues are not addressed through refactoring at an early stage, future changes can introduce more defects and increase the maintenance cost [3]. Therefore, code smells are a barrier to program comprehension, and there exists a negative relationship in terms of correlation between them. Thus, the findings of the study will help developers to refactor those impactful smells on priority basis to reduce maintenance cost through improving program comprehensibility.

1.3 Issues in State-of-the-Art Approaches

Code smells can negatively impact various aspects of code quality and maintainability, which may contribute to the introduction of faults [1]. Beck and Fowler identified 22 distinct types of code smells and proposed their corresponding refac-

toring strategies to improve code design [1]. Since the introduction of these 22 code smell types, extensive research has been conducted on code smells, exploring their implications and mitigation strategies. Over the past two decades, code smells have attracted significant attention from both software engineering practitioners and researchers, highlighting their importance in maintaining high-quality and maintainable software systems [31].

Previous research has empirically examined the impact of individual code smells on various aspects of software maintainability, including fault occurrence [32, 33, 34], maintenance effort [35, 36, 37], and change proneness [38, 39, 40]. In a large-scale empirical study, Palomba et al. [10] confirmed that classes affected by code smells exhibit higher change-proneness and fault-proneness compared to those without smells. Further studies have explored the impact of co-occurring code smells on software maintenance [41, 42]. Zhu et al. [40] investigated the correlation between code smells and fundamental source code operations such as file addition, deletion, and modification. Analyzing 58 release versions across four Java projects, they concluded that code smells are strongly associated with file modifications, while showing weaker correlations with file additions and deletions. Abbes et al. [37] demonstrated that the presence of multiple code smells within the same source code segment adversely affects program comprehensibility. Kádár et al. observed that, in practice, classes with poor maintainability undergo more refactorings compared to those with higher technical quality [43, 44]. For example, highly coupled classes are more likely to be refactored than those with lower coupling [45, 46, 47].

Several studies have been conducted to prioritize important code smells. Vidal et al. [48] proposed a tool called JSpIRIT that prioritizes code smells based on multiple criteria such as relevance of the smells, history of the system, etc. Several other studies used developers' perceived criticality and preferences to pri-

oritize code smells [21, 49]. In the recent years, researchers have worked on the evolution of code smells [50, 51]. They investigated that code smells and their associated design problems live in the software system for a long time after these are introduced. Moreover, it is interesting from their analysis that very few smells are refactored intentionally. In another study, M. Tufano et al. stated that only 9% smells are removed as a direct consequence of refactoring operations where 80% smells survived in the systems [52]. A. Rani et al. illustrated that feature envy and god class code smells occur mostly after analysing four Java systems [53]. However, very few researches have been found about their impact on software quality metrics such as size, complexity, coupling, etc. [9]. Moreover, program comprehensibility plays a significant role in software development and maintenance, as it is required 58% to 70% of the time in reading source code [12]. Griffith et al. proposed an automated tool that refactor code smells to improve the code comprehensibility [54].

Refactoring is employed as a technique to remove code smells and improve code quality. Therefore, a number of tools have been developed for refactoring as well as its suggestions over the years [55, 56]. F. Palomba et al. also analyzed the relation of refactoring on changeability, that is, fault repairing, feature introduction, and general maintenance [57]. On the other hand, A. Brito et al. introduced a novel concept of refactoring graphs from which refactoring nature such as refactoring types, commits related to refactoring, developers involved refactoring, etc. can be understood over time [58]. E.A. AlOmar et al. analyzed the relationship between refactoring and quality attributes [59]. They found that while refactoring, most often developer's intention is to optimize these quality attributes such as cohesion, coupling, and complexity rather than encapsulation, abstraction, polymorphism, and design size. In another interesting study, C. Tavares et al. analyzed the impact of refactoring on code smells where it is found that refactoring operations remove smells as well as introduce another ones [60].

As mentioned above, these studies emphasize the importance of removing code smells through refactoring to improve code quality and simplify maintenance activities. They also examine how code smells impact software quality in ways that complicate maintenance efforts [61]. Over the years, various tools have been developed to automate code smell detection and refactoring. However, there has been limited research focused on identifying specific code smells that most significantly impact software quality and maintainability by analyzing real software systems. Additionally, prioritizing these code smells is crucial to minimize maintenance costs, as not all smells hold the same level of impact. Consequently, more research is needed in this area to support software practitioners and researchers in enhancing software quality and maintainability while reducing maintenance cost.

1.4 Research Questions

Existing research shows a range of approaches to code smells, their refactoring, and their impact on software quality and maintainability. However, manually identifying the most significant code smells from the many possible types is a challenging and time-consuming task for software engineers. This manual approach becomes a barrier to maintaining software at lower cost. To facilitate maintenance efforts, it is essential to improve code quality and reduce maintenance costs.

Prioritizing code smells is a key strategy for achieving the goal of maintaining software at lower cost. It allows software engineers to concentrate their efforts on the smells that have the most significant impact on software quality and maintainability. By focusing on high-priority code smell types, the engineers can allocate their refactoring efforts more efficiently, leading to more impactful improvements in software quality. To the best of the author's knowledge, no existing work addresses maintenance cost reduction through prioritization of code smells. Consequently, this study leads to answer the following research questions.

RQ1: What is the impact of code smells on software quality attributes?

This research question aims to explore the relationship between code smells such as *God Class, Long Method, Long Parameter List,* etc. and software quality attributes such as *lines of code, complexity, coupling,* etc. By analyzing diverse software systems, the study seeks to identify which code smells most significantly affect these attributes. The findings will guide software engineers in refactoring the smell based on the impact analysis, ultimately supporting the development of higher quality software.

RQ2: What is the impact of code smells on software maintainability attributes?

This research question aims to explore and establish relationships between code smells and software maintainability attributes such as *change-proneness and fault-proneness*. By analyzing a variety of software systems, this study seeks to determine the code smells which have the most substantial impact on these maintainability attributes. Answering the research question will identify the impactful code smells based on the maintainability of a software system. This facilitates software engineers to refactor the code smells based on the impact analysis to develop a maintainable software.

RQ3: How to prioritize various types of code smells based on the impact analysis of code smells in RQ1 and RQ2?

Since there exists various types of code smells, prioritization of these becomes essential to enhance software quality and maintainability. Based on the combination of impact analysis in RQ1 and RQ2, this research question aims to prioritize code smells that are most relevant to effective maintenance activities. In particular, this RQ3 will provide three prioritization levels of code smell – *high, moderate* and *low*. This prioritized set of code smells will guide software engineers by high-

lighting the refactoring tasks that should take in advance. This will allow them to proactively address the most critical areas of the source code to enhance the system's long-term quality and maintainability.

RQ4: What is the impact of the set of prioritized code smells on maintenance cost?

The set of prioritized code smells not only assists software engineers to identify the important ones but also improves code quality and eases maintenance tasks. This research question seeks to analyze the impact of prioritized code smells on maintenance costs, with a particular focus on program comprehensibility – a vital factor in effective software maintenance. A clear understanding of the system is essential for efficient maintenance, as it reduces the likelihood of introducing defects and helps lower maintenance costs. However, the presence of code smells hinders program comprehension, complicates the maintenance process and increases maintenance costs. This research question aims to identify which levels of code smell prioritization most negatively impact comprehensibility. The answer to this research question will guide software engineers to focus refactoring efforts on the most harmful smells to improve program comprehensibility and reduce maintenance costs.

1.5 Contributions of the Research

The thesis provides a structured prioritization of 13 types of code smells, classified into three levels – *high, moderate,* and *low.* This prioritization is developed through a multi-phase empirical study. The first phase involves a detailed empirical study examining the relationship between 13 code smell types and 25 internal software attributes or metrics. This analysis identifies which smells most impact *lines of codes, code complexity, coupling,* and other internal metrics. In the second phase, the study investigates how the code smell types relate to two significant main-

tainability attributes or metrics – *change-proneness* and *fault-proneness*. This phase highlights which code smells are more likely to lead to frequent changes or faults. Based on these analyses, the study assesses the impact of each code smell and classifies them into prioritized levels. This ranking allows developers to focus on the most detrimental code smells to improve software maintainability and quality efficiently. Finally, the study evaluates how the prioritized set of code smells influences program comprehensibility. By addressing high impactful code smells, developers can improve code comprehensibility, which results in reducing maintenance costs. In a nutshell, the major contributions of this thesis are given below:

1. **Impact of code smells on software metrics:** The relationship between various types of code smells and internal software metrics such as *lines of code, complexity, coupling,* etc. has been thoroughly examined to identify the most impactful code smells. By inferring these correlations, software engineers can recognize which code smells most significantly degrade software quality. This targeted identification enables them to prioritize optimizing specific metrics, which results in enhancing the overall quality of the software. Additionally, understanding the impact of these code smells supports researchers in developing effective refactoring strategies and automated tools tailored to address the most critical issues identified through impact analysis.

2. **Impact of code smells on maintainability metrics:** The relationship between various types of code smells and key software maintainability metrics – *change-proneness* and *fault-proneness* has been analyzed to identify the code smells that most significantly affect maintainability. By establishing these correlations, software engineers can better understand which code smells contribute to a higher likelihood of changes and faults, thereby degrading the software's overall stability and ease of maintenance. This

targeted identification allows engineers to prioritize addressing specific code smells that negatively impact maintainability metrics, enhancing the software's robustness and long-term adaptability. Furthermore, understanding the influence of these impactful code smells aids researchers in devising effective refactoring strategies and developing automated tools that focus on mitigating critical issues highlighted through the impact analysis. Such tools and strategies not only support the improvement of software maintainability but also encourage better coding practices by guiding developers toward writing cleaner, more maintainable code. Also, by addressing these high impactful code smells, software engineers contribute to creating software that is not only resilient to frequent changes but also less prone to faults, leading to lower maintenance costs and higher software quality over time.

3. **Prioritization of code smells:** Based on both software quality and maintainability metrics, code smells are categorized into three priority levels – *high, moderate* and *low*. This prioritization enables developers to focus on the most impactful code smells first, allowing them to optimize key metrics to enhance both software quality and maintainability. By systematically addressing high priority code smells, software engineers can achieve significant improvements in code clarity, reduce fault-proneness and change-proneness. This will create a software system that is easier to maintain and more robust over time. This prioritization strategy not only aids in immediate code refinement but also guides long-term maintenance efforts, ensuring that resources are directed toward areas that will yield the greatest positive impact on software resilience and adaptability.

4. **Comparison between developers' perception and metric-based impact analysis of code smells:** The contribution of the thesis lies in examining expert developers' perceptions of code smell impacts on software

systems and comparing these perceptions with the findings of metric-based analysis. Understanding developers' perspectives on code smell types is essential, given their hands-on experience in software development and maintenance. By aligning developers' subjective insights with objective metric-based analyses, this investigation provides a comprehensive understanding of code smell significance, offering valuable guidance for targeted refactoring and system improvement efforts.

5. **Impact of code smells on program comprehensibility:** Program comprehensibility plays a critical role in determining the overall cost of software maintenance. One of the key factors impacting program comprehensibility is the presence of code smells. So, the relationship between the impact of various types of code smells and program comprehensibility has been analyzed to identify the smells that most significantly affect the maintenance cost. This analysis assists developers to refactor the smell types which ultimately reduces the long-term maintenance cost of the software.

6. **Synthesized Smelly Dataset:** This thesis contributes a comprehensive dataset comprising 74,253 smelly files, categorized into 13 distinct code smell types across 35 software systems. Each code smell type is stored separately, allowing for targeted analysis of specific smells within the dataset. This resource is intended to support the research community, offering a valuable foundation for further studies and enabling researchers to extend and build upon our findings. By making this dataset available, we aim to facilitate deeper exploration into code smells, aiding the development of refined refactoring strategies, automated detection tools, and broader analyses of code quality and maintainability.

1.6 Organization of the Thesis

This section gives an overview of the subsequent chapters of this thesis. The chapters are organized as follows –

- Chapter 2 describes the fundamental background information about code smells, software and maintainability metrics including necessary definitions, terminologies, tools and technologies used in this thesis.

- Chapter 3 extensively presents related literature reviews about impact of code smells and their prioritization techniques. The goal of this chapter is to find out the research gaps and limitations of existing state-of-the-art techniques, and to set the direction of our research.

- Chapter 4 describes how the set of various code smell types are selected in this thesis. The code smell types are selected and filtered based on their frequency analysis by conducting an empirical study.

- Chapter 5 presents an empirical study that identifies the relationship between code smells and software metrics to quantify the impact of these smells. Then, it describes how these smells are categorized into three impact levels –*high, moderate,* and *low*. Finally, a comparative analysis between the frequency and impact of code smells is demonstrated. Moreover, a detailed dataset generation process is shown in this chapter.

- Chapter 6 presents an empirical study that identifies the relationship between code smells and two significant maintainability metrics – *change-proneness* and *fault-proneness* to quantify the impact of these smells. Then, it describes how these smells are categorized into three impact levels –*high, moderate,* and *low*.

- Chapter 7 demonstrates how the code smells are prioritized based on the impact analysis of Chapter 5 and Chapter 6. It also discusses the expert

developers' perceptions about the impact on software systems and prioritization of the code smells. Finally, the chapter compares the developers' perception with the findings of the metric-based system analysis to reveal how developers are far behind from the empirical analysis.

- Chapter 8 discusses how the code smells are categorized based on their comprehensibility levels, and analyzes the relationship between the impact of code smells and program comprehensibility.

- Chapter 9 concludes the research with a summary of the achievements. This chapter also suggests some future directions for further research in this area.

CHAPTER 2

BACKGROUND STUDY

Prioritization of code smells is significant for software developers as it helps in minimizing maintenance cost and enhancing program comprehensibility. The code smells such as *Anti Singleton, Complex Class*, *Long Method*, etc. have negative impact on software quality and maintainability attributes such as cyclomatic complexity, coupling, fault-proneness, etc. These issues can lead to significant challenges in the long-term maintainability of software systems. By understanding the relationship between the smells and these attributes, developers can determine which smells to address first, as some may have a more severe impact on software systems in terms of quality and maintainability than others. The objective of this chapter is to provide a comprehensive background on prioritizing code smells. In this chapter, the foundational aspects such as code smells, their influence on software quality, program comprehensibility, and the challenges associated

with smell prioritization are discussed.

2.1 Code Smell

Code smells refer to indicators in the source code that may suggest deeper issues in the system's design or implementation. In 1999, coined by Martin Fowler and Kent Beck in their book, *Refactoring: Improving the Design of Existing Code*, code smells are not considered as bugs but can lead to poor maintainability, reduced performance, and increased complexity over time [1]. The smells do not make the code non-functional but indicate opportunities for improvement [62].

2.1.1 Definition of Code Smell

"*A code smell is a surface indication that usually corresponds to a deeper problem in software systems*" – Martin Fowler, the father of code smell [1]. While not inherently incorrect, code smells often reflect underlying issues such as poor design choices, technical debt [63], or lack of adherence to programming best practices [64]. Undesired design flaws, known as code smells or bad smells or smells, are widely considered as indicators of poor software quality and maintainability [65, 10]. Examples of code smells are *Anti Singleton, Long Method, Complex Class,* etc.

Code smells are usually not bugs or errors, these are not technically incorrect and do not prevent the program from functioning. These are structural characteristics of a software that may indicate a design problem making software difficult to evolve and maintain [66, 67]. Since these smells are responsible for poor quality and maintainability, they reduce program comprehensibility and increase maintenance effort, ultimately slowing down software development [68, 32]. It exists in the source code due to poor design that makes the software difficult to maintain, and hence it will be a cost intensive activity. Although code smells do not interfere

with the functionality, accuracy or performance of a software, it makes a code difficult to understand and change [41]. For instance, a long method with hundreds of statements might provide accurate outputs in real time, but it certainly is not easy for a developer to comprehend it. In the same way, system having high coupling and low cohesion makes difficult to update or accommodate changes. Therefore, code smells do not obstruct the users of a software, but those are big challenges for the developers.

2.1.2 Refactoring

The technique developers use to remove code smells from a software system is known as *refactoring* [1]. The term *'refactoring'* was coined by Opdyke in 1992 that is used to restructure source code to improve its quality [69]. *"Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure. It is a structured way to clean up code that decreases code complexity and maintenance cost [70]. In essence when you refactor you are improving the design of the code after it has been written."* – Martin Fowler [1]. Generally, it is a way of improving software design quality and minimizing the existence code smells. Refactoring has a significant impact on enhancing software quality, maintainability and readability [71, 72, 73, 74, 75]. It is also very effective for reducing defects and unintended changes in a software system [76, 77]. Examples of refactoring [78] are *move method, extract method, extract class,* etc.

2.1.3 Example of A Code Smell and Its Refactoring

A *Long Method* is a code smell that occurs when a single method grows too large and performs multiple tasks. This reduces code readability and makes the method difficult to understand, test, and maintain.

Problematic Code Example – *Long Method* Smelly Code:

The Listing 2.1 shows a code snippet having *Long Method* code smell. Here, the method *processOrder(Order order)* suffers from the smell as it contains too many statements with many branchings and iterations.

Listing 2.1: Long Method Code Example

```
1  public class OrderProcessor {
2    public void processOrder(Order order) {
3      // Validate order
4      if (order == null || order.getItems().isEmpty()) {
5            throw new IllegalArgumentException("Invalid order"
                 );
6      }
7      // Calculate total price
8      double totalPrice = 0;
9      for (Item item : order.getItems()) {
10       totalPrice += item.getPrice() * item.getQuantity();
11     }
12     // Apply discounts
13     if (order.isEligibleForDiscount()) {
14       totalPrice *= 0.9; // Apply 10% discount
15     }
16     // Print receipt
17     System.out.println("Order Summary:");
18     for (Item item : order.getItems()) {
19       System.out.println(item.getName() + " x " + item.
             getQuantity() + " = " + item.getPrice() * item.
             getQuantity());
20     }
21     System.out.println("Total: " + totalPrice);
```

```
22    }
23 }
```

Refactored Code Example:

The refactored version of the previous smelly code is shown in Listing 2.2. Here, the long method *processOrder(Order order)* is partitioned into multiple methods using *extract method* refactoring technique [79, 80].

Listing 2.2: Refactored Code Example

```
1  public class OrderProcessor {
2    public void processOrder(Order order) {
3      validateOrder(order);
4      double totalPrice = calculateTotalPrice(order);
5      printReceipt(order, totalPrice);
6    }
7    private void validateOrder(Order order) {
8      if (order == null || order.getItems().isEmpty()) {
9        throw new IllegalArgumentException("Invalid order");
10     }
11   }
12   private double calculateTotalPrice(Order order) {
13     double totalPrice = 0;
14     for (Item item : order.getItems()) {
15       totalPrice += item.getPrice() * item.getQuantity();
16     }
17     if (order.isEligibleForDiscount()) {
18       totalPrice *= 0.9; // Apply 10% discount
19     }
20     return totalPrice;
```

```
21    }
22    private void printReceipt(Order order, double totalPrice)
          {
23      System.out.println("Order Summary:");
24      for (Item item : order.getItems()) {
25        System.out.println(item.getName() + " x " + item.
              getQuantity() + " = " + item.getPrice() * item.
              getQuantity());
26      }
27      System.out.println("Total: " + totalPrice);
28    }
29  }
```

Explanation of Refactoring:

- **Extract Method:** The *processOrder* method is broken down into smaller as well as more focused methods: *validateOrder, calculateTotalPrice,* and *printReceipt.*
- **Improved Readability:** Each method now has a single responsibility [81], making the code easier to understand and maintain.
- **Reusability:** The refactored methods can be reused in other parts of the application if needed.

2.1.4 Types of Code Smells

According to the definition described by Martin Fowler, there are 22 types of code smells [1]. The most common code smell types are described below.

1. **Alternative Classes with Different Interfaces:** Alternative Classes with Different Interfaces refers to a scenario where two classes perform sim-

ilar tasks but expose different interfaces. This inconsistency can lead to confusion, as developers must learn and work with multiple interfaces for performing the same or related tasks. It complicates maintenance and reduces code consistency – the uniformity in coding style, structure, and behavior across the system – making the software harder to understand and extend.

2. **Comments:** Excessive or unnecessary comments that compensate for poorly written code. Ideally, the code itself should be self-explanatory.

3. **Data Class:** A Data Class is a class that primarily consists of fields and basic methods, such as *getters* and *setters*, to access them. These classes serve as simple containers for data, often relied upon by other classes. They lack additional functionality and cannot independently perform operations on the data they encapsulate.

4. **Data Clumps:** A Data Clump is a code smell where groups of data items that are often used together but are not encapsulated in their own object. This can make the code harder to understand, modify, and maintain over time.

5. **Divergent Change:** Divergent Change is a code smell that arises when a single class undergoes multiple, unrelated changes for different reasons. This indicates poor cohesion, as the class is responsible for too many different aspects of the system. When changes affect a class in various ways, it becomes harder to maintain, as modifications in one area may impact unrelated functionality within the same class. This lack of cohesion makes the system more fragile and increases the effort needed for future changes.

6. **Duplicated Code:** Duplicated Code is a code smell that occurs when the same or similar code appears in multiple places within an application. This

usually results from copy-paste programming, where developers copy a block of code and reuse it in different parts of the application instead of creating reusable functions or classes. Duplicated code increases the maintenance burden. When the same logic is repeated across multiple locations, any changes must be applied to each instance. This process is error-prone and can lead to inconsistencies throughout the codebase.

7. **Feature Envy:** Feature Envy is a code smell that happens when a method in one class seems more interested in the data or behavior of another class than its own. This typically indicates that the method is using the other class's fields or methods excessively, often to the detriment of the class it resides in. The result is a misallocation of responsibilities, leading to high coupling between classes and making the system harder to maintain.

8. **Inappropriate Intimacy:** Inappropriate intimacy refers to classes or modules that have overly close relationships, causing them to depend heavily on each other's internal details such as attributes or methods. This tight coupling can lead to issues such as violation of encapsulation, increased maintenance challenges, and difficulties in testing and understanding the code.

9. **Incomplete Library Class:** The code smell occurs when a library class does not fulfill the requirements of a user completely. That is, it is almost perfect for fulfilling the requirements but lacks some of the functionalities we want, leading developers to modify or wrap it.

10. **Large Class or God Class:** A Large Class or God Class is a code smell that occurs when a class becomes excessively large and is responsible for handling too many tasks or managing too many responsibilities. These classes tend to control or manipulate many other classes and often violate the principle of single responsibility. Due to their size and complexity, god classes are difficult to understand, test, and maintain. They lack proper cohesion, as

they are trying to solve multiple problems at once instead of focusing on one specific responsibility.

11. **Lazy Class:** A Lazy Class is a class that does too little – it does not have enough functionalities to justify its existence. It may have been created during the design phase with the expectation of future responsibilities that never materialized. Such a class might be unnecessary, and its functionalities could be made a part of another class.

12. **Long Method:** A Long Method is a code smell that occurs when a method becomes excessively large, meaning it contains too many lines of code or too many distinct operations, making it hard to read, understand, and maintain. Instead of focusing on a single, well-defined purpose, it tries to perform multiple responsibilities, violating the principle of single responsibility and often leading to tightly coupled logic.

13. **Long Parameter List:** Long Parameter List is a code smell that occurs when a method takes an excessive number of parameters, typically more than four. It becomes harder to keep track of what each parameter represents and how they interact. So, this can make the method difficult to understand, maintain, and test. It also increases the complexity of method calls, as developers need to provide numerous arguments each time the method is invoked.

14. **Message Chains:** Message Chains refers to a long sequence of method calls that occur one after another, such as *object.getA().getB().getC()*. Long sequences of methods calls indicate hidden dependencies by being intermediaries. The problem with this smell is that any change in the intermediate relationship forces the client code to change, as well as can make code difficult to understand and maintain.

15. **Middle Man:** Middle Man refers to a class that acts solely as a delegator, forwarding requests and responsibilities to other classes without adding significant value or functionality of its own. This type of class merely acts as an intermediary, which often leads to unnecessary complexity and reduced clarity in the code, as the delegation could be handled directly by the classes involved.

16. **Parallel Inheritance Hierarchies:** Parallel Inheritance Hierarchies occur when two inheritance hierarchies are interdependent, where a subclass in one hierarchy depends on a specific subclass in another hierarchy through composition. This results in a tightly coupled system, where changes in one hierarchy often necessitate corresponding changes in the other. This leads to increased complexity and difficulties in maintaining the system.

17. **Primitive Obsession:** Primitive Obsession is a code smell that occurs when primitive data types like integers, strings, or booleans are used to represent complex domain concepts. Instead of creating dedicated classes or data structures, developers rely on basic types, which can result in code that is harder to understand, maintain, and extend. This approach lacks the expressiveness and encapsulation of behavior that well-designed domain-specific classes offer.

18. **Refused Bequest:** Refused Bequest refers to a situation where a subclass inherits from a parent class but only uses a subset of the methods or functionality implemented in the parent. This indicates that the inheritance hierarchy is inappropriate, as the subclass is not fully utilizing the capabilities of the parent class. It can lead to unnecessary complexity and violates the Liskov Substitution Principle [81], as the subclass is expected to be interchangeable with the parent class but fails to do so.

19. **Shotgun Surgery:** Shotgun Surgery is a code smell that occurs when a sin-

gle change to a class requires making numerous small changes across many different classes. This indicates that the system has poor encapsulation, as the responsibility for a particular feature or behavior is spread out across multiple classes, rather than being contained within one. As a result, maintenance becomes more difficult because changes in one place ripple through the system, increasing the risk of errors and making it harder to track and understand dependencies.

20. **Speculative Generality:** Speculative Generality refers to the practice of implementing code with unnecessary complexity or at an overly granular level in anticipation of future needs that may never materialize. This smell is a form of over-engineering, which means designing software with more complexity or flexibility than currently needed. It is driven by speculative assumptions about potential future requirements that may never actually materialize, leading to unnecessary code, abstractions, or features that increase maintenance burden without immediate value.

21. **Switch Statements:** Switch Statements refer to the excessive use of conditional constructs like *switch* or *if-else* statements, which can indicate that the code is more procedural or structural rather than object-oriented (OO). When these conditionals are used for type or object checking, instead of leveraging polymorphism to delegate behavior to the appropriate objects, it violates core OO principles [81]. This approach leads to rigid, hard-to-maintain code and makes it difficult to extend or modify functionality without modifying the conditionals in multiple places.

22. **Temporary Field:** Temporary Fields refer to the fields in a class which are only relevant under specific circumstances or conditions. Outside of these conditions, the fields serve no purpose and can create unnecessary complexity. Such fields can make the class harder to understand, as they

introduce unnecessary state and contribute to confusion about the class's responsibilities.

More Code Smell Types: Besides the code smells defined by Martin Fowler, there exist some other code smells (or *anti-patterns* - design issues) in the literature [82, 19, 49, 10, 16]. These are Anti Singleton, Base Class Should Be Abstract, Blob, Class Data Should Be Private, Complex Class, Many Fields Attributes But Not Complex, Spaghetti Code, Base Class Knows Derived Class, Functional Decomposition, Swiss Army Knife, Traditional Breaker, etc. These are described as follows.

23. **Anti Singleton:** Anti Singleton refers to a class that provides mutable class variables, which can be accessed and modified globally. Unlike a true Singleton, which restricts instantiation to a single instance, the Anti Singleton exposes shared state across the system, leading to potential issues with concurrency, maintainability, and unpredictable behavior.

24. **Base Class Knows Derived Class:** The Base Class Knows Derived Class code smell occurs when a base class has knowledge of or dependencies on its derived classes. This violates the principle of abstraction [81], as the base class should be independent of its sub-classes, leading to tight coupling and reduced flexibility in the code.

25. **Base Class Should Be Abstract:** Base Class Should Be Abstract refers to a scenario where an inheritance hierarchy has base classes that are not abstract. The base class should be abstract to prevent it from being instantiated directly, allowing only concrete sub-classes to provide specific implementations. This ensures that the base class serves as a blueprint for other classes rather than being used as a standalone class.

26. **Blob:** A Blob refers to a class that is excessively large and lacks sufficient cohesion, meaning its responsibilities are loosely related or scattered, rather

than being focused on a single, well-defined purpose. It dominates the system by handling most of the processing and decision-making, often becoming overly complex and difficult to maintain. A Blob class is typically tightly coupled with data classes and performs multiple, unrelated tasks violating the Single Responsibility Principle (SRP) [81].

27. **Class Data Should Be Private:** Class Data Should Be Private refers to a class that exposes its fields directly, violating the principle of encapsulation. This lack of data hiding increases the risk of unintended modifications, and reduces maintainability.

28. **Complex Class:** Complex Class refers to a class that contains at least one large, overly complicated method – typically marked by high cyclomatic complexity and too many lines of code. This makes the class harder to read, understand, and maintain, increasing the likelihood of bugs and reducing overall code quality.

29. **Functional Decomposition:** Functional Decomposition is a code smell that occurs when a class is structured to perform a single function, breaking the object-oriented design principle of modeling real-world entities with behavior and state. This approach is commonly seen in code written by inexperienced developers, where procedural thinking is applied in an object-oriented context.

30. **Many Fields Attributes But Not Complex:** Many Fields Attributes But Not Complex refers to a class that, despite being relatively simple in design, contains an excessive number of fields, many of which are publicly accessible. This can lead to poor encapsulation, increased coupling, and a higher risk of unintended modifications.

31. **Spaghetti Code:** Spaghetti Code refers to poorly structured, disorganized

code that is hard to understand, maintain, or extend. It often appears as a class with long, monolithic methods that lack parameters, making it difficult to apply polymorphism and resulting in tightly coupled and rigid designs.

32. **Swiss Army Knife:** A Swiss Army Knife code smell refers to a class that contains multiple sets of methods, each serving unrelated and diverse functionalities. Instead of focusing on a single responsibility, the class tries to do too many things at once, leading to low cohesion and making the code harder to understand, test, and maintain.

33. **Traditional Breaker:** A Traditional Breaker code smell occurs when a subclass does not meaningfully specialize or extend the behavior of its superclass. Instead of adding value or new functionality, it may simply inherit methods without modification or override them in a way that breaks the intended polymorphic behavior, violating principles like the Liskov Substitution Principle [81].

The various types of code smells occur into the system due to different reasons such as time constraints of feature development, evolving software requirements, lack knowledge about coding best practices, and technical debts [2]. Refactoring to eliminate code smells is an essential practice in maintaining a healthy system that is easy to work with, adapt, and extend.

2.1.5 Why Do Code Smells Occur?

Code smells occur in the source code of a software system due to many reasons. Several key reasons are mentioned as follows [83, 84, 85, 86, 87, 88, 89, 90].

1. **Lack of skills and awareness:** One of the most common reasons for the emergence of code smells is a lack of technical skills and awareness among developers. Developers who are new to programming or unfamiliar with best

practices may not fully understand principles like clean code [91], design patterns [92], or object-oriented design [81]. As a result, they may write code that works functionally but is difficult to read, maintain, or scale over time. Without proper training or mentoring, these issues can accumulate quickly.

2. **Frequent changes in requirements:** In many software projects, requirements are constantly evolving due to changing business needs, customer feedback, or market conditions. While adaptability is crucial in software development, frequent and unplanned changes often lead to rushed updates or temporary fixes. These quick solutions may compromise architectural integrity and introduce code smells, especially when changes are made without considering the long-term structure and maintainability of the codebase.

3. **Programming language and technology constraints:** Sometimes the tools and technologies used in a project impose limitations that make it challenging to implement clean and efficient designs. Certain programming languages may lack modern features or enforce specific paradigms that hinder flexibility. Additionally, legacy systems or outdated technologies might not support best practices, forcing developers to resort to workarounds that introduce code smells into the codebase.

4. **Inappropriate or missing documentation:** Documentation plays a vital role in helping developers understand the purpose and structure of the code. When documentation is poorly written, outdated, or entirely missing, developers may struggle to grasp the original intent behind certain implementations. This often results in incorrect assumptions, redundant code, or poorly integrated changes – all of which contribute to code smells and technical debt [63].

5. **Ineffective development processes:** Development processes that lack

structure or discipline can lead to inconsistencies and bad practices in the codebase. For example, unclear workflows, weak code review processes, insufficient testing, and lack of version control can result in fragmented, error-prone code. Over time, these inefficiencies manifest as code smells that degrade overall software quality.

6. **Schedule pressure:** Tight deadlines and pressure to deliver quickly often force developers to focus on immediate results rather than long-term code health. Under such constraints, developers may cut corners, skip proper testing, or overlook refactoring, resulting in code that is functional but messy and fragile. This prioritization of speed over quality is a major contributor to the spread of code smells in fast-paced development environments.

7. **Prioritizing features over code quality:** In many organizations, business goals center around adding new features to satisfy customer needs or beat competitors to market. While feature development is important, constantly prioritizing it over code quality leads to technical debt [63]. Without dedicated time for refactoring or maintaining the existing codebase, the quality deteriorates, and code smells begin to accumulate.

8. **Poor human resources and team culture:** The people and culture behind a software project have a significant influence on code quality. Teams that lack collaboration, communication, or effective leadership often struggle to maintain clean and consistent code. Poorly managed teams with unclear roles, limited mentorship, or high turnover rates may develop poor coding habits and overlook best practices, leading to a codebase that contains numerous smells and inconsistencies.

All of these factors combined can lead to a decline in software maintainability and long-term quality.

2.1.6 Impact of Code Smells

Code smells can have a significant negative impact on software systems, affecting various aspects of development and maintenance [10, 19, 90]. Below are some key impacts:

1. **Reduced Maintainability:** Code smells make the source code difficult to understand and modify, leading to increased effort and time for debugging, fixing bugs, implementing changes, or adding new features.

2. **Increased Complexity:** The presence of smells adds unnecessary complexity, making the code difficult to comprehend and increasing the cognitive load on developers.

3. **Higher Risk of Bugs:** Poorly designed code, that is, the existence of code smells is more prone to errors. These smells often lead to subtle bugs that are harder to identify and fix, increasing the likelihood of system failures.

4. **Poor Scalability:** As the source code of a system grows, the issues caused by code smells multiply, making it difficult to extend or scale the system effectively.

5. **Technical Debt:** Code smells are a major contributor to technical debt [63, 93], which refers to the long-term cost of taking shortcuts in software design and implementation. These quick fixes may offer immediate results but often compromise code quality and maintainability. As code smells accumulate, the system becomes increasingly complex and harder to modify. Over time, this leads to greater effort and cost required for refactoring and improvements. Addressing technical debt early helps maintain a healthier, more sustainable codebase.

6. **Decreased Productivity:** Developers spend more time in software development and maintenance due to poorly written code, leading to reduced

productivity and slower delivery of new features.

7. **Lower Code Quality:** The presence of code smells often reflects a lack of adherence to good coding practices [91], resulting in a decline in overall code quality and increased risk of long-term maintenance issues.

8. **Higher Costs:** The presence of code smells badly impact on code quality, maintainability and comprehensibility, which results in increasing the cost of future changes.

While code smells do not always indicate immediate defects, they are early indicators of deeper structural problems in the software system. Addressing these smells through regular refactoring and adherence to clean coding principles can mitigate their impact and result in a more robust, maintainable, and high-quality software system.

2.2 Software Quality Metrics

Software metrics have been widely used in measuring software quality [94, 95]. Software quality is an important factor in measuring how much a software system is maintainable. In ISO/IEC 9126 set by the International Organization for Standardization standard, software quality is defined by a series of attributes or metrics that describe and assess the quality of a software system [8, 96]. These metrics are typically classified into two categories: *internal quality metrics* and *external quality metrics* [7]. This section discusses these two types of metrics related to code smells.

2.2.1 Internal Software Quality Metrics

The internal software quality metrics or attributes are those that can be directly measured from the source code and its structure. These metrics serve as in-

dicators of various aspects of software, including complexity, coupling, cohesion, abstraction, encapsulation, and documentation [14]. These metrics also emphasize technical aspects of the software and are measurable without executing the program. These internal software quality metrics are referred to as *Software Quality Metrics* or only *Software Metrics* in this thesis for ease of reading. The common 25 software metrics [14] used in this thesis are defined as follows.

1. **CountDeclMethod (NOM):**

   Definition: Total number of methods declared in a class.

   Explanation: Indicates the size and potential complexity of the class in terms of behavior.

   Example: A class with 10 declared methods has NOM = 10.

2. **CountDeclMethodDefault (NDM):**

   Definition: Number of methods with default (package–private) access.

   Explanation: Helps assess how many methods are accessible only within the same package.

   Example: A class with 3 methods without access modifiers has NDM = 3.

3. **CountDeclMethodPrivate (NPriM):**

   Definition: Number of methods marked as `private` in a class.

   Explanation: Reflects encapsulated behavior within the class, such as helper functions.

   Example: A class with 5 private helper methods has NPriM = 5.

4. **CountDeclMethodProtected (NProM):**

   Definition: Number of `protected` methods declared in a class.

   Explanation: Useful in understanding inheritance and subclass access.

   Example: A superclass with 4 protected methods has NProM = 4.

5. **CountDeclMethodPublic (NPM):**

Definition: Number of `public` methods in a class.

Explanation: Indicates the external interface of a class.

Example: An API class with 7 public methods has NPM = 7.

6. **CountStmt (NOS):**

   Definition: Total number of statements in the code.

   Explanation: Measures the amount of logic written, including executable and declarative code.

   Example: A function with 50 statements has NOS = 50.

7. **CountStmtDecl (NDS):**

   Definition: Number of declarative statements in the code.

   Explanation: Includes variable declarations and other structure-defining lines.

   Example: A class with 15 declared variables has NDS = 15.

8. **CountStmtExe (NExS):**

   Definition: Number of executable statements in the code.

   Explanation: Includes actual operations like loops, assignments, and method calls.

   Example: A method with 20 executable lines has NExS = 20.

9. **SumCyclomatic (CC):**

   Definition: Sum of the cyclomatic complexity of all methods in a class.

   Explanation: Reflects decision-making complexity and the number of paths through the code.

   Example: Five methods with complexities totaling 25 result in CC = 25.

10. **LCOM (Lack of Cohesion in Methods):**

    Definition: Measures how related the methods in a class are to one another.

    Explanation: Lower LCOM indicates better cohesion; higher values suggest

poor design.

Example: A cohesive class has a low LCOM.

11. **Depth of Inheritance Tree (DIT):**

    Definition: Maximum inheritance depth from the base class to the current class.

    Explanation: Indicates inheritance complexity and potential reuse.

    Example: A class at the third level in the hierarchy has DIT = 3.

12. **IFANIN:**

    Definition: Number of immediate base classes (superclass + interfaces) for a class.

    Explanation: Represents the breadth of inheritance and interface implementation.

    Example: A class implementing 2 interfaces and extending 1 base class has IFANIN = 3.

13. **Coupling Between Objects (CBO):**

    Definition: Number of classes to which a given class is coupled.

    Explanation: High coupling reduces modularity and increases dependency.

    Example: A class using 8 other classes has CBO = 8.

14. **Number of Children (NOCh):**

    Definition: Number of direct subclasses inheriting from a class.

    Explanation: Indicates the potential influence and reuse of a superclass.

    Example: A class with 6 subclasses has NOCh = 6.

15. **Response For a Class (RFC):**

    Definition: Total number of distinct methods that can be executed from a class.

    Explanation: Includes local methods and those called by them. Measures

behavior complexity.

Example: A class invoking 30 methods has RFC = 30.

16. **Number of Instance Methods (NIM):**

    Definition: Number of methods that can only be accessed via an instance, that is, non-static methods.

    Explanation: Indicates behavior specific to instances of the class.

    Example: A class with 5 instance methods has NIM = 5.

17. **Number of Instance Variables (NIV):**

    Definition: Number of fields accessible only through an instance of the class, that is, non-static fields.

    Explanation: Represents the per-instance data size of the class.

    Example: A class with 10 instance variables has NIV = 10.

18. **Weighted Methods per Class (WMC):**

    Definition: Sum of complexities of all methods in the class.

    Explanation: Indicates the total effort and complexity involved in implementing the class.

    Example: A class with methods of complexity 2, 3, and 4 has WMC = 9.

19. **Classes (NOC):**

    Definition: Total number of classes in the system.

    Explanation: A high NOC suggests a larger, potentially more complex project.

    Example: A system with 50 classes has NOC = 50.

20. **Files (NOF):**

    Definition: Total number of source code files in the system.

    Explanation: Gives an idea of how the system is organized and distributed.

    Example: A project with 200 source files has NOF = 200.

21. **Lines (NL):**

    Definition: Total lines of source code in the system, including code, comments, and blank lines.

    Explanation: Basic measure of project size.

    Example: A project with 10,000 total lines has NL = 10,000.

22. **Lines Blank (BLOC):**

    Definition: Number of blank lines in the source code.

    Explanation: Reflects code formatting and readability.

    Example: A file with 50 blank lines has BLOC = 50.

23. **Lines of Code (LOC):**

    Definition: Number of lines with actual executable code.

    Explanation: Excludes blank and comment lines; indicates implemented logic.

    Example: A file with 200 lines of executable code has LOC = 200.

24. **Lines Comment (NCL):**

    Definition: Number of comment lines.

    Explanation: Shows how well the code is documented.

    Example: A file with 80 comment lines has NCL = 80.

25. **RatioCommentToCode (RCTC):**

    Definition: Ratio of comment lines to lines of code (NCL / LOC).

    Explanation: Indicates the extent of documentation in the codebase.

    Example: A file with 100 LOC and 50 comment lines has RCTC = 0.5.

2.2.2 External Software Quality Metrics

The external software quality metrics or attributes are those that can be indirectly assessed based on the behavior and performance of the software. These metrics

are often evaluated based on end-user experience and operational outcomes.

The literature presents several external quality metrics that are essential for evaluating software systems [97, 9]. These include *Maintainability* – emphasizes the effort needed to modify a component for fault correction or adaptation to requirement changes. *Adaptability* – pertains to the ease of modifying a system or component for use in applications other than its original purpose. *Understandability* – evaluates how clear and comprehensible a software component's purpose and functionality are to its users. *Reusability* – measures the extent to which a component can be effectively employed in multiple software systems or for constructing other components with minimal adaptation. Finally, *Testability* – encompasses the attributes of software that determine the effort required to validate the product, ensuring it aligns with its specified requirements.

Among the various external software quality metrics, maintainability is a critical aspect that significantly influences the software maintenance phase of the Software Development Life Cycle (SDLC). This phase is particularly resource-intensive, consuming approximately 67% of the total SDLC time [3]. During maintenance, developers address various tasks, including fixing faults, modifying source code to accommodate changing requirements, adding new functionalities, and improving overall code quality. Approximately 50% to 80% of software costs are associated with the maintenance activities [98]. Therefore, the two key metrics *Change-proneness* and *Fault-proneness* are usually used to measure the maintainability of a software system and are significantly impacted by code smells [10]. These two metrics used in this thesis are referred to as *maintainability metrics*.

1. **Change-proneness:** Change-proneness refers to the likelihood that a software component such as a class, method, or module will undergo modifications over time [99, 10]. These changes may include feature enhancements, bug fixes, or performance improvements. In practice, change-proneness is of-

ten assessed using historical version control data (e.g., frequency of changes, number of revisions), code complexity metrics, and past defect trends.

Frequent changes to software components can lead to increased development costs and risks, especially if changes are not well-documented or thoroughly tested. Components with high change-proneness are often harder to maintain and may result in ripple effects, where a change in one part of the source code unintentionally affects other parts.

2. **Fault-proneness:** Fault-proneness measures the likelihood that a software component will contain defects or bugs [100, 10]. It reflects the overall reliability and quality of a component. Code that is difficult to understand, poorly structured, or has high complexity is more fault-prone because developers are more likely to introduce errors while making modifications.

   Fault-prone components negatively affect software maintainability and reliability, leading to more debugging and testing efforts. This can result in increased maintenance costs, reduced customer satisfaction, and higher risk of system failures.

2.3 Prioritization of Code Smells

Code smell prioritization is the systematic process of identifying, assessing, and ranking code smells based on their potential impact on the software's quality and maintainability. In other words, it is a significant process that involves determining the order in which different code smells should be refactored to enhance the quality and maintainability of the software system. The goal is to focus limited resources on addressing the most critical issues first, ensuring that the development team's efforts yield the maximum benefit in terms of code quality and project outcomes.

2.3.1 Importance of Code Smell Prioritization

In large-scale software systems, it is common to encounter numerous code smells scattered across the source code. To maintain software quality and maintainability, it is essential to detect code smells in advance and refactor them on priority basis [101, 102, 103]. Addressing all of them may not be feasible due to resource constraints, such as time, budget, or personnel [2, 103]. Therefore, it is crucial to prioritize these smells based on their impact of the software systems. The key importance of smell prioritization is described as follows.

1. **Efficient Resource Allocation:** Prioritizing code smells enables development teams to strategically focus their limited resources such as development time, maintenance effort, and computational tools on the most impactful areas of the source code. This prevents wasted effort on minor issues and ensures that critical problems that may affect system performance or stability are addressed first. As a result, teams can achieve greater efficiency and productivity throughout the software lifecycle.

2. **Risk and Change Mitigation:** Giving higher priority to severe code smells helps reduce the chances of critical failures or performance bottlenecks that may arise from poorly designed or deteriorated code segments. By resolving these issues early, the software becomes more stable and less prone to defects. Additionally, well-structured code tends to require fewer changes, which decreases the likelihood of introducing new bugs during the maintenance phase.

3. **Software Quality Improvements:** Addressing prioritized code smells helps maintaining and enhancing essential qualities of object-oriented software systems such as modularity, encapsulation, and data abstraction. These qualities contribute to cleaner architecture, better separation of concerns,

and increased reusability of code components. As a result, the overall quality, robustness, and adaptability of the software are significantly improved.

4. **Maintainability Improvements:** Prioritization of code smells supports the long-term sustainability of software by guiding developers to improve code maintainability. Clean and well-organized code is easier to understand, modify, and extend, especially as the project grows in size and complexity. This reduces technical debt [63] and makes it easier for new team members to understand the code or for the team to adapt to evolving project requirements.

2.3.2 Factors Influencing Prioritization

Prioritizing code smells requires evaluating various factors, such as design quality, maintainability, smell severity, developers' perception, etc. [103]. Among these several important factors are discussed as follows.

1. **Severity of the Smell:** The extent to which a code smell affects functionality, performance, or readability of the system. More severe smells tend to have a higher priority for resolution.

2. **Impact on Software Quality:** Code smells that negatively affect quality attributes such as complexity, size, coupling, cohesion, etc. are considered more critical and should be addressed first.

3. **Fault Density:** The probability that a code smell contributes to faults or defects in the software. Smells associated with high fault density are prioritized to reduce system errors.

4. **Change Density:** The frequency with which a code region is modified. Smells in frequently changed code are more likely to cause instability and are thus prioritized.

5. **Developer Relevant Context:** Developers' insights and experiences play a role in assessing the importance of a smell, based on its perceived risk or maintainability impact within the specific system context.

6. **Refactoring Cost:** The estimated effort, time, or resources required to eliminate a smell. Smells with lower refactoring cost and high impact are often addressed earlier.

2.3.3 Challenges in Code Smell Prioritization

Prioritizing code smells effectively is a complex task due to several inherent challenges [98]. These difficulties arise from the subjective nature of code smells, resource constraints, and dynamic project environments. Below are the key challenges in code smell prioritization, discussed in detail.

1. **Subjectivity:** Code smells often depend on interpretation, and what one developer considers a critical issue may be seen as a minor inconvenience by another. This subjectivity can lead to inconsistent prioritization.

2. **Tool Limitations:** Automated tools like SonarQube [104] are useful for identifying code smells, but they often lack the ability to understand the full context of a software system. This includes domain-specific requirements, maintenance concerns, and architectural or design constraints. As a result, they may produce false positives or overlook important smells that require deeper analysis.

3. **Overwhelming Volume of Code Smells:** In large or legacy systems, code smells may be diffusive, making it challenging to determine where to begin. Also, there exist variety types of smells impacting various aspect of code quality and maintainability aspects such as complexity, change-proneness, fault-proneness, etc.

4. **Resource Constraints:** Limited time, budget, and personnel often restrict the extent to which code smells can be addressed. Therefore, critical smells may be ignored due to the pressure to meet deadlines or prioritize new feature development over refactoring.

5. **Dynamic Project Needs:** Evolving project requirements and changing stakeholder expectations can alter the significance of certain code smells. For example, smells like *Complex Class, God Class,* or *Long Method* may become more critical as the system scales or new features are introduced. Therefore, prioritization must be continuously adjusted to align with the current goals and context of the project.

6. **Difficulty in Measuring Impact:** Quantifying the actual impact of a code smell on code quality, maintainability, performance, or technical debt is often challenging. Without clear metrics, it becomes difficult to justify prioritization decisions to allocate resources effectively.

By addressing these challenges through structured processes, code smell prioritization at an earlier stage can become an effective practice for industry practitioners in improving software quality, maintainability, extendibility and program comprehensibility [105].

2.4 Program Comprehensibility

Program comprehensibility is the degree to which a software system can be understood by developers, maintainers, and other stakeholders [3]. It refers to the ease with which the program's functionality, structure, and intent can be grasped. Comprehensibility is a critical aspect of software quality, as it directly affects maintainability, debugging, and the ability to extend the system effectively. A highly comprehensible program reduces cognitive effort, enabling developers to

work efficiently and minimizing the risk of errors. In contrast, poor program comprehensibility can lead to increased development and maintenance time, cost, higher error rates, and greater difficulty in development team members.

2.4.1 Importance of Program Comprehensibility

Program comprehensibility has significant importance to improve software maintainability. It is an important activity that consumes approximately more than 50% of the total effort expended throughout the life cycle of a software system [3, 12]. Several key importance of the program comprehensibility are describes as follows.

1. **Maintenance Facilitation:** Most software systems spend a significant portion of their lifecycle in the maintenance phase. Comprehensible code makes it easier to identify and fix bugs, implement changes, and update features.

2. **Developer Productivity Improvement:** Clear and well-structured code allows developers to quickly understand its functionality, reducing the time spent on analysis and increasing overall productivity.

3. **Team Collaboration Support:** In collaborative environments, comprehensible code ensures that all team members can work on the source code without extensive dependency on specific individuals.

4. **Technical Debt Reduction:** By promoting clear design and coding practices, program comprehensibility helps avoid the accumulation of technical debts [63], such as code smells, which can otherwise slow down development over time.

2.4.2 Factors Affecting Program Comprehensibility

Program comprehensibility requires evaluating various factors, such as code structure, documentation, developers' expertise, etc. [3]. Among these several important factors are discussed as follows.

1. **Code Structure and Design:** Software programs followed by design principles such as modularity, cohesion, and separation of concerns enhances comprehensibility. A well-structured code with clear boundaries between components is easier to understand and maintain.

2. **Coding Practices:** Consistent naming conventions, proper indentation, and meaningful comments improve readability. Avoidance of anti-patterns [19], such as code smells, overly complex logic or deep nesting, etc. reduces cognitive load.

3. **Quality of Documentation:** Adequate and proper documentation including API references, architectural diagrams or flowcharts, inline comments, user manual, program logic manual, etc. provides context and helps comprehension.

4. **Size and Complexity of Source Code:** Larger and more complex source code can be harder to understand, especially if they lack organization or are poorly documented.

5. **Developer Expertise and Familiarity:** The knowledge and experience of developers play a crucial role in their ability to comprehend the program. Familiarity with the domain, programming language, and coding standards also impacts understanding.

Program comprehensibility is a foundation of high-quality software development. By focusing on clear design, consistent coding practices, and effective documentation, software engineers can ensure that their code is easy to understand

and maintain. This, in turn, leads to improved productivity, reduced maintenance cost, and better collaboration among developers. As software complexity continues to grow, prioritizing program comprehensibility is essential for sustaining long-term success in software projects.

2.5 Summary

This chapter has laid the groundwork for the thesis by introducing fundamental concepts related to code smells and their various types. Since code smells can degrade the structure, readability, and overall quality of a software system, addressing them is essential for sustainable development. To support this, we examined key software quality and maintainability metrics, emphasizing their role in assessing and guiding improvements in source code. The chapter also highlighted the value of prioritizing code smells based on their impact, cost, and context to enable more strategic and efficient remediation. Furthermore, we discussed program comprehensibility as a crucial aspect of reducing long-term maintenance effort and ensuring system longevity. The next chapter will present a detailed review of existing literature on the effects of code smells and current approaches to their prioritization.

CHAPTER 3

LITERATURE REVIEW

After defining 22 types of code smells by Martin Fowler [1], extensive research has been conducted on these smells. Over the past two decades, the topic of code smells has gained significant attention from both researchers and practitioners [31]. Much of this research has focused on developing detection techniques and refactoring strategies for addressing these code smells [55, 55, 106]. For instance, Palomba et al. employed historical software release data to identify code smells [107, 106]. It is important to note that different code smells have varying impacts on software systems, leading to differing levels of importance. Consequently, numerous studies have explored the impacts of code smells on software quality, maintainability and strategies for their prioritization. This section provides an overview of the existing literature on these aspects.

3.1 Impact of Code Smells on Software Quality

Code smells are critical indicators of potential design issues that can significantly influence the quality of software. These structural flaws, while not immediately causing errors, may lead to increased maintenance costs, reduced performance, and decreased code readability over time. According to Lehman's second law, as projects grow, they tend to increase in complexity [108], which leads to the occurrence of new code smells in the evolving system [109]. As such, understanding their impact on software quality attributes is crucial for both developers and researchers. In this section, we review various studies that have analyzed how code smells affect key internal software quality attributes or software quality such as complexity, coupling, performance, size, etc.

Internal software metrics are commonly utilized to detect various types of code smells [55, 70, 110, 111, 112]. For instance, one notable tool leveraging such metrics is JDeodorant, an Eclipse plug-in designed to detect six specific types of code smells, namely – *god class, feature envy, state checking, type checking, long method* and *duplicate code*, through static code analysis [55]. The tool not only identifies these smells but also provides automated refactoring suggestions to address them, enabling developers to improve code quality efficiently. Another Eclipse plug-in, inCode identifies four types of code smell related to an improper distribution of intelligence among classes. Specifically these are – *god class*, *data class*, *code duplication* and *feature envy* [113].

Moha et al. introduced a rule-card-based approach for detecting code smells, which systematically identifies 18 different types of smells by defining a set of rules and thresholds for internal software metrics [82]. This approach involves the creation of rule cards, each tailored to a specific code smell, and specifies the relevant metrics and their corresponding threshold values that indicate the presence of a smell. This method enhances precision by explicitly mapping metric

combinations to specific smells, reducing ambiguity in detection. Furthermore, it improves reliability by formalizing the detection process, making it systematic and repeatable across different systems. By using this approach, developers and researchers can detect smells consistently and prioritize refactoring efforts more effectively.

Building on these methods, Sousa et al. utilized internal metrics to detect five distinct types of code smells and conducted an exploratory study examining the relationship between design patterns and these smells [114]. Their analysis revealed that certain design patterns, such as *adapter, proxy,* and *strategy*, frequently co-occur with specific code smells. For instance, the *adapter* pattern often displayed associations with smells like *Large Class* and *Divergent Change*, suggesting that while design patterns aim to enhance flexibility and reuse, their misuse or overuse can inadvertently introduce code smells. Apart from that, Azeem et al. emphasized on machine learning techniques to detect code smells in an SLR [115].

Martins et al. examined the refactoring practices of developers addressing 60 code smell co-occurrences over three months across five closed-source projects [116]. Their analysis focused on four key aspects: (i) The impact of code smell co-occurrences on internal quality attributes; (ii) Identifying the most harmful co-occurrences from developers' perspectives; (iii) Developers' perceptions of refactoring activities for removing code smell co-occurrences; and (iv) Challenges faced during the refactoring process. The findings revealed that refactoring certain co-occurrences, such as *dispersed coupling–god class*, led to improvements in quality attributes. However, developers found refactoring these co-occurrences challenging, primarily due to difficulties in understanding the code and the complexity of refactoring methods. Additionally, many developers expressed insecurities regarding the identification and refactoring of code smells and their co-occurrences.

These insights highlight the need for improved tools and strategies to support developers in managing code smell co-occurrences effectively.

Fontana et al. explored the relationships and co-occurrence among code smells to enhance their detection and prioritization for removal of them [63]. They emphasized focusing on the most critical smells, such as those that co-occur with others, as these may have a greater impact on software quality compared to isolated smells. The study highlighted that targeting specific smells can address particular quality attributes. For instance, addressing *Intensive Coupling, Dispersed Coupling,* and *Shotgun Surgery* can improve system coupling, while resolving *God Class, Brain Method,* and *Intensive Coupling* can enhance cohesion. This approach provides actionable insights beyond merely analyzing metric values. Additionally, they proposed three visualization tools, including a related code smell view, to analyze the detected smells and their impact on system design quality. These personalized views offer detailed insights into individual code smells, supporting developers in making informed refactoring decisions to improve software quality.

Smell detection and refactoring approaches use machine learning and genetic algorithms to improve system quality [117, 31, 118, 119, 120, 121]. Mhawish et al. proposed a code smell prediction framework that leverages machine learning techniques in combination with software metrics [122]. To enhance the interpretability of the machine learning models, they incorporated the Local Interpretable Model-Agnostic Explanations (LIME) algorithm, enabling a clearer understanding of the predictions made by the models and their underlying rationale. The framework also employs a genetic algorithm-based feature selection method to improve the predictive accuracy of the machine learning models. By identifying and selecting the most relevant features within each dataset, this approach reduces noise and enhances the performance of the models. The results of their study demonstrate the significant potential of machine learning techniques in predicting code smells.

Mayvan et al. proposed a methodology for detecting code smells using software quality metrics which is a multi-step process [123]. The approach starts by defining formal specifications for code smells based on relevant software metrics. These specifications are then applied to identify an initial set of candidate instances for each type of code smell. Once the candidates are identified, the methodology incorporates a refinement step where each instance is examined and compared against predefined refactoring scenarios specific to the corresponding code smell. This comparison helps eliminate false positives generated during the initial detection phase, ensuring a more accurate identification of genuine code smells. The effectiveness of this methodology was evaluated on four open-source systems, with results indicating a significant improvement in the accuracy and reliability of code smell detection. This enhanced detection process not only reduces the overhead of manual inspection but also provides a more precise foundation for subsequent refactoring efforts, which contributes to better software quality.

Mäntylä et al. explored the strength of relationships among code smells and demonstrated that several smells are interrelated [124]. For example, the *message chains* smell is correlated with the *middle man* smell. Understanding these relationships can help developers and researchers gain insights into how different smells are interconnected.

3.2 Impact of Code Smells on Software Maintainability

Software maintenance which is an important phase of software development life cycle (SDLC), involves 67% of the its total costs [3]. On the other hand, code smell is a design problem which affects software maintainability in this phase

[125]. These smells often emerge during change activities or when fixing software bugs [126]. Therefore, several studies have been conducted to analyze the impact of code smells on software maintainability.

Khomh et al. conducted a comprehensive investigation into the impact of code smells on two critical aspects of software maintainability: change-proneness and fault-proneness of classes within a system [19]. The study analyzed 13 different types of code smells across multiple releases of four open-source software systems. The researchers discovered that code smells negatively affect both change-proneness and fault-proneness, indicating a detrimental influence on software maintainability. Additionally, they observed that while some code smells, as suggested by their definitions, are associated with size-related metrics, size alone does not fully account for the higher change- and fault-proneness observed in the affected classes. In a separate study, Khomh et al. focused on the relationship between code smells and change-proneness [38, 127]. They demonstrated that classes containing code smells are significantly more prone to changes compared to those without any code smells. This finding reinforces the notion that the presence of code smells is a strong indicator of reduced software maintainability.

Several other studies have also provided evidence supporting the positive relationship between code smells and fault-proneness. Classes having code smells are more associated with the fault-proneness than classes without code smells [128]. For instance, Li and Shatnawi [32], and Olbrich et al. [39] demonstrated that the presence of code smells correlates with an increased likelihood of faults in software systems. In a systematic literature review (SLR), Cairo et al. identified that 24 code smells have greater influence on bugs, specially *God Class, Comments, Message Chains,* and *Feature Envy* code smells [109]. However, several code smells such as *Data Clumps, Speculative Generality, Middle Man, Data Class* and *Tradition Breaker* have less impact on software bugs [129, 128]. These findings highlight

the adverse effects of code smells on software maintainability.

Saboury et al. investigated the impact of code smells in JavaScript systems and observed their negative impact on fault-proneness [130]. Their study further reinforces the idea that code smells are a significant factor contributing to increased fault-proneness across different programming languages and environments. Expanding this scope to security-related issues, Yi Zhong et al. found that security smells-specific design or implementation choices that compromise security – also have a notable impact on faults in Android applications [131]. This study emphasizes the broader implications of smells, extending beyond maintainability to include security vulnerabilities as well.

Ahmed et al. examined the relationship between code smells and merge conflicts, specifically their impact on the likelihood of bugs in merged results [132]. The study found that program elements involved in merge conflicts contain, on average, three times more code smells than those not involved in conflicts. Of the 16 code smells co-occurring with merge conflicts, 12 were significantly associated with them. Among these, *God Class, Message Chains, Internal Duplication, Distorted Hierarchy* and *Refused Parent Bequest* were particularly notable.

Palomba et al. conducted a large-scale empirical study that supported findings from prior research, confirming that classes affected by code smells exhibit higher change-proneness and fault-proneness compared to those without smells [10]. Their investigation involved 13 different types of code smells and a dataset of 30 Java systems, providing robust evidence of the negative impact of code smells. The study revealed that long and complex code smells, such as *Long Method* and *Complex Class*, are particularly prone to changes and faults. Furthermore, they highlighted the criticality of classes affected by multiple smells, demonstrating that these classes are more problematic than those affected by a single smell.

In another empirical study, Rahman et al. explored the impact of different

code smells on fault-proneness, focusing on the frequency of occurrence [133]. They found that more frequent code smells, such as *Complex Class, Large Class* and *Long Parameter List*, etc. have a moderate impact on fault-proneness. In contrast, less frequent code smells, including *Anti Singleton, Blob* and *Class Data Should Be Private* exhibit a higher impact on fault-proneness. These findings suggest that even rare smells can pose significant risks to software quality and should not be overlooked during software maintenance and improvement efforts.

Abbes et al. investigated the impact of two code smells, namely – *Blob* and *Spaghetti Code* on program comprehension [37]. The results of their empirical study showed that the occurrence of one code smell in the source code of a system does not have a significant impact on developers' ability to comprehend the code. Instead, a combination of multiple code smells affecting the same source code strongly reduces program comprehensibility. In another study, Rahman et al. used IR methods to remove *Feature Envy* code smells [134], as well as showed that feature envy code smells have impact on software standards called naming conventions, which reflect program comprehension [135].

There exist several other studies on the interaction among different code smell types on the same code components. For example, Yamashita et al. investigated inter smell relations and their effects on maintenance problems using 13 smells on four Java systems [41]. They showed that developers face more difficulties while working on classes affected by more than one code smell. In another study, they identified the important maintainability factors such as inheritance, simplicity, logic spread, etc. from the software maintainer's perspective those are reflected by code smells [125]. In addition, they investigated inter-smell relations where they used static dependency analysis involving both open-source and industrial systems [136]. They observed that code smell interactions occur not only in the same file but also across coupled files having static dependencies.

Palomba et al. conducted an empirical study revealing that nearly 59% of smelly classes are affected by more than one type of code smell [42]. Their analysis highlighted that method-level smells often act as precursors to class-level smells, suggesting a hierarchical relationship where issues in methods propagate to the class level. This finding highlights the interconnected nature of smells and the importance of addressing method-level issues to mitigate broader system issues. Furthermore, Martins et al. investigated the co-occurrence of code smells, discovering that certain types of smells tend to occur together within a system [137, 116]. They demonstrated that the removal of these co-occurring smells can significantly reduce system complexity. This suggests that addressing smell co-occurrences, rather than focusing on individual smells in isolation, may have a more substantial impact on improving software maintainability and overall quality.

Sjoberg et al. conducted a study to evaluate the impact of 12 different code smells on the maintenance effort in software systems [24]. Their findings indicated that code smells do not have a significant direct association with maintenance effort. Instead, they observed that class size often plays a more critical role in influencing maintainability. Conversely, other studies have shown that developers generally perceive code smells related to long and complex source code as design problems. For instance, Palomba et al. [138] and Taibi et al. [139] found that while some code smells are recognized as indicators of design flaws, others are not consistently regarded as problematic by developers. This suggests variability in how different types of code smells are interpreted and addressed in practice.

Bavota et al. conducted an in-depth study to understand the relationship between refactoring practices and the presence of code smells [140]. Their findings revealed that while 42% of refactoring operations target code entities affected by code smells, only 7% of these refactorings effectively remove the smells. In their study, they analyzed 11 different types of code smells across three open-source

systems. Their investigation shed light on the motivations behind refactoring, suggesting that developers primarily engage in refactoring to enhance the overall maintainability of the source code, rather than to explicitly address code smells flagged by software quality metrics. Additionally, the study observed that refactoring activities predominantly focus on addressing a subset of well-known code smells, such as *Blob* and *Long Method*, which are widely recognized for their detrimental impact on software quality. This selective focus implies that developers may not consistently consider the full spectrum of code smells when planning and executing refactoring operations. In addition, Murphy-Hill et al. [141] showed that refactorings are performed frequently and almost 90% are manually, after conducting an extensive study using four data sets.

Code smells have negative impact on device's performance such as CPU, memory, and battery [142]. Alkandari et al. studied the impact of code refactoring on improving CPU and memory usage [143]. They analyzed three code smells – *HashMap Usage, Member Ignoring Method*, and *Slow Loop* using eight open-source mobile applications from GitHub. Each smell was refactored individually and collectively to evaluate resource usage on mobile devices. Measurements across five versions of the applications revealed that refactoring *HashMap Usage* and *Member Ignoring Method* improved CPU usage by 12.7% and 13.7% on average, respectively, while refactoring all three smells together improved memory usage by 7.1%. The study highlights the significant benefits of targeted refactoring in optimizing smartphone resource usage and improving software quality, providing guidelines for efficient coding practices.

Code smells affect various aspects of software, including quality, maintainability, reliability, testability, performance, and change-proneness [83]. Beyond these technical aspects, code smells also influence the software development process [83, 88]. For example, a high prevalence of smells can negatively impact the

morale and motivation of the development team, potentially leading to increased attrition rates.

Amandeep Kaur [9] conducted a systematic literature review and observed that different code smells have different effects on various software quality attributes, and hence the impact of code smells on software quality is not uniform. The study also observed that 92% of the primary studies (PSs) considered external quality attributes such as maintainability, readability, changeability, etc. to investigate the impact of code smells. On the other hand, only 8% of the PSs considered five internal quality attributes namely size, coupling, cohesion, complexity and inheritance to investigate the impact. Therefore, it is necessary to study more about code smells and their impact considering both the internal and external quality attributes to achieve more uniform information.

3.3 Prioritization of Code Smells

Code smells are important design factors but all of these are not equally significant for improving software quality and maintainability [9, 22]. Moreover, it is a good practice to prioritize impactful code smells and refactor them first to save time and effort [103, 144]. Therefore, important code smell types are essential to be identified so that developers can prioritize these in their refactoring activities. So, several studies have been found about code smell prioritization in the literature.

Vidal et al. proposed a tool called JSpIRIT that prioritizes code smells based on multiple criteria such as relevance of the smells, history of the system, etc. which can be customized by developers [48]. In addition, they proposed a semi-automated approach that prioritizes code smells based on three criteria – (i) past component modifications, (ii) important modifiability scenarios for the system, and (iii) relevance of the kind of smell [22, 145]. However, the third criterion is a subjective value that might vary from developer to developer. Several other

studies used developers' perceived criticality and preferences to prioritize code smells [21, 49]. However, these approaches require human intervention and might provide a different prioritized set when development personnel are changed.

Sae-Lim et al. proposed a technique for prioritizing code smells based on the developers' perception and current context [146, 147, 23]. This context is determined by the change descriptions of issues slated for implementation in a specific milestone. However, the approach may not be applicable if issue descriptions are unavailable. In another study, Islam et al. introduced a strategy to prioritize code smells by focusing on frequently used and change-prone areas of a system's source code [148]. Their approach combines business logic, heat maps or frequency data, commit history analysis, and the severity levels of code smells to identify and prioritize critical areas for refactoring.

Guggulothu et al. proposed a method to prioritize four code smells based on their inter-relationships and relevant static code metrics, such as lines of code and coupling [149, 150]. They identified the relevant metrics for each smell using a machine learning technique called feature selection.

Code smells are prioritized based on the their severity [63, 151, 152]. For instance, Fontana et al. introduced the concept of an intensity index for prioritizing six code smells [63]. This index quantifies the severity of each code smell instance relative to others of the same type, based on the detection rules and the distribution of corresponding metrics. This facilitates the identification of the most critical code smell instances. Similarly, Gupta et al. prioritized five code smells by leveraging their severity levels, determined using static code metrics [153].

Oliveira et al. suggested two promising heuristics for prioritizing code smells – density and diversity of smells [154] . These heuristics aim to guide developers in effectively focusing their refactoring efforts. In another study, A. Ouni et al. introduced a novel approach for automated refactoring that prioritizes risky code

smells during the correction process [49]. This method facilitates the removal of high-impact smells, results in improving overall code quality.

Alshammari et al. proposed a model to prioritize bad smells based on their impact on software maintainability using the Analytical Hierarchy Process (AHP) [155]. The model measures code before and after refactoring to assess the maintainability impact of bad smells. Validated against five bad smells and five open-source projects, the model also includes a visualization of the relationships between class maintainability and bad smell rankings. This prioritization model helps software practitioners focus their efforts and optimize resource utilization.

Some studies focused on prioritizing code smells to identify architectural problems [156, 157]. Vidal et al. proposed prioritize a group of inter-related code smells known as *agglomeration* to help developers to focus on the potential sources of architectural problems [158].

Verma et al. proposed an automated prioritization technique for classes associated with the *God Class* code smell, based on three criteria – the number of code smell instances, the type of code smell, and the number of changes in the class [159]. The methodology employs the Mamdani fuzzy inference system (MFIS) to automate the prioritization process. Notably, this is the first study to utilize MFIS for prioritizing god classes. The approach is designed to deliver unbiased, error-free, and efficient results.

Several studies proposed techniques to prioritize classes for refactoring based on the code smells and software metrics [160, 161, 162, 163]. For instance, Zhang et el. suggested refactoring suggestions based on the code smells having impact on software faults [164]. Chug et al. proposed a technique to determine a refactoring sequence for classes in advance with the help of heuristic search A* algorithm based on maintainability [165]. Tsantalis et al. proposed a technique for prioritizing refactoring opportunities based on past source code changes [166].

In another study, Alkharabsheh et al. proposed a prioritization technique specifically for the *god class* code smell, combining three criteria – historical information, design smell density, and developer context [167]. The technique was applied to two versions of 24 open-source systems and evaluated by professional developers from the same teams. Developers were also asked to identify factors they considered for ranking God Class smells. Using Spearman's correlation coefficient, the study found a weak overall correlation between the technique's rankings and those provided by evaluators, though better alignment was observed with more experienced developers.

In a systematic literature review, Verma et al. identified many factors influencing code smell prioritization, including cyclomatic complexity, module importance, maintenance overhead, and faults [103]. However, they observed that most studies did not utilize ranking formulas to prioritize code smells, despite this being a critical aspect of research in prioritization. Additionally, some researchers relied on existing ranking formulas without enhancing them with new parameters. Singh et al. concluded that prioritizing is important from a maintenance point of view and it significantly reduces maintenance efforts [168]. This highlights the need to develop improved prioritization methods by incorporating relevant software quality and maintainability metrics to effectively identify the most impactful set of code smells.

Code smells are usually recognized as patterns in the source code that reduce program comprehensibility and increase the likelihood of errors [30]. Moreover, program comprehensibility plays a significant role in software development and maintenance, as it is required 58% to 70% of the time in reading source code [12]. Griffith et al. proposed an automated tool that refactor code smells to improve the code comprehensibility [54]. In another study, they observed that refactoring requires understandability of the code to which it is applied [169].

3.4 Summary

The studies discussed in this chapter highlight that code smells affect various attributes of software quality and maintainability, such as coupling, cohesion, and fault-proneness. In addition, there is no consensus on what the important attributes are and how to measure their impact [170]. However, the specific impact of each individual type of code smell on these attributes remains unclear and requires further investigation to gain more precise insights. Furthermore, it is significant to prioritize code smells based on their specific impact because not all of the code smell types affect software quality to the same degree. Effective prioritization enables addressing the most harmful smells first, enhancing software quality and reducing maintenance costs over time.

CHAPTER 4

FILTERING CODE SMELLS AND SELECTING METRICS

Reusing source code containing code smells can induce significant amount of maintenance time and cost. A long list of code smells has been identified in the literature and developers are encouraged to avoid the smells from the very beginning while writing new code or reusing existing code. Again, remembering and refactoring a long list of smells are difficult specially for the new developers. Besides, two different types of software development environment – open-source and industry, might have an effect on the occurrences of code smells. Therefore, this chapter shows an empirical study on the occurrences of code smells in open-source and industrial systems which can provide insights about the most frequently occurring smells in each type of software development system. The insights can make

developers aware of the most frequent occurring smells, and researchers to focus on the improvement and innovation of automatic refactoring tools or techniques for the smells on priority basis. The results show that 5 out of 18 analyzed code smells have not occurred in any system, and 13 smells have occurred 24,487 times in total, where 61.55% in the open-source systems and 38.45% in the industrial systems. *Long Method, Complex Class, Long Parameter List* and *Large Class* have been seen as frequently occurring code smells. These findings conclude that all smells do not occur at the same frequency and some smells are very frequent. The study also concludes that industry and open-source environments do not have significant impact on the occurrences of code smells. Based on the frequency analysis in this chapter, the 13 code smells have been finally selected to be analyzed in the subsequent chapters to prioritize these smells. In the last part of this chapter, relationship between the selected internal and external software metrics which are used in smell prioritization is discussed.

4.1 Introduction

Developers usually introduce code smells due to less awareness in design and program comprehensibility [1]. This situation of code smell occurrences is correct for both categories of software development environment – industrial (also known as in-house) and open-source systems or projects. An industrial system is the one where a team of several software engineers develops and manages it within a same working environment [171]. As it is developed sitting on the same environment or house, it is also known as in-house system. On the other hand, an open-source system (OSS) is one where developers from various environment of different countries can contribute in the development of it [172, 173]. The source code of an OSS is publicly available, allowing other users or developers to view, modify, contribute and distribute it freely under specific licenses.

There are two mix opinions on the presence of code smells in open-source software systems and industrial software systems. One group believes that open-source software systems contain less number of code smells comparing to industrial developed systems. This is because of having no time and budget constraints, no release pressure and developers freedom of work. They get enough time to write the smell free code. Another group believes that existence of code smells is higher in open-source systems than industrial systems. The reason is that developers have to follow standards while writing code as imposed by the company. Besides, they have to refactor the code considering the long term evolution of the system. On the other hand, open-source developers are not bound to refactor the code as there is no client pressure on them for the long-term maintenance. Therefore, due to these different development environments, there may have different occurrences of code smells in those systems.

Nowadays, software development is performed by reusing source code with little modification if required to reduce development time and cost. Source code can be reused from open-source systems and previously developed industrial systems (if allowed). However, if developers are not aware of the presence of code smells in the code to be reused, such reuse might spread these smells across the systems, and so increase more maintenance time and cost. So, developers from both industrial and open-source environment should be enlightened with the statistics of the presence of the smells in the existing source code, specially the frequently occurring smells. For example, if a developer wishes to reuse code from open-source systems (or industrial) and knows the probable frequently occurring smells in those systems in advance, s/he can refactor those smells (if applicable) before reusing the code in his/her system. Such type of activity will make the system more maintainable in the long run. In addition, it is very difficult for developers to write code considering all possible code smells. Hence, a list of frequently occurring code smells might help them to focus and be careful about only those smells while developing systems.

As a result, it might be helpful to reduce the smells from the systems at the time of development. Therefore, it is important to find out which smells occur more frequently in which environments – industrial and/or open-source, to make awareness about those smells among the developers of both environments.

Most of the existing research focused on the effects of code smells on software maintainability using open-source software systems [174, 41, 19, 10]. In a study, Yamashita et al. [136] investigated inter-smell relations where they observed that patterns of inter-smell relations vary between open-source and industrial systems by analyzing only three systems – two open-source and one industrial systems. However, there is a noticeable lack of knowledge about the occurrences of code smell in industrial systems. Moreover, to the best of the author's knowledge, there exists no such empirical work regarding the comparative analysis of code smell occurrences between open-source and industrial systems. To mitigate this research gap, the study of this chapter presents a large-scale empirical study aimed at investigating the occurrence of code smells in both open-source and industrial systems and comparing the results.

In order to conduct the analysis, an empirical study has been carried out on 40 Java systems, where 25 of them are popular open-source systems and rest 15 are industrial systems. open-source systems have been selected based on the popularity, number of stars and contributors. Each industrial system has been developed following a standard Software Development Life Cycle (SDLC) model and is currently used by clients. A list of 18 code smells has been selected for the investigation. An existing tool named DÉCOR [82] has been used for smell detection, because the tool shows good accuracy and has been used in the earlier studies [175, 10, 19]. To the best of the author's knowledge, this is the largest study aimed at analysing the frequency of code smell occurrences in both open-source and industrial systems.

The results of the study show that 13 code smells have been found in both open-source and industrial systems. However, 5 code smells – *Base Class Knows Derived Class, Functional Decomposition, Message Chain, Swiss Army Knife*, and *Traditional Breaker* have not been seen in any systems. A total number of 24,487 occurrences of the 13 smells have been found where 38.45% are in the industrial systems and 61.55% are in the open-source systems. Among the 13 smells, *Long Method* have been seen the most frequent smells as its occurrences are 33.16% and 32.46% of the total occurrences in the industrial and open-source systems respectively. The next most prevalent smells are *Complex Class, Long Parameter List* and *Large Class*, as the occurrences of the first one are 32.38% and 31.09% of total occurrences of the smells in industrial and open-source systems respectively. The presence of the second one, that is, *Long Parameter List* is 16.85% of the total occurrences in the industrial systems and 9.91% in the open-source systems. The third one, that is, *Large Class* is 11.50% of the total occurrences in the industrial systems and 14.74% in the open-source systems. For every 41 methods in the industrial systems and 45 methods in the open-source systems, one method is found as a *Long Method* code smell. Besides, one of every 6 classes in the industrial systems and one of every 8 classes in the open-source systems have been found as a *Complex Class* code smell. In addition, one of every 17 classes in the industrial systems and one of every 16 classes in the open-source systems has been found as a *Large Class* smell. One method has been seen containing *Long Parameter List* smell out of every 80 and 149 methods in industrial and open-source systems, respectively. Other smells have not been seen so prevalent. One tailed t-test with 5% level of significance has shown that there is no difference between the occurrences of the code smells in the industrial and open-source systems except two smells – *Base Class Should Be Abstract* and *Refused Parent Bequest*. These two smells are found more in open-source systems than industrial systems as supported by the t-test. Based on the frequency analysis, 13 code smells are finally selected

for the prioritization in the subsequent chapters of the thesis.

4.2 Empirical Study Design

The goal of this study is to identify the frequently occurring code smells in both types of software development systems – open-source and industrial, and compare the results between them, with the purpose of having insightful information about the smell occurrences.

4.2.1 Formulating the Research Goal

The study aims at investigating the presence of different code smells in open-source and industrial systems by answering the following research questions:

RQ1 [Code smells in open-source and industrial systems]: ***What types of code smells do occur in open-source and industrial systems respectively?***

Developers can reuse source code from two sources – industrial systems and open-source systems. Usually, developers follow standard software development process to develop industrial systems. On the other hand, usually, development process of most of the open-source systems does not force developers to follow such rigid steps like industrial system development. Due to the different nature of the two categories of systems and their development process, it is necessary for developers to know the types of code smells occur in each category of the systems independently. The answer to the question will help them to know whether they need to deal differently for each of type of systems while reusing source code.

RQ2 [Frequency of code smells]: ***Which types of code smells do frequently occur in open-source and industrial systems***

Literature contains many code smells, but not all of the smells occur frequently and are important for software development. Answering the question can help researchers and practitioners to know the frequently occurring code smells in both open-source and industrial systems individually. The insights will motivate them to develop and improve refactoring tools and techniques for the smells on priority basis.

RQ3 [Differences between the occurrences of code smells in open-source and industrial systems]: ***Are there any differences between the occurrences of the code smells in industrial and open-source systems?***

The ideology of industrial development environment is to do business with the developed systems or earn money by delivering software meeting client requirements. On the contrary, open-source systems are developed as contributions to the user and developer community without any business intentions. In addition, the environment, organizational setup and operations also differ from one another. The question aims to find whether the contrast of motive and operational process between industrial and open-source development process has any impact on the occurrences of code smells.

4.2.2 Systems under Study

The context of the study consists of (i) software systems from both open-sources and industries and (ii) code smells. In order to conduct the empirical study and answer the research questions, 40 large scale Java systems are analyzed as the dataset: 25 well-known open-source systems and 15 industrial systems. Both categories of the systems are large based on their size and development time. The average development time span for the open-source systems is around 6.6 years, whereas for the industrial is 6 years. The industrial systems have been collected from six different software industries in four countries around the world. The

description of the dataset are shown in Table 4.1. It is noted that, source code of the industrial systems studied in this research cannot be disclosed, as these are confidential to the companies.

Table 4.1: Description of the dataset in the study

Category of Systems	Systems #	KLOC		NOM		NOC	
		Total	Average	Total	Average	Total	Average
open-source	25	3276	131	222151	8886	36228	1449
Industrial (In-house)	15	2093	140	127631	8509	18479	1232

The smell detection tool, DÉCOR [82] and three size-metric (Table 4.1) measurement tool, SourceMonitor[1] that produced the results have been provided to the corresponding industry personnel and they have given the results without disclosing the system information such as system, class and method name, system architecture, etc. The tools have been used to calculate the metrics for the open-source systems as well.

4.2.3 Detection of the Code Smells

To carry out the investigation, a list of 18 code smells has been selected as shown in Table 4.2 (first column). A smell detection tool, DÉCOR [82] has been selected to identify the occurrences of each code smell in the systems.

4.3 Result Analysis

To answer **RQ1**, the occurrences of code smells have been analyzed. The presence of code smells across the systems in the study along with frequencies has been studied to answer **RQ2**. The differences between the occurrences of the code smells in industrial and open-source systems have been reported with t-test results in the answer of **RQ3**.

[1] https://www.derpaul.net/SourceMonitor/

Table 4.2: Comparative result analysis between industrial and open-source systems

Code Smell (S_i)	Industrial system						Open Source system						p-value
	Total Smell Frequency (f_{S_i})	**Average Smell Frequency (Per System)**	*Spreadity* = **System# having** S_i / **Total System#**	$ISF_i(LOC)$ *Score* = LOC / (f_{S_i})	$ISF_i(NOM)$ *Score* = NOM / (f_{S_i})	$ISF_i(NOC)$ *Score* = NOC / (f_{S_i})	**Total Smell Frequency** (f_{S_i})	**Average Smell Frequency (Per System)**	*Spreadity* = **System# having** S_i / **Total System#**	$ISF_i(LOC)$ *Score* = LOC / (f_{S_i})	$ISF_i(NOM)$ *Score* = NOM / (f_{S_i})	$ISF_i(NOC)$ *Score* = NOC / (f_{S_i})	
Anti Singleton (AS)	86	5.73	0.67	24336	1484	215	213	8.52	0.80	15380	1043	170	0.114
Base Class Knows Derived Class (BCKDC)	0	0.00	0.00	NA	NA	NA	0	0.00	0.00	NA	NA	NA	NA
Base Class Should Be Abstract (BCSBA)	5	0.33	0.13	418578	25526	3696	59	2.36	0.52	55524	3765	614	**0.008**
Blob	82	5.47	0.47	25523	1556	225	72	2.88	0.72	45499	3085	503	0.460
Class Data Should Be Private (CDSBP)	238	15.87	0.93	8794	536	78	783	31.32	0.96	4184	284	46	0.382
Complex Class (CC)	3049	203.27	1.00	686	42	6	4685	187.40	1.00	699	47	8	0.160
Functional Decomposition (FD)	0	0.00	0.00	NA	NA	NA	0	0.00	0.00	NA	NA	NA	NA
Large Class (LC)	1083	72.20	1.00	1932	118	17	2222	88.88	1.00	1474	100	16	NA
Lazy Class (LzC)	141	9.40	0.60	14843	905	131	581	23.24	0.84	5638	382	62	0.222
Long Method (LM)	3122	208.13	1.00	670	41	6	4892	195.68	1.00	670	45	7	0.080
Long Parameter List (LPL)	1587	105.80	0.93	1319	80	12	1493	59.72	1.00	2194	149	24	0.286
Many Field Attributes But Not Complex (MFABNC)	5	0.33	0.20	418578	25526	3696	13	0.52	0.20	251992	17089	2787	0.211
Message Chains (MC)	0	0.00	0.00	NA	NA	NA	0	0.00	0	NA	NA	NA	NA
Refused Parent Bequest (RPB)	7	0.47	0.07	298985	18233	2640	26	1.04	0.36	125996	8544	1393	**0.016**
Spaghetti Code (SC)	7	0.47	0.13	298985	18233	2640	21	0.84	0.40	155995	10579	1725	0.110
Speculative Generality (SG)	4	0.27	0.13	523223	31908	4620	11	0.44	0.28	297808	20196	3293	0.420
Swiss Army Knife (SAN)	0	0.00	0.00	NA	NA	NA	0	0.00	0	NA	NA	NA	NA
Tradition Breaker (TB)	0	0.00	0.00	NA	NA	NA	0	0.00	0	NA	NA	NA	NA
Total	**9414**	**627.73**	**1.00**	**222**	**14**	**2**	**15071**	**538.25**	**1.00**	**217**	**15**	**2**	**0.137**

4.3.1 RQ1: Code Smells in open-source and Industrial Systems

According to the results of the study as shown in Table 4.2, out of the 18 code smells, 5 smells have not been found in both open-source and industrial systems. These smells are *Base Class Knows Derived Class, Functional Decomposition, Message Chain, Swiss Army Knife* and *Traditional Breaker*. Rest 13 smells have been seen present in both categories of the systems with different extents. The finding indicates that developers should be more careful on these smells, and researchers should give more emphasis on the improvement and invention of detection and refactoring techniques for the smells.

4.3.2 RQ2: Frequency of Code Smells

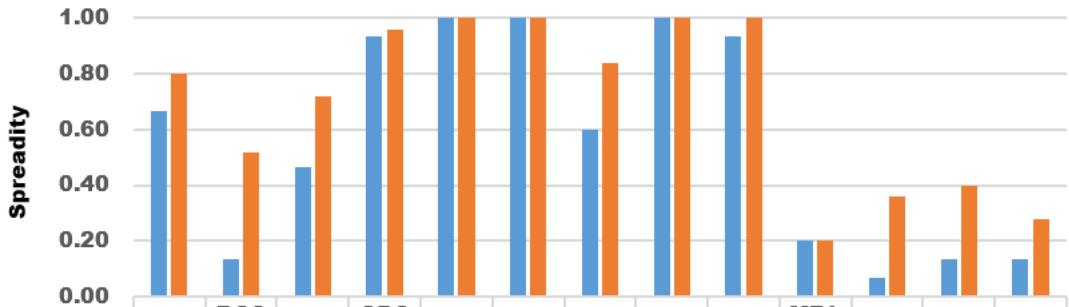
To get an idea about the extent of the presence of the smells, the answer of **RQ2** describes the frequency of the occurrences of these smells from different granular levels.

To understand what types of code smells frequently occur, the spreadity for each smell that appears at least once in any of the systems in the study. *Spreadity* is a metric that defines the existence of a smell in the systems with respect to the total number of systems. The metric is calculated using the following equation.

$$spreadity(x) = \frac{n(x)}{N} \tag{4.1}$$

Here, x = code smell, $n(x)$ = number of systems containing smell x and N = total number of systems. Higher value of the metric for a smell indicates higher number of systems contain the smells. The maximum value will be 1 if all the systems contain the smell and it will be zero if none of the systems contain it.

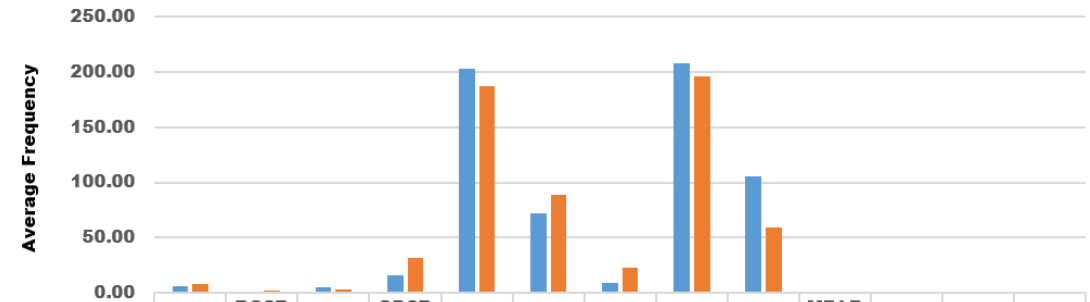
Figure 4.1 depicts the spreadity for the 13 smells, and rest 5 smells are inten-



	AS	BCSBA	Blob	CDSBP	CC	LC	LzC	LM	LPL	MFABNC	RPB	SC	SG
In-house	0.67	0.13	0.47	0.93	1.00	1.00	0.60	1.00	0.93	0.20	0.07	0.13	0.13
Open-source	0.80	0.52	0.72	0.96	1.00	1.00	0.84	1.00	1.00	0.20	0.36	0.40	0.28

Figure 4.1: Spreadity of each code smell

tionally omitted from the figure, since they have not occurred in any of the systems. *Complex Class, Large Class* and *Long Method* have been found in all the systems of both the open-source and industrial having spreadity 1. *Long Parameter List* has been found in all the open-source systems, but 93% of the industrial systems contain the smell. *Many Field Parent Attributes but Not Complex, Refused Parent Bequest, Spaghetti Code* and *Speculative Generality* seem less frequent in both types of systems. Another important observation is that the spreadity of the code smells are comparatively less in the industrial systems than open-source ones.



	AS	BCSBA	Blob	CDSBP	CC	LC	LzC	LM	LPL	MFABNC	RPB	SC	SG
In-house	5.73	0.33	5.47	15.87	203.27	72.20	9.40	208.13	105.80	0.33	0.47	0.47	0.27
Open-source	8.52	2.36	2.88	31.32	187.40	88.88	23.24	195.68	59.72	0.52	1.04	0.84	0.44

Figure 4.2: Average frequency of each code smell

The analysis is further extended to investigate the frequency of the occurrences of the smells in open-source and industrial systems. Figure 4.2 depicts the

average frequency of the occurrences of each smell per industry and open-source system. Like spreadity, *Long Method* tops with the frequency of 208.13 per industrial system and 195.68 per open-source system. *Complex Class* has followed the smell with the second highest appearances, 203.27 and 187.40 per industrial and open-source system respectively. *Large Class* is the third highest appearances having the average frequency of 72.20 and 88.88 in the industrial and open-source systems respectively. *Many Field Parent Attributes but Not Complex, Refused Parent Bequest, Spaghetti Code* and *Speculative Generality* are less frequent both in open-source and industrial systems which are also supported in Figure 4.2. Although the spreadity is comparatively higher in the open-source systems as shown in Figure 4.1, the average frequency seems higher in the industrial systems than the open-sources as shown in Figure 4.2.

The size of the systems under the study differs from one another. A large system is more likely have more code smells than smaller ones. So, the frequency of the occurrences of the smells have been further studied using a normalized metric – *inverse smell frequency,* $ISF_i(metric)$ score for a particular smell i in terms of a software metric (Equation 4.2). The use of this metric has been inspired by the Information Retrieval technique, *inverse document frequency, idf* calculation [176].

$$ISF_i(metric) = \frac{metric\ value}{smell\ count_i} \tag{4.2}$$

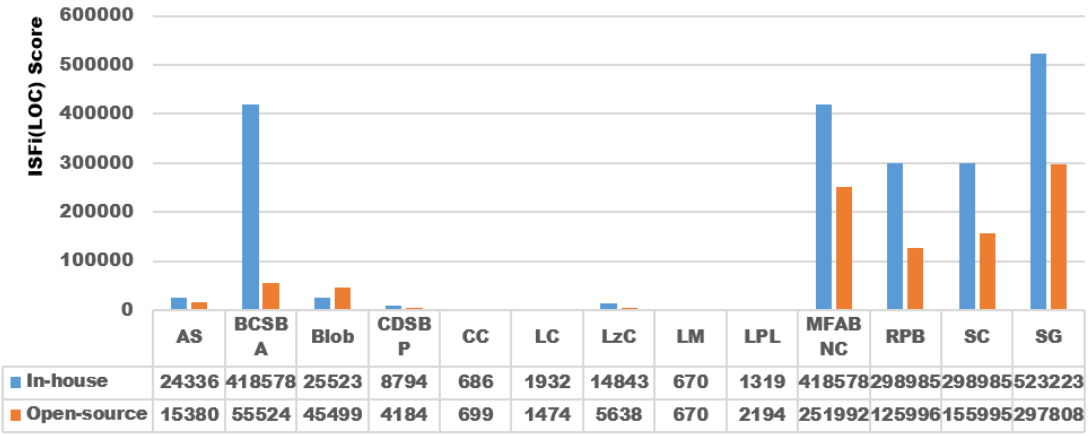
Here, *metric value* – a particular software metric value, such as - *LOC, NOM, NOC,* etc. *smell* $count_i$ – is for a particular code smell i, such as *Long Method* smell count, *Blob* smell count, etc.

From this metric, more three metrics have been derived for the study – inverse smell frequency for a particular smell i in terms of Line of Code ($ISF_i(LOC)$), Number of Methods ($ISF_i(NOM)$), and Number of Classes ($ISF_i(NOC)$), re-

spectively. $ISF_i(LOC)$ defines the average number of lines of code for which a smell occurs. $ISF_i(NOM)$ dictates the average number of methods for which a smell occurs. $ISF_i(NOC)$ asserts the average number of classes for which a smell is found. The lower the value of the metrics for a smell indicates the more frequent of the occurrence of the smell. The metrics can also assist to reveal after how many *LOC, NOM* and *NOC* a particular smell can occur. This insights can enlightened the developers with more clearer view about the frequency of each type of the code smells.

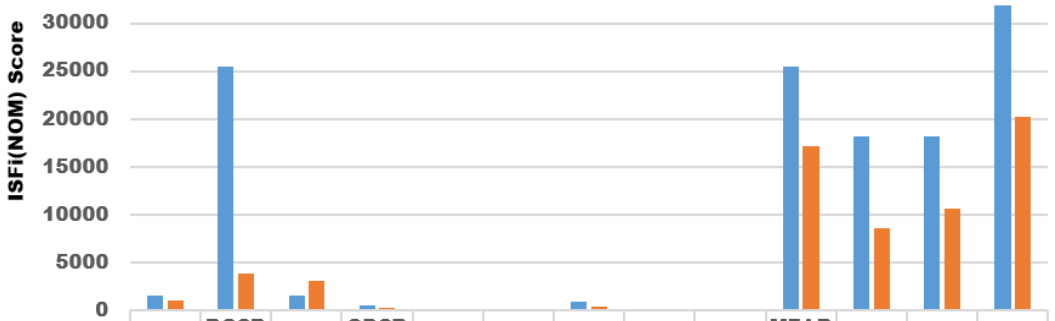
Figure 4.3, 4.4 and 4.5 illustrate the score of $ISF_i(LOC)$, $ISF_i(NOM)$ and $ISF_i(NOC)$ for all the smells under the study, except the 5 smells which have not been appeared in any system as described in the answer of **RQ1**. According to the figure, for every 670 line of code both in the industrial systems and in the open-source systems, a *Long Method* code smell occurs. Again, one *Long Method* is found for every 41 methods in the industrial systems and 45 methods in the open-source systems. In terms of number of classes, for every 6 classes in the industrial systems and 7 classes in the open-source systems, one *Long Method* smell is seen. The second most frequent code smell is *Complex Class* as for every 686 lines of code in the industrial systems and 699 lines of code in the open-source systems, the smell prevails. In fact, the smell is so frequent that for every 6 classes in the industrial systems and 8 classes in the open-source systems, one class is found as *Complex Class*. Third position is occupied by *Long Parameter List* as one method is found to have the smell in every 80 methods in the industrial systems and 149 methods in the open-source systems respectively. Another code smell *Large Class* is also seen frequent as it occurs after every 17 classes in the industrial systems and 16 classes in the open-source systems. *Lazy Class* and *Anti Singleton* are quite frequent as they occur 141 and 86 times in the industrial system respectively, and 581 and 213 times in the open-source systems respectively. Besides, one of every 131 classes in the industrial systems and 62 classes in the open-source systems has

been identified as *Lazy Class*. Other smells are not so prevalent as the observations of the study highlighted in Table 4.2.



	AS	BCSBA	Blob	CDSBP	CC	LC	LzC	LM	LPL	MFABNC	RPB	SC	SG
In-house	24336	418578	25523	8794	686	1932	14843	670	1319	418578	298985	298985	523223
Open-source	15380	55524	45499	4184	699	1474	5638	670	2194	251992	125996	155995	297808

Figure 4.3: $ISF_i(LOC)$ score of each code smell

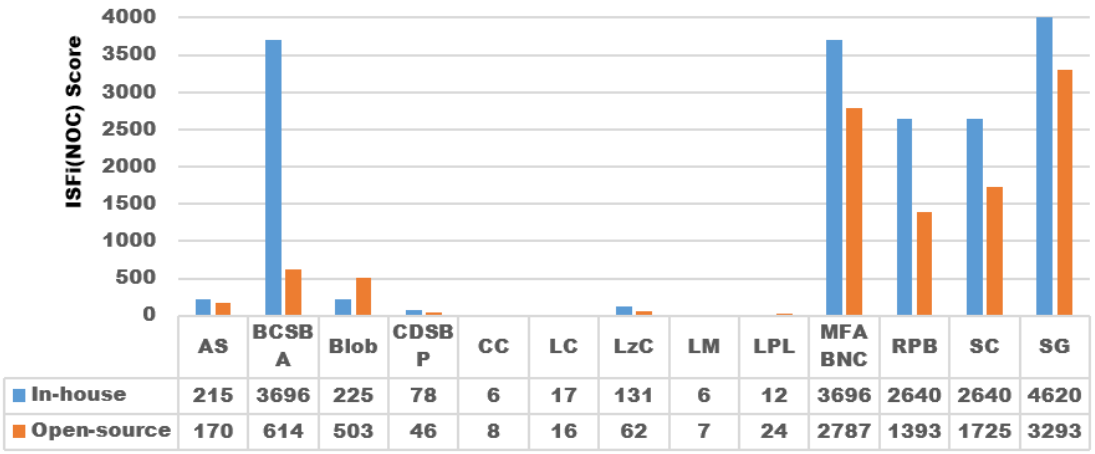


	AS	BCSBA	Blob	CDSBP	CC	LC	LzC	LM	LPL	MFABNC	RPB	SC	SG
In-house	1484	25526	1556	536	42	118	905	41	80	25526	18233	18233	31908
Open-source	1043	3765	3085	284	47	100	382	45	149	17089	8544	10579	20196

Figure 4.4: $ISF_i(NOM)$ score of each code smell

4.3.3 RQ3: Differences between the Occurrences of Code Smells in open-source and Industrial Systems

It has been seen in Figure 4.1 that the spreadity of the smells is more in the open-source systems than the industrial systems. Conversely, Figure 4.2 illustrates that the smells are more frequent in the industrial systems than the open-source



	AS	BCSBA	Blob	CDSBP	CC	LC	LzC	LM	LPL	MFABNC	RPB	SC	SG
In-house	215	3696	225	78	6	17	131	6	12	3696	2640	2640	4620
Open-source	170	614	503	46	8	16	62	7	24	2787	1393	1725	3293

Figure 4.5: $ISF_i(NOC)$ score of each code smell

systems. This research question aims to identify whether the occurrences of code smells differ between open-source and industrial systems.

A statistical hypothesis testing, one tailed t-test with 5% level of significance [177] has been conducted where the hypotheses are as following.

- *Null hypothesis (H_0): There is no difference between the occurrences of a particular code smell in open-source and industrial systems.*
- *Alternative hypothesis (H_1): The occurrences of a particular smell in open-source systems is greater than in industrial systems.*

The test has been carried out for each smell for which at least one existence has been found in any of the systems in the study. For a particular smell, frequency of the smell has been normalized by dividing the total occurrences of the smell in a system with the total number of line of code of the system. The normalized data have been divided into two groups: (i) one group contains the normalized system-wise frequencies for the open-source systems, and (ii) another group contains the normalized frequencies for the industrial systems. The results of the t-test has been shown in Table 4.2. The column p-value depicts the results for each of the smell under the study, where *NA* indicates the test has not been conducted for

the smell as it has not occurred in any of the systems.

According to the p-value, the null hypothesis (H_0) has been accepted for all the smells except *Base Class Should Be Abstract* and *Refused Parent Bequest*. This result indicates that there is no difference between the occurrences of the smells in open-source and industrial systems. The rejection of the null hypothesis (H_0) for the smells *Base Class Should Be Abstract* and *Refused Parent Bequest* states that these smells occur more frequently in open-source systems than industrial ones.

4.4 Relationship between Software and Maintainability Metrics

Apart from our primary research objective, this section aims to thoroughly investigate the relationships between the 25 software quality metrics and two vital maintainability metrics, namely change-proneness and fault-proneness. This analysis provides deeper insight into the role these metrics play in the context of this thesis, particularly in the prioritization of code smells.

For this investigation, we analyze the relationships among these metrics within the well-known open-source systems from the Apache and Eclipse ecosystems (Table 5.1) using the statistical correlation method – Spearman's Rank Correlation Coefficient [177]. We have used this correlation method because it is a non-parametric measure, making it suitable for our datasets that do not follow a normal distribution. To conduct the data analysis, we have used the correlation method for each of the software metrics with respect to change-proneness and fault-proneness of the analysed systems. For instance, we have taken the score of metric *Lines of Code (LOC)* of each system as the x variable and change-proneness (or fault-proneness) of the corresponding system as the y variable to measure the correlation coefficient between them. Thus, we have calculated the

relationships for all the metrics with change- and fault-proneness. The software metrics measurement process by the Understand tool [82] and maintainability metrics measurement process are detailed in Chapter 5 and Chapter 6 respectively.

A comprehensive presentation of these detailed results is provided in Table 4.3. The correlation coefficients (r_{cp}) and (r_{fp}) listed in the table indicate the strength of the relationship between each metric and the tendency of the system to undergo changes and faults respectively.

Table 4.3: Relationship of Internal Software Metrics with Change-proneness and Fault-proneness

Id No.	Software Metric	Description	Correlation with Change-proneness, r_{cp}	Correlation with Fault-proneness, r_{fp}
1	CountDeclMethod, NOM	Number of local methods	**0.50**	*0.35*
2	CountDeclMethodDefault, NDM	Number of local default methods	*0.47*	0.22
3	CountDeclMethodPrivate, NPriM	Number of local private methods	*0.46*	0.25
4	CountDeclMethodProtected, NProM	Number of local protected methods	*0.32*	0.15
5	CountDeclMethodPublic, NPM	Number of local public methods	**0.52**	*0.37*
6	CountStmt, NOS	Number of statements	*0.46*	*0.33*
7	CountStmtDecl, NDS	Number of declarative statements	*0.45*	*0.31*
8	CountStmtExe, NexS	Number of executable statements	*0.46*	*0.32*
9	SumCyclomatic, CC	Sum of cyclomatic complexity of all nested methods	*0.48*	*0.33*
10	LCOM	Lack of cohesion in methods	*0.49*	*0.33*
11	Depth of Inheritance Tree, DIT	Maximum depth of class in inheritance tree	*0.41*	*0.32*
12	IFANIN	Number of immediate base classes	**0.51**	*0.38*
13	Coupling Between Objects, CBO	Number of other classes to which a particular class is coupled	**0.51**	*0.36*
14	Number of Children, NOCh	Number of immediate subclasses	*0.32*	0.26
15	Response For a Class, RFC	Number of methods including inherited ones	*0.32*	*0.30*
16	Number of Instance Methods, NIM	Number of methods defined in a class that are only accessible through an object of that class	*0.49*	*0.32*
17	Number of Instance Variables, NIV	Number of variables defined in a class that are only accessible through an object of that class	*0.48*	0.29
18	Weighted Methods per Class, WMC	Sum of the complexities of all class methods	*0.46*	0.29
19	Classes, NOC	Number of Classes	*0.49*	*0.36*
20	Files, NOF	Number of Files	*0.38*	0.23
21	Lines, NL	Number of all lines	*0.47*	*0.32*
22	Lines Blank, BLOC	Number of blank lines of code	**0.50**	*0.33*
23	Lines of Code, LOC	Number of lines containing source code	*0.46*	*0.32*
24	Lines Comment, NCL	Number of comment lines of code	0.25	0.12
25	RatioCommentToCode, RCTC	Ratio of comment lines to code lines	*-0.47*	*-0.42*

[* Bold, italic and normal fonts indicate high, moderate and low (or no) correlation respectively.]

To interpret the correlation coefficient, we have followed the Cohen's guidelines [178]. Based on the correlation coefficients ($r = r_{cp}$ or r_{fp}), the software metrics are categorized into the following three impactful groups:

1. **High Impactful Metrics ($0.5 \leq r \leq 1$):** Five metrics related with change-proneness (r_{cp}) fall into this group as shown in Table 4.3. Interestingly, no

metrics related with fault-proneness (r_{fp}) have been found in this group.

2. **Moderate Impactful Metrics ($0.3 \leq r < 0.5$):** 19 and 17 metrics fall into this group related to change-proneness and fault-proneness respectively as shown in the table.

3. **Low Impactful Metrics ($0.0 \leq r < 0.3$):** Only one metric related to change-proneness falls into this group: *Lines Comment (Number of comment lines of code, NCL)*. On the other hand, eight metrics related to fault-proneness fall into this group.

Only one metric *RatioCommentToCode (RCTC)* has a moderate but negative correlation with both change-proneness and fault-proneness. That is, if comments increase in a system, change-proneness and fault-proneness decrease and vice-versa. It could be possible that comments help in understanding source code and thus resisting to code changes and faults.

In summary, the results revealed that most of the metrics have little to no correlation with fault-proneness. However, metrics related to inheritance, coupling, and comments showed a moderate to high correlation with change-proneness. These findings will assist developers to minimize the higher correlated software metrics to enhance maintainability in terms of change- and fault-proneness. Additionally, these insights can guide researchers in developing new approaches for predicting changes and faults by incorporating the metrics that have been shown to have stronger correlations.

4.5 Threats to Validity

The construct, external and reliability threats to validity [179] of this empirical study are discussed in this section.

1. Threats to construct validity

The potential threats are related to the accuracy of the tool used in this study to detect code smells from source systems. The inaccurate results might lead to a different phenomenon than the investigation of this study. However, to mitigate this threat, the tool that has been evaluated in previous studies for its good accuracy has been used in this study.

2. Threats to external validity

Three potential threats have been identified of this study while generalizing the findings of the study from the sample. First, Java systems have been used in this study, and there is a possibility that the results would be different for other object-oriented languages such as *C#*. Second, results cannot be generalized to other types of code smells. Finally, these results might not be extrapolated to more other open-source and industrial systems. Since it is not available to have industrial systems for this research domain, analyzing a lot of systems is difficult. However, 15 industrial systems have been collected from various industries of different countries instead of one industry or country and 25 open-source systems which might mitigate the threats to some extent.

3. Threats to reliability validity

The potential threats concerns to the replication of the analysis performed in this study using the industrial systems. Since these systems are confidential, information about them cannot be provided. However, the results can be reproduced by using the dataset for open-source systems using the tool mentioned in Section 4.2.

4.6 Summary

The study of this chapter attempts to identify the most frequent occurring code smells in both types of systems – open-source and industrial. The inclusion of the industrial systems along with the open-source ones makes the study more

interesting in a sense that most of the smell-occurrences are similar to both types of the systems. In addition, it is interesting that 5 out of 18 smells have not occurred in any system, whereas four smells, namely *Long Method, Complex Class, Long Parameter List* and *Large Class* have been the most frequent ones in both types of systems. Therefore, these findings will assist developers to reduce the smells while developing systems, and researchers to innovate the smell refactoring techniques from the viewpoint of frequency analysis. Finally, the list of 13 frequent code smells are selected to prioritize these smells in the subsequent chapters of the thesis.

CHAPTER 5

SOFTWARE METRIC BASED IMPACT ANALYSIS OF CODE SMELLS

Code smells are symptoms of poor design and implementation choices that decrease software quality and maintainability. Despite the effort devoted by the research community in detecting and refactoring code smells, the impact of code smells on software quality metrics such as size, complexity, coupling, etc. remains still unclear. To mitigate this research gap, in this chapter, we present an empirical investigation on the impact analysis of code smells based on the 25 software quality metrics. To the best of our knowledge, this is the largest empirical study about the impact analysis of code smells with respect to the number of software metrics.

Particularly for this study, we identify 13 code smells in 35 open-source software systems, and analyze – (1) the relationship between code smells and software metrics, (2) which code smells are highly impactful that affect the metrics, and (3) which impactful smells occur frequently in the systems. The results show the correlation based impact of the code smells with the software metrics. Three categories of impact for the code smells have been identified, namely – *High, Moderate* and *Low*, where *Long Method, Complex Class, Large Class* and *Long Parameter List* smells have high impact on the software metrics and their frequencies of occurrences are also high. On the contrary, *Many Field Attributes But Not Complex* smell has both low impact and frequency. We also observe that perceptions about the impact of code smells varies from developers to developers and they most cases refactor the smells based on their intuition. Our findings will help them to refactor the smells on objective-based instead of intuition-based, which will be more significant to improve the software quality. For example, refactoring the smells having high impact on *coupling between objects* metric can be an objective. Moreover, our findings will not only assist developers to prioritize refactoring activities, but also researchers to innovate refactoring and smell prioritization tools based on their impact.

5.1 Introduction

Code smells are the symptoms of poor design quality indicating that a software system is hard to maintain [1, 10]. To improve the quality of a system, developers identify and remove code smells by following a standard process which is known as refactoring [1]. It is expected that they should identify all the smells and refactor properly to make the system smooth for long term maintenance. However, due to the release pressure, budget and time constraint issues, they might not be able to refactor all the smells. Moreover, it is very difficult for them to write code con-

sidering all possible smells [2]. Usually, in this case, they apply refactoring on the smells which are well-known to them based on their intuition [140]. Since all the smells do not have the same effect on the decline of software quality, refactoring the low impactful and leaving the higher impactful smells might not help to ultimately achieve the optimum goal (that is, making software more maintainable in lesser time). So, a study is required to understand which smells have higher impact on the software quality that will help developers in their refactoring decisions.

Code smells have negative impact on the quality of a software system that affect its maintainability [9]. Software quality can be measured by software metrics such as size, complexity, coupling, etc. [13, 14]. So, code smells are important design factors that affect the quality metrics of a software system. Minimizing these metrics through removing important or impacted code smells is one of the goals of developers in the maintenance phase of the system. Therefore, it is necessary to identify the relationship between code smells and software metrics so that the metrics impacted by a particular smell can be recognized. It is very difficult for the developers to remove or work with all the code smells at a time from a developed system, as it is a huge time consuming task. It is also not wise to refactor only known or frequent code smells, as not all of the smells have the similar impact or importance to be removed. For example, some smells might have less impact on software quality, and also some smells occur with very low frequency, which developers can focus with less priority, as low frequency means less effect on maintainability [2]. So, it is essential to identify such code smells having higher impact on the software quality metrics to be refactored. It will help developers to refactor not only the known smells by definition but also the impactful ones which ultimately improve the design quality of the system.

In this chapter, we define *impactful code smell* as one that affects a software system, particularly in terms of software quality metrics such as size, complexity,

coupling, etc. In other words, a code smell is considered highly impactful if it strongly correlates with these metrics, and less impactful if the correlation is weak.

Software has two types of quality attributes or metrics – internal and external software quality metrics [7]. In the literature, many works have covered the impact of code smells on the external software metrics such as maintainability [9]. For instance, Khoma et al. [19] and Palomba et al. [10] investigated the impact of code smells on two maintainability metrics, namely code change-proneness and fault-proneness. Abbes et al. [37] showed that multiple code smells affecting the same source code have negative impact on program comprehensibility. However, very few researches have been found about their impact on internal software metrics such as size, complexity, coupling, etc. [9]. Also, there is a noticeable lack of knowledge about the identification of impactful code smells based on the internal software metrics that helps prioritizing these smells in the maintenance phase. To mitigate this research gap, we present an empirical study aimed at investigating the relationship between code smells and internal software quality metrics to identify their impact.

In order to conduct the analysis, in this chapter, an empirical study has been carried out on 35 open-source Java systems. From these systems, we have identified 13 types of code smell and 25 internal software metrics to further measure the relationship between them and identify the impactful code smells. For this research, we refer 'code smell type' as 'code smell', that is, we use 'code smell type' and 'code smell' interchangeably. In the context of this chapter, the 'impact' of a code smell refers to the relationship based on correlation between the code smell and the software metrics. To the best of author's knowledge, this is the largest empirical study investigating the impact or relationship between the presence of code smells and software metrics, particularly in terms of the number of metrics examined. Additionally, we quantify the impact of each code smell based on cor-

relation analysis, which aids in the prioritization of code smells. Moreover, we analyze which impactful code smells are most frequent, meaning those that occur most frequently in the systems. The comparative analysis between the impact of code smells and their frequency of occurrences makes the study different from the existing works. Moreover, this comprehensive approach allows us to better understand which code smells have the greatest effect on software quality and how often they appear, providing valuable insights for improving software maintenance and development practices.

In summary, this chapter of the thesis makes the following contributions:

(i) **Quantifying the impact of code smells:** We establish the relationship between code smells and software metrics to quantify the impact of these smells.

(ii) **Impact levels of code smells:** We categorize code smells into three impact levels – *High, Moderate,* and *Low* to help prioritize them. This prioritization can be used to optimize software metrics and enhance software quality.

(iii) **Comparative analysis of frequency and impact:** We conduct a comparative analysis between the frequency and impact of code smells, demonstrating that developers should focus on those smells that have both high impact and high frequency.

(iv) **Synthesized smelly dataset:** We provide a synthesized dataset containing 74,253 smelly files where each of the 13 smelly files are stored separately for the 35 systems. This dataset can be used by the research community for further studies and to build upon our research.

Structure of the chapter: Section 5.2 describes the design of the empirical study and Section 5.3 discusses the synthesized dataset generation process. Section 5.4 presents and discusses the results of the study, while the threats to their validity

are discussed in Section 5.5. Finally, Section 5.6 concludes the chapter describing the key findings of the work.

5.2 Empirical Study Design

The goal of the study is to assess the impact of 13 code smells based on 25 internal software metrics (also known as internal quality attributes) by analysing 35 open source software systems with the purpose of helping in maintenance activities. It is noted that, this is the largest empirical study with respect to the number of internal quality metrics of software systems. More specifically, the study aims at identifying the impactful set of code smells that affects the metrics. It is worth remarking that the term "impactful", when associated to a code smell, refers to the correlation of the code smell with the software metrics. Indeed, some smells might be highly correlated with the metrics but rarely occurred in the systems or vice versa. Therefore, frequency analysis of code smells having their impacts is also another goal of the study. In this section, the research questions of the study (Sub-section 5.2.1), selected software systems under the study (Sub-section 5.2.2), selected code smells (Sub-section 5.2.3) and software metrics (Sub-section 5.2.4) are discussed.

5.2.1 Formulating Research Goal

The study aims at identifying the impactful set of code smells based on their relationship with software metrics by answering the following three research questions:

RQ1: What is the relationship between code smells and software metrics?

The research question investigates which software metrics are highly correlated with which code smells by analyzing the correlation between them in the software systems. More specifically, a set of clusters of software metrics for each code smell

based on the relationship is to be identified. The outcome will assist software developers to emphasize on the smells to be refactored in order to optimize the highly correlated metrics.

RQ2: What is the impactful set of code smells?

Code smells which have higher correlation with software metrics are highly impactful. The list of these highly impactful smells should be given higher priority during maintenance activities, specially when performing refactorings to improve software quality. The target of this research question is to categorize the code smells into three impact levels – (*High, Moderate* and *Low*) based on the relationship analysis in RQ1.

RQ3: Which frequent code smells are highly impactful?

Not all of the highly impactful code smells might be important based on their frequency of occurrences in the systems. It could be possible that some high impactful code smells could occur with very low frequency, and hence these could be avoided in the maintenance phase. Similarly, low impactful smells with high frequency could also be avoided. On the other hand, high impactful smells having higher frequency are important, since these actually affect the maintenance activities. So, only impact analysis of the smells is not enough but frequency analysis is also important. he objective of this research question is to compare the impact and frequency of various code smells. By doing so, we aim to identify the code smells that occur frequently and have a significant impact on software quality. These are the code smells that should be prioritized during the maintenance phase to ensure the most effective improvements in software quality.

5.2.2 Systems Under Study

The context of the study consists of (i) software systems, (ii) code smells, and (iii) software metrics. In order to conduct the empirical study and answer the research questions, we have analyzed 35 Java systems belonging to two major

ecosystems: Apache[1] and Eclipse[2]. The systems of these ecosystems are well-known in code smell research domain [10, 21]. Table 5.1 summarizes the analyzed systems, the latest stable version of the systems up to this study, their size in terms of number of classes (NOCs), number of methods (NOMs) and lines of codes (LOCs), and history of the projects as starting to ending year including their duration. These systems have been selected because these are in different sizes (from 8,069 to 1,762,363 LOCs), larger enough having an average NOCs of 4,925; NOMs of 38,450 and LOCs of 460,522. In addition, the selected systems have an average duration of 12.5 years and cover different application domains such as *ArgoUML* - a UML-based system modeling tool, *Eclipse* - an IDE, *Lucene* - a search manager, *Xerces* - an XML parser, etc.

5.2.3 Code Smells Selection

To carry out the investigation, a list of 13 code smells has been selected. These 13 code smells including a short description are listed in Table 5.2. An existing tool named DÉCOR [82] has been used for the detection of these smells from the selected systems, because the tool shows good accuracy and has been widely used in the previous studies [175, 10, 19].

We choose these code smells for the study because – (1) these are representative of various object-oriented design issues such as inheritance, polymorphism, encapsulation, etc., (2) the frequencies of these smells are great in numbers to conduct the empirical study, and (3) their frequencies are almost similar between open-source and industrial systems [2].

[1] https://www.apache.org/.
[2] https://www.eclipse.org/org/.

Table 5.1: Software systems involved in the study

Id No.	**Project**	**Version**	**#Classes**	**#Methods**	**LOCs**	**Project History (Start to end year, Duration in years)**
1	Activemq	5.17.0	4,747	42,349	421,979	2005 to 2022, 17
2	Ant	1.9.9	1,595	13,768	138,391	2000 to 2013, 13
3	Ant-ivy	2.5.0	814	7,540	76,678	2005 to 2019, 14
4	ArgoUML	0.35.1	2,096	17,955	177,402	1998 to 2014, 16
5	Cassandra	4.0.5	8,101	76,237	1,483,583	2009 to 2021, 12
6	Cayenne	4.1	6,656	41,339	485,781	2007 to 2020, 13
7	CDT	10.6.2	4,254	26,883	267,165	2002 to 2022, 20
8	CXF	3.5.3	11,109	103,490	1,120,148	2008 to 2021, 13
9	DeepLearning4j	1.0.0	9,630	59,725	698,622	2019 to 2022, 3
10	Drill	1.10.0	6,414	60,203	533,478	2012 to 2017, 5
11	Eclipse Core	R4.9	4,391	32,977	305,762	2001 to 2018, 17
12	ElasticSearch	8.3.3	23,187	154,096	2,097,667	2010 to 2022, 12
13	Freemind-mmx	1.0.0	744	6,804	63,933	2011 to 2013, 2
14	Hadoop	3.3.4	17,393	143,785	1,762,363	2009 to 2022, 13
15	HBase	3.0.0	9,457	71,619	823,856	2007 to 2021, 14
16	Hive	3.1.3	14,466	107,711	1,257,002	2008 to 2018, 10
17	Hsqldb	2.7.0	952	13,874	208,333	2007 to 2022, 15
18	Incubator-livy	0.7.0	201	811	8,069	2015 to 2020, 5
19	Jackrabbit	2.9.0	3,209	29,053	332,826	2004 to 2014, 10
20	JBoss-modules	2.0.3	301	1,956	20,586	2010 to 2022, 12
21	Jena	4.5.0	7,077	66,138	576,575	2012 to 2022, 10
22	JFreeChart	1.5.3	955	11,336	136,786	2007 to 2021, 14
23	JHotDraw	9.0	768	7,630	80,440	2000 to 2020, 20
24	Karaf	4.4.1	1,760	10,332	126,431	2007 to 2022, 15
25	Lucene	9.3.0	7,501	52,968	827,867	2001 to 2022, 21
26	Mahout	14.1	1,551	9,552	111,437	2008 to 2019, 11
27	Nutch	2.4	519	3,136	38,268	2005 to 2019, 14
28	Opennlp	1.9.4	1,075	5,303	75,724	2010 to 2021, 11
29	Pig	0.9.2	2,222	15,433	231,371	2007 to 2012, 5
30	Poi	5.2.2	4,291	40,032	413,657	2002 to 2022, 20
31	Qpid	0.32	3,374	29,189	327,821	2006 to 2014, 8
32	Struts	6.0.0	2,378	18,640	186,595	2006 to 2022, 16
33	Tomcat	10.0.22	3,989	33,433	355,081	2006 to 2022, 16
34	Wicket	7.2.0	4,402	20,678	215,286	2004 to 2016, 12
35	Xerces	2.9.1	800	9,809	131,326	1999 to 2006, 7
Total		-	**172,379**	**1,345,784**	**16,118,289**	-
Average		-	**4,925**	**38,450**	**460,522**	**12.5**

5.2.4 Software Metrics Selection

Software quality is an important factor in measuring how much a software system is maintainable. In ISO/IEC 9126 set by the International Organization for Standardization [8] standard, software quality is defined by a series of attributes or metrics that describe and assess the quality of a software system. These quality

Table 5.2: Code smells involved in the study

Id No.	Code Smell	Description
1	Anti Singleton (AS)	A class that provides mutable class variables, which consequently could be used as global variables.
2	Base Class Should Be Abstract (BCSBA)	An inheritance tree contains roots that are not abstract - only the leaves should be concrete.
3	Blob	A class that is too large and not cohesive enough, that monopolises most of the processing of a system, takes most of the decisions, and is associated to data classes.
4	Class Data Should Be Private (CDSBP)	A class that exposes its fields, thus violating the principle of encapsulation or data hiding.
5	Complex Class (CC)	A class that contains (at least) one large and complex method, in terms of cyclomatic complexity and LOCs.
6	Large Class (LC)	A class that is large in size in terms of LOCs.
7	Lazy Class (LzC)	A class that has few fields and methods (with little complexity), and that does not do too much in the system.
8	Long Method (LM)	A method that is very large in size and complex.
9	Long Parameter List (LPL)	A method that contains a long list of parameters.
10	Many Field Attributes But Not Complex (MFABNC)	A class that is not complex but has too many fields those are public.
11	Refused Parent Bequest (RPB)	A class that inherits functionalities from its parent but never uses these, thus violating polymorphism.
12	Spaghetti Code (SC)	A class without structure that declares long methods without parameters, thus prevents polymorphism.
13	Speculative Generality (SG)	A class that is defined as abstract but it has very few children, which does not make use of its methods.

attributes are divided into two categories [7]: *internal quality attributes* and *external quality attributes*. Internal quality attributes are those that can be directly measured from the software's source code, such as lines of code. External quality attributes, on the other hand, are measured indirectly from the software, such as maintainability. However, the study in this chapter focuses solely on internal quality attributes, leaving external quality attributes beyond its scope.

In this research, we examine 25 internal software quality attributes, which we refer to as internal software quality metrics or simply software metrics for ease of reading. These metrics are computed by a well-known tool designed for research purposes, called Understand [180, 14]. The metrics according to the name provided by the tool, their abbreviations, and short description are shown in Table 5.3. For clarity and readability, these abbreviations are used in the results section.

Table 5.3: Software metrics involved in the study

Id No.	Software Metric	Description
1	CountDeclMethod, NOM	Number of local methods
2	CountDeclMethodDefault, NDM	Number of local default methods
3	CountDeclMethodPrivate, NPriM	Number of local private methods
4	CountDeclMethodProtected, NProM	Number of local protected methods
5	CountDeclMethodPublic, NPM	Number of local public methods
6	CountStmt, NOS	Number of statements
7	CountStmtDecl, NDS	Number of declarative statements
8	CountStmtExe, NExS	Number of executable statements
9	SumCyclomatic, CC	Sum of cyclomatic complexity of all nested methods
10	LCOM	Lack of cohesion in methods
11	Depth of Inheritance Tree, DIT	Maximum depth of a class in inheritance tree
12	IFANIN	Number of immediate base classes
13	Coupling Between Objects, CBO	Number of other classes to which a particular class is coupled
14	Number of Children, NOCh	Number of immediate subclasses
15	Response For a Class, RFC	Number of methods including inherited ones.
16	Number of Instance Methods, NIM	Number of methods defined in a class that are only accessable through an object of that class
17	Number of Instance Variables, NIV	Number of variables defined in a class that are only accessable through an object of that class
18	Weighted Methods per Class, WMC	Sum of the complexities of all class methods
19	Classes, NOC	Number of Classes
20	Files, NOF	Number of Files
21	Lines, NL	Number of all lines
22	Lines Blank, BLOC	Number of blank lines of code
23	Lines of Code, LOC	Number of lines containing source code
24	Lines Comment, NCL	Number of comment lines of code
25	RatioCommentToCode, RCTC	Ratio of comment lines to code lines

These metrics have been selected for the study in this chapter for several reasons discussed below.

1. **Comprehensive Coverage of Design Factors:** The chosen metrics encompass six critical design factors of a software system: complexity, coupling, cohesion, abstraction, encapsulation, and documentation [14]. These factors are essential in providing a well-rounded assessment of software quality.

2. **Measurement of Software Maintainability:** These design factors are capable of measuring various aspects of software maintainability, as outlined in the study by Zhang et al. [14]. This ensures that our analysis is robust and relevant to the practical needs of software maintenance.

3. **Representation of Different Levels:** The metrics are divided into two

groups based on their levels. Metrics Id No. 1 to 19 represent class-level (or file-level) metrics, while metrics 20 to 25 represent project-level metrics. For the purpose of our analysis, we have aggregated the class-level metrics to create project-level metrics, allowing us to maintain a consistent level of analysis across the dataset.

5.3 Synthesized Dataset Generation

In order to analyze the impact of each code smell on the software metrics, we need to create a dataset from the selected systems from which we can collect information about the smells and metrics. This dataset consists of a set of directories containing smelly files for each of the 13 smells separately for each of the 35 projects. In this study, we define 'smelly file' as – *a file is called a smelly file for a code smell type, if it contains that type of code smells.* For example, if a file contains a *Long Method* code smell, then this file is called a 'long method smelly' file. It is noted that we consider any level of smells such as class, method or block level as a smelly file. Because – most of the code smells are introduced at the time of file creation [126] and smelly files are more likely to undergo changes [40]. Therefore, developers write and manage code at the file level, making it a base unit of analysis. Moreover, when developers examine code smells, they tend to look for issues that affect the entire file, rather than just specific methods or blocks of code.

In particular, the dataset contains the following three levels of directories. The generic structure of the dataset directory is shown in Figure 5.1, where PDi (i = *1, 2, 3, ..., N*) is the i^{th} project directory, SDj (j = *1, 2, 3, ..., M*) is the j^{th} smelly directory, and in this study $N = 35$, $M = 13$.

(i) **Level 0 - Master Directory:** It is the name of the root directory that contains project directories. In our study, it has 35 (N) directories for the

corresponding 35 project names.

(ii) **Level 1 - Project Directory (PD):** For each of the 35 project directories in the master directory, it contains 13 (M) directories for the corresponding 13 code smells. These directories are used to store smelly files of the corresponding code smell directory of the corresponding project.

(iii) **Level 2 - Smelly Directory (SD):** It is the last level of the directories. For each of the 13 smelly directories, it contains the corresponding smelly files for a particular project.

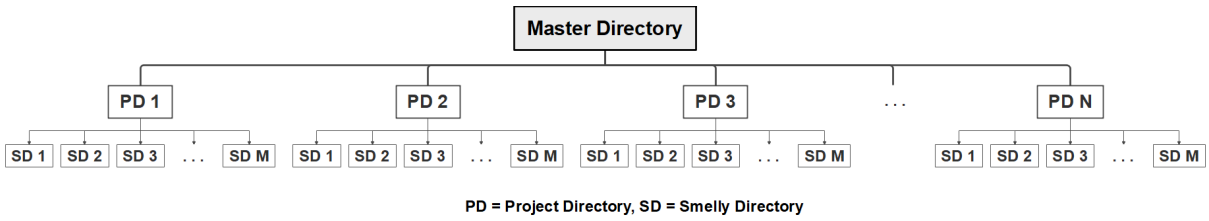


Figure 5.1: Generic structure of the dataset

For example, *Activemq* project has 13 smelly directories in which *Anti Singleton* smelly files reside in the *AntiSingleton* directory, *Long Method* smelly files are in the *LongMethod* directory, and so on. In this way, we have $35 \times 13 = 455$ smelly directories in total for all the 35 systems. We refer this whole dataset as a *'synthesized dataset for code smells'*. The dataset is given here[3] so that more research and analysis about code smells can be performed.

The dataset generation is a two-step process – (1) code smell detection and (2) smelly directory generation. These steps are discussed in the following subsections.

5.3.1 Code Smell Detection

To create the synthesized dataset, all the 13 code smells are needed to be detected for each of the 35 systems. A popular smell detection tool DÉCOR [82] has been

[3] https://bitbucket.org/masudshrabon/impactstudy/

used for this purpose. This tool detects the code smells for the 35 systems as well as provides a list for each of the smells which is unstructured, since it contains many information including the corresponding smelly file names. We then organize the unstructured smelly lists to make these structured by filtering only the smelly file names for each smell type. Therefore, we can use these structured files to generate the smelly directories having the smelly files.

We have developed a python script that makes this structuring process of filtering easier as manual work requires a huge time. However, we also manually validated the output of the structured files to avoid whether there exist any missing smelly file names. It is noted that, all the structured files containing the smell names as well as scripts related to this dataset generation process are provided[3].

5.3.2 Smelly Dataset Generation

According to the structure given in Figure 5.1, we have first created 35 empty directories (folders) for the corresponding 35 systems in the *master directory*. These are the *project directory* of our dataset structure. Then, in each of the project directories, we have created another 13 empty *smelly directories*.

Now, we need to fill up the empty smelly directories with the corresponding smelly files. Since there exists a lot of smelly files for each of the systems, it will be a hectic and huge time consuming task to manually find and copy smelly files. For this, we have written a bash script to find and copy the smelly files from the structured files of smell names from a project automatically. For each smell of each project, we have copied the actual smelly *Java* files into the corresponding smelly directory from the corresponding project using this script. However, we have also manually validated the files in the smelly directories and smelly file names to avoid inconsistencies. For example - we have resolved missing files manually; we have resolved duplicate file names of a project as *same_filename*, *same_filename2*, and

so on. It takes 25 minutes on an average for one system for finding and copying the 13 types of smelly files automatically by the script, excluding the manual intervention. That is, for each smell of each systems, it takes less than two (*25 divided by 13*) minutes. So, for the 35 systems, it requires around $25 \times 35 = 875$ minutes or 14 hours 35 minutes. In this way, we have generated the synthesized dataset separately and stored project-wise smelly files.

The hard disk space or size for the synthesized smelly dataset is 1.35 GB which contains only *Java* files. Whereas the actual space for the raw 35 systems is 2.08 GB, that includes both *Java* and *non-Java* files. It is noted that, the synthesized dataset contains 74,253 number of smelly files in total. We have also observed that there is a small difference (only - 0.73 GB) between the sizes of these two types of dataset. This is because that multiple smelly directories of a project contain several same files because these same files contain multiple types of code smells. For example, file *A* of a project contains two smells - long method and blob, and hence this file, *A* will be in the both smelly directories of that project. Therefore, in the smelly dataset, the number of total files of a project increases more than the unique number of files.

We use the dataset for analysing impact of code smells on various software metrics in this study. Researchers can use this dataset to investigate more about the code smells individually, their impact on different software aspects such as code comprehensibility, maintainability, etc. Refactoring and smell detection tools can also be validated for each particular type of smells using the dataset.

5.4 Data Analysis and Results

In this chapter, a large scale empirical study has been conducted on 35 open-source software systems to understand the impact and frequency of 13 code smells. Besides, relationships between the prevalence of code smells and software metrics

have been investigated. A comparative analysis between the impact and frequency of the smells has also been conducted. The detailed result analysis is explained as follows.

5.4.1 Relationship between Code Smells and Software Metrics [RQ1]

The aim of the research question (RQ1) is to identify the relationship between each code smell and each software metric using correlation analysis. Based on the correlation strength, software metrics are clustered for each individual smell. Moreover, based on the overall relation with respect to all code smells, the metrics are categorized into three impact levels – *High, Moderate* and *Low* impact. This overall relationship is referred to as 'impact level' of the metrics in this chapter. Also, in this research, we define 'smell frequency' as the number of occurrences of a code smell in a system.

5.4.1.1 Data Analysis Process for RQ1

1. Smell frequency calculation

To conduct the study, we have calculated the smell frequency for each of 13 code smells from the 35 software systems using the *DÉCOR* tool. That is, in this step, we have 13 types of smell frequency for each of the systems, or in other words, we have 35 smell frequency for each code smell considering all the systems.

2. Software metric calculation

We have used the *Understand* tool in the synthesized dataset of the smelly files to calculate the 25 software metrics for each code smell (from the corresponding smelly folder) of each of the 35 systems. Here, we have calculated these metrics at the file level for each smelly folder of each system in the dataset. Then, we have summed up the metric scores of the files of each smelly folder individually to get the smell level metric scores of each system. Thus, we will be able to get

the 25 metric scores for each of the 13 smells in the 35 systems. For example, we have calculated the *LOC* scores for each file in the *LargeClass* smelly folder of a system, then summed up the scores of all the files in this smelly folder for that system, and thus got the *LOC* score for the *LargeClass* smell of the system. In this way, we have 35 scores of each software metric for each code smell considering all the 35 software systems.

3. Correlation coefficient measurement

Up to this step, we have collected data on 13 different types of code smell, including their frequencies and 25 software metric scores for each of the 35 software systems in our study. For each code smell, we have calculated the Spearman's Rank Correlation Coefficient [177] between the smell frequencies and the software metric scores for all 35 systems. We have chosen the Spearman's correlation because it is a non-parametric measure, making it suitable for our datasets that do not follow a normal distribution.

For example, we have examined the frequency of the *Large Class* code smell in each system as our x variable. Correspondingly, we have taken the *Lines of Code (LOC)* metric scores of the classes identified as *Large Class* in those systems as our y variable. We have then calculated the correlation coefficient to determine the relationship between these two variables. It is noted that, to calculate the correlations, we have used a high-level data analysis language called R [181].

This method has been repeatedly used for all the 25 software metrics for each code smell. Therefore, we have computed 25 correlation coefficients for each of the 13 code smells, resulting in a total of $13 \times 25 = 325$ correlation pairs. This comprehensive analysis of the correlation helps us understand which metrics are significantly impacted by each code smell and which ones remain unaffected. This insight is significant for identifying potential areas for code quality improvement and prioritizing refactoring efforts.

To understand the correlation strength of a variety of degrees, the correlation scores have been grouped into 5 categories according to the approach by Schober et al. [182]. According to them, the range 0.00 to 0.10 represents negligible correlation, 0.10 to 0.39 defines weak correlation, 0.40 to 0.69 indicates moderate correlation, 0.70 to 0.89 denotes strong correlation, and 0.90 to 1.00 is interpreted as very strong correlation. The categories have been further extended to five more categories to understand the negative correlation between occurrences of code smells and software metrics. The ranges are 0.00 to -0.10 indicating negatively weak correlation, -0.10 to -0.39 representing weak, -0.40 to -0.69 moderate, -0.69 to -0.90 strong, and -0.90 to -1.00 very strong correlation but negative. These categories are significant to understand the relationship level of each smell with the metrics.

5.4.1.2 Result of RQ1

1. Correlation categories

We have analyzed the correlation scores to identify the relationship between code smells and software metrics. Table 5.4, 5.5, and 5.6 show the relationship of each code smell with the 25 software metrics based on correlation strength. Based on the correlation categories discussed earlier, we have grouped the software metrics into these categories for each code smell as shown in the tables.

For example, *Anti Singleton* smell is 'very strongly' correlated with 22 software metrics (category 0.90 to 1.00), 'strongly' correlated with only 2 metrics (category 0.70 to 0.89), and 'negative weakly' correlated with only 1 metric (-0.10 to -0.39). We refer to these correlation categories for a code smell as a cluster of metrics for that particular smell. So the above example is the cluster of the metrics for the *Anti Singleton* smell. Similarly, Table 5.4, 5.5, 5.6 show the clusters (each smelly column in the tables) for the 13 smells, which represent the strength between each smell and each metric. However, the detail about the raw correlation scores is

Table 5.4: Cluster of software metrics based on *AS, BCSBA, Blob* and *CDSBP*

Correlation	**Anti Singleton**	**Base Class Should Be Abstract**	**Blob**	**Class Data Should Be Private**
0.90 to 1.00	NOM, NPriM, NProM, NPM, NSM, NDM, NExM, CC, LCOM, DIT, IFANIN, CBO, RFC, NIM, NIV, WMC, NOC, NOF, NL, BLOC, LOC, NCL		NOM, NPM, NSM, NDM, CC, LCOM, DIT, IFANIN, CBO, RFC, NIM, NIV, WMC, NOC, NOF, NL, BLOC, LOC	NOM, NProM, NPM, NSM, NDM, DIT, IFANIN, NOC, NOF, Lines Blank
0.70 to 0.89	NDM, NOCh	NOM, NPriM, NPM, NSM, NDM, NExM, CC, LCOM, DIT, IFANIN, CBO, NOCh, RFC, NIM, NIV, WMC, NOC, NOF, NL, BLOC, LOC, NCL	NPriM, NProM, NExM, NCL	NDM, NPriM, NExM, CC, LCOM, CBO, NOCh, RFC, NIM, NIV, WMC, NL, LOC, NCL
0.40 to 0.69		NDM, NProM, RCTC	NDM, NOCh	
0.10 to 0.39			RCTC	
-0.10 to -0.39	RCTC			RCTC

Table 5.5: Cluster of software metrics based on *CC, LC, LzC, LM* and *LPL*

Correlation	**Complex Class**	**Large Class**	**Lazy Class**	**Long Method**	**Long Parameter List**
0.90 to 1.00	NOM, NPriM, NProM, NPM, NSM, NDM, NExM, CC, LCOM, DIT, IFANIN, CBO, NOCh, RFC, NIM, NIV, WMC, NOC, NOF, NL, BLOC, LOC	NOM, NPriM, NProM, NPM, NSM, NDM, NExM, CC, LCOM, CBO, RFC, NIM, NIV, WMC, NOC, NOF, NL, BLOC, LOC		NOM, NPriM, NProM, NPM, NSM, NDM, NExM, CC, LCOM, DIT, IFANIN, CBO, NOCh, RFC, NIM, NIV, WMC, NOC, NOF, NL, BLOC, LOC	NOM, NPriM, NProM, NPM, NSM, NDM, CC, LCOM, DIT, IFANIN, CBO, NOCh, RFC, NIM, NIV, WMC, NOC, NOF, NL, BLOC, LOC
0.70 to 0.89	NDM, NCL	NDM, DIT, IFANIN, NOCh, NCL		NDM, NCL	NDM, NExM, NCL
0.40 to 0.69			NOM, NDM, NPriM, NProM, NPM, NSM, NDM, NExM, CC, DIT, IFANIN, CBO, NOCh, RFC, NIM, NIV, WMC, NOC, NOF, NL, BLOC, LOC, NCL		
0.10 to 0.39			LCOM		
-0.10 to -0.39	RCTC	RCTC	RCTC	RCTC	RCTC

given in Table A.1 in the appendix section.

However, the tool *DÉCOR* detects code smells that relies on the software metrics using a set of rules called rule cards[4] [82]. Therefore, we have provided a

[4]Link of rule cards:
https://github.com/ptidejteam/ptidej-Ptidej/tree/master/SAD%20Rules%20Creator/src/main/resources,

Table 5.6: Cluster of software metrics based on *MFABNC, RPB, SC* and *SG*

Correlation	Many Field Attributes But Not Complex	Refused Parent Bequest	Spaghetti Code	Speculative Generality
0.90 to 1.00	NOM, NSM, NDM, DIT, IFANIN, NOCh, RFC, NIM,WMC, NOC, NOF, NL, BLOC, LOC, NCL	NOM, NPM, NSM, NDM, NExM, CC, LCOM, DIT, IFANIN, NOCh, RFC, NIM, NIV, WMC, NOC,NOF, NL, BLOC, LOC, NCL	NOM, NPM, NSM, NDM, NExM, CC, LCOM, DIT, IFANIN, RFC, NIM, WMC, NOC, NOF, NL, BLOC, LOC, NCL	NOM, NPM,NSM, NDM, LCOM, DIT, IFANIN, RFC, NIM, NIV, WMC, NOC, NOF, NL, BLOC, LOC, NCL
0.70 to 0.89	NPriM, NPM, NExM, CC, LCOM, NIV, RCTC	NPriM, NProM, CBO	NDM, NPriM, NProM, CBO, NIV	NPriM, NProM, NExM, CC, CBO, NOCh, RCTC
0.40 to 0.69	CBO	NDM, RCTC	NOCh, RCTC	NOMDefault
0.10 to 0.39	NDM, NProM			

list of the metrics used by the tool for each code smell as shown in Table 5.7. The list is useful to understand the correlation of the code smells with the software metrics which are not affected by the smell-detection metrics of the tool.

Table 5.7: Software metrics used in *DÉCOR* for each code smell

Code Smell	**Metrics used in DÉCOR**
AS	Global Variable
BCSBA	Depth of Inheritance Tree (DIT), Number of Children (NOCh)
Blob	Number of Class Attributes (NAD), Number of Class Methods (NMD), Lack of Cohesion in Methods (LCOM5)
CDSBP	Public Field
CC	McCabe Cyclomatic Complexity
LC	Number of Class Attributes (NAD), Number of Class Methods (NMD), Lack of Cohesion in Methods (LCOM5)
LzC	Weighted Methods per Class (WMC), Number of Class Attributes (NAD), Number of Class Methods (NMD)
LM	Method LOC
LPL	Number of Parameters (NOParam)
MFABNC	Weighted Methods per Class (WMC), Number of Class Attributes (NAD)
RPB	Inherited from its Parents (IR)
SC	Method LOC, Number of Parameters (NOParam), Depth of Inheritance Tree (DIT)
SG	Number of Children (NOCh)

Last Accessed on 14 June, 2024.

2. Discussion and examples

For example, Table 5.5 shows the clustered metrics with the level of correlation for the code smell *Long Method* (column 5). From the table, it can be seen that out of 25 metrics, only one metric namely *RatioToCommentCode, (RCTC)* is negatively correlated with the frequency of the smell although the correlation is weak. By further analyzing the systems, we have observed that, this is because developers might write comments to make short methods understandable. *CountDeclMethodDeafult, (NDM)* and *LinesOfComment, (NCL)* are strongly correlated with the smell, whereas other metrics 22 metrics have very strong correlation with the smell. This could be because by definition *Long Methods* are large in size and so are strongly correlated with the size related metrics such as *NOM, NOF, NSM,* etc. Moreover, the smell is also strongly correlated with other significant metrics such as *LCOM, DIT, CBO,* etc. We have got this type of observation for other smells too. These observations make the study significant in a sense that the study not only shows the high relationship with the smell-definition based metrics, but also provides other important metrics which developers should focus to improve software quality. *Long Method* has also been found in RQ2 (Sub-section 5.4.2) as one of the high impactful code smells in our study.

For *Base Class Should be Abstract*, no negatively correlated metrics have been seen. In addition, no metric under the study has been found which has very strong correlation value for the occurrences of the smell. This indicates that this smell has lower impact on the software metrics than other smells. For existence of *Blob*, we have noticed variety of correlations for different metrics, for example, *NDM* and *NOCh* are moderately correlated, and *NPrim, NProM, NExM* and *NCL* are highly correlated with the presence of the smell. Other metrics have very strong correlation but only *RCTC* has very weak impact for the occurrences of the smell. This represents the moderate impact on the systems. We found very strong and high relationship for the metrics with the smell *Class Data Should be Private*

except *RCTC* which contributes negatively. For *Complex Class, Large Class, Long Method* and *Long Parameter List*, we have observed that almost the same set of metrics has very strong correlation values for having the smell. However, we have not found any metric having strong correlation for the presence of the smell *Lazy Class*. It is the only code smell in our study where the metrics moderately reflect their influences for the presence of the smell. From our observation, by definition, the smell contributes less to systems, and hence no strong relationship is found from the smell with the software metrics. Quite the same list of metrics can be seen for the existences of the smells - *Many Field Attributes But Not Complex, Refused Parent Bequest, Spaghetti Code* and *Speculative Generality*. It is interesting that, *RCTC* has negative correlation with most of the smells indicating that comments might help in reduction of the smells.

3. Impactful software metrics identification

To identify the impactful software metrics, we have further investigated the degree of correlation for all the metrics with the frequencies of code smells. The correlation scores for each of the smells with respect to the metrics are squared individually, which is formally known as coefficient of determination (r^2). The reason of squaring the correlation scores is that it allows us to consider both the positive and negative impacts of the metrics. Next, we have calculated the average r^2 (*Impact Score*) scores for each metric across all 13 code smells, assuming equal importance of each code smell. Because identifying impacts or weights is the purpose of this research and assigning weights to the smells is the future scope of our research. This calculation has been performed using the following formula, Equation 5.1:

$$Impact\ Score = \frac{\sum_{i=1}^{N} r_i^2}{N} \tag{5.1}$$

where r_i is the correlation score between the metric and each code smell, and N is the total number of code smells (in this case, $N = 13$).

In this study, this average r^2, referred to as the 'impact score' or simply 'impact,' represents the overall effect of a particular metric across all code smells. In this way, the impact scores for all the 25 software metrics have been measured.

Afterward, we categorized the metrics into three levels of impact: *High, Moderate,* and *Low*. This categorization is based on statistical quartiles, as described by Gupta [177]. We used the first quartile (Q1), the second quartile (Q2), and the third quartile (Q3) to determine these levels of impact. Specifically, we defined the impact levels as follows:

1. *High Impact*: Metrics with r^2 values greater than or equal to Q3, representing the top 25% of the r^2 scores.

2. *Low Impact*: Metrics with r^2 values less than or equal to Q1, representing the bottom 25% of the r^2 scores.

3. *Moderate Impact*: Metrics with r^2 values between Q1 and Q3, representing the middle 50% of the r^2 scores. We have combined Q2 and Q3 [183] due to the relatively low number of scores in these quartiles and the minimal difference between the scores in these ranges.

In this approach, the *High* impact level encompasses the metrics with the highest r^2 values, indicating a strong correlation and significant impact. The *Low* impact level includes the metrics with the lowest r^2 values, indicating a weak correlation and minimal impact. The *Moderate* impact level includes the metrics with intermediate r^2 values, indicating a moderate level of impact. We have combined Q2 and Q3 [183] due to the relatively low number of scores in these quartiles and the minimal difference between the scores in these ranges. This adjustment ensures a more balanced and meaningful categorization. This categorization provides valuable insights into which software metrics are most affected by code smells, helping to identify the key areas for potential improvements in software quality and maintainability.

Table 5.8: Impact of software metric based on code smells

Software Metric	**Impact Score**	**Impact Level**
NOF	0.93	**High**
NOC	0.88	
DIT	0.86	
NDS	0.85	
IFANIN	0.85	
NOS	0.83	
BLOC	0.82	
NOM	0.82	
LCOM	0.82	
NL	0.82	
LOC	0.82	
WMC	0.81	**Moderate**
NIM	0.80	
NPM	0.80	
RFC	0.79	
NExS	0.78	
CC	0.77	
NIV	0.76	**Low**
NCL	0.76	
NPriM	0.69	
CBO	0.68	
NProM	0.64	
NOCh	0.64	
NDM	0.46	
RCTC	0.21	

Table 5.8 presents a comprehensive list of metrics along with their corresponding average r^2 scores (referred to as impact scores) and their impact levels. The table reveals that 11 out of the 25 metrics are highly correlated with the occurrence of code smells. Additionally, 6 metrics exhibit a moderate impact level, while the remaining 8 metrics are considered to have a low impact. The metric *Number of Files (NOF)* has the highest r^2 value of 0.093, making it the most impactful metric. This indicates that an increase in the number of source files leads to more source code being added by developers, thereby raising the likelihood of introducing code smells. This trend is also evident in other size-related metrics with high impact, such as *Number of Classes (NOC)*, *Lines of Code (LOC)*, *Number*

of Methods (NOM), etc. Consequently, developers should prioritize these metrics to enhance software quality. Moreover, two inheritance-related metrics, *Depth of Inheritance Tree (DIT)* and *Number of Immediate Fan-in (IFANIN)*, also show a high impact. Developers should pay attention to these metrics as well. Conversely, the metric for comments (*Ratio of Comment to Code, RCTC*) has the lowest impact on the system. This is because comments serve as documentation to make the source code more understandable, rather than contributing to code complexity or other detrimental attributes. Metrics related to code complexity, such as *Weighted Methods per Class (WMC)* and *Cyclomatic Complexity (CC)*, have a moderate impact on the presence of code smells. Generally, an increase in the values of highly impactful metrics is associated with a higher likelihood of introducing code smells like *Complex Class*, *Long Method*, *Refused Parent Class Bequest*, etc.

The difference in impact scores between the adjacent levels seems very small. To determine whether these differences are statistically significant, we have conducted hypothesis testing using the statistical T-test [177]. Here are the hypotheses for our tests:

- *Null hypothesis (H_0): There is no significant difference in impact scores between adjacent impact levels (e.g., High vs. Moderate, Moderate vs. Low).*
- *Alternative hypothesis (H_1): There is difference in impact scores between adjacent impact levels (e.g., High vs. Moderate, Moderate vs. Low).*

We have carried out the one-tailed T-test (r^2 is a positive value) to compare the impact scores between the following pairs of impact levels: High vs. Moderate and Moderate vs. Low. Results of the hypothesis testing is shown below:

- *High vs. Moderate Impact Levels:* p-value = 0.001 (which is less than the significance level of 0.05)

- *Moderate vs. Low Impact Levels:* p-value = 0.002 (which is also less than the significance level of 0.05)

In both cases, the p-values are well below the 0.05 threshold, where we consider 5% level of significance. Therefore, we reject the null hypothesis (H_0), indicating that the differences in impact scores between the adjacent levels are statistically significant. The results show that there is a significant difference in impact scores between the *High* and *Moderate* levels, as well as between the *Moderate* and *Low* levels. These findings imply that the impact scores at different levels are indeed distinct and meaningful, providing a valid basis for categorizing the software metrics into *High, Moderate,* and *Low* impact levels. Consequently, this analysis confirms that the impact scores serve as a discriminating factor, effectively distinguishing between the different levels of impact.

These findings provide valuable insights for developers, enabling them to optimize the source code based on the impact of various metrics while maintaining the system. By focusing on the most influential metrics, developers can improve software quality and reduce the occurrence of code smells.

4. Implication of the result

Overall, the results show that many of the smells are highly correlated with most of the metrics indicating that code smells are harmful for the quality of a software system with respect to the internal software metrics. This means that as the frequencies of the smells increase, these metrics' scores also increase, and the overall quality of the system decreases. So, our results indicate that developers should be careful about the smells which are highly correlated with the metrics.

The correlation-categories assist software developers to identify the impactful metrics associated with a particular code smell. It does not only show high relations of a code smell with the smell-definition based metrics, but also provide other significant and higher relevant metrics associated with the smell. As a result,

this contribution help the developers to refactor code smells with objective-based rather than traditional intuition and general code improvement based refactoring approach. For instance, while refactoring a code smell, they will be able to learn about which metrics will get optimized. Similarly, if their objective is to optimize one or more certain significant metrics according to a system necessity, they will be able to get insight about certain impactful and relevant code smells which they should focus. For example, if a developer wants to optimize the metric *Coupling Between Objects (CBO)* from a system, s/he might focus on the smells such as *Anti Singleton, Blob, Complex Class, Large Class, Long Method, and Long Parameter List*, since these have high relationship with the *CBO* metric. In other words, if a developer wants to optimize a metric, say *CBO*, the developer should focus on the *Long Method* smell though by definition the metric is not related with the smell. Similarly, if s/he refactors a code smell, s/he will be aware of which metrics are getting optimized.

Summary for RQ1. Many of the code smells such as *Anti Singleton, Blob, Complex Class, Large Class, Long Method* and *Long Parameter List* are strongly correlated with most of the software metrics, specially related to the size (e.g., number of files, methods, classes, etc.). Our finding is significant in sense that, code smells are not only correlated with the metrics based on their definitions, but also related with other significant quality metrics such as *DIT, IFANIN, LCOM,* etc. Therefroe, developers should focus on both types of the metrics while improving system quality. Also, different software metrics have different impact levels on code smells. On the contrary, *comments* have negative impact on most of the code smells although many think *comment* itself as a smell.

5.4.2 Impactful Set of Code Smells [RQ2]

The aim of the research question (RQ2) is to identify the overall impact of each code smell based on the correlation scores with the software metrics and categorize

these smells into three levels of impact – *(i) High, (ii) Moderate* and *(iii) Low*. The RQ2 is different from RQ1 in a sense that, the goal of RQ2 is to measure and quantify the overall impact of the 13 code smells based on the correlation after combining the 25 software metrics, whereas RQ1 is based on direct correlation between each smell and each metric individually.

5.4.2.1 Data Analysis Process for RQ2

In this study, we have measured the impact of a code smell using the coefficient of determination (r^2), which is simply the square of the correlation, r [177] to incorporate both the negative and positive correlation. Firstly, to measure the impact of a code smell, we have calculated the square of each correlation score between the code smell and each of the 25 software metrics. Then we have calculated the average r^2 score for the code smell using the same Equation 5.1, where in this case r is the correlation score between the smell and each software metric, and N is the total number of software metrics ($N = 25$). This average r^2 is considered as 'impact score' or simply 'impact' of the code smell in this study. In this way, the impact scores for all the 13 code smells have been measured. We have further categorized the code smells into three levels of impact – *High, Moderate* and *Low*. Similar to the sub-section 5.4.1.2, the categories are based on the statistical quartiles [177]: quartile 1 (Q1), quartile 2 (Q2) and quartile 3 (Q3), where impact is *High if* $r^2 \geq Q3$*; Low if* $r^2 \leq Q1$*; and Moderate if* $Q3 > r^2 > Q1$. We have combined Q2 and Q3 [183] due to the relatively low number of scores in these quartiles and the minimal difference between the scores in these ranges.

5.4.2.2 Result of RQ2

1. Impactful code smells identification

The result based on the impact score of code smells is shown in Table 5.9. According to the table, firstly, it is found that five code smells are high impactful

and those are *Long Method, Anti Singleton, Complex Class, Large Class* and *Long Parameter List*. Secondly, four smells have moderate impact – *Refused Parent Bequest, Spaghetti Code, Speculative Generality* and *Blob*. Finally, remaining four smells have low impact – *Class Data Should Be Private, Many Field Attributes But Not Complex, Base Class Should Be Abstract* and *Lazy Class*.

Table 5.9: Impact of code smell based on software metrics

Code Smell	Impact Score	Impact Level
Long Method	0.86	**High**
Anti Singleton	0.85	
Complex Class	0.85	
Large Class	0.83	
Long Parameter List	0.83	
Refused Parent Bequest	0.82	**Moderate**
Spaghetti Code	0.82	
Speculative Generality	0.80	
Blob	0.79	
Class Data Should Be Private	0.77	**Low**
Many Field Attributes But Not Complex	0.76	
Base Class Should Be Abstract	0.59	
Lazy Class	0.26	

This high impactful set of smells has 42.93% impact of the total impact. The percentage has been calculated as the percentage of total impact of a category (e.g., high) divided by the total impact of all categories. Secondly, moderate impactful set of smells has 32.84% of the total impact. Finally, low impactful set of smells has 24.23% impact. Moreover, the results provide a prioritized list of code smells based on the impact score in descending order as shown in Table 5.9.

The difference in impact scores between the adjacent levels seems very small. To determine whether these differences are statistically significant, we have conducted hypothesis testing similar to the Sub-section 5.4.1.2. Here are the hypotheses for the tests:

- *Null hypothesis (H_0): There is no significant difference in impact scores between adjacent impact levels (e.g., High vs. Moderate, Moderate vs. Low).*

- *Alternative hypothesis (H_1): There is difference in impact scores between adjacent impact levels (e.g., High vs. Moderate, Moderate vs. Low).*

Results of the hypothesis testing using the one-tailed T-test is shown below:

- *High vs. Moderate Impact Levels:* p-value = 0.004 (which is less than the significance level of 0.1)
- *Moderate vs. Low Impact Levels:* p-value = 0.06 (which is less than the significance level of 0.1)

In both cases, the p-values are below the 0.1 threshold, where we consider 10% level of significance. Therefore, we reject the null hypothesis (H_0), indicating that the differences in impact scores between the adjacent levels are statistically significant. These findings indicate that the impact scores at different levels are distinct and meaningful, providing a valid basis for categorizing the code smells into *High, Moderate,* and *Low* impact levels. Therefore, this analysis confirms that the impact scores serve as a discriminating factor, effectively distinguishing between the different levels of impact.

2. Implication of the result

Our categorization of code smells by their impact provides valuable insights for both experienced and novice developers, encouraging them to reassess the significance of different smells and prioritize their efforts to enhance software quality. This categorization supports a shift from an intuition-based to an objective-based refactoring approach during maintenance activities, leading to significant improvements in overall system quality.

By understanding which code smells have the greatest impact, developers can make more informed decisions about where to focus their refactoring efforts. This ensures that the most harmful issues are addressed first, resulting in a more efficient and effective maintenance process. Additionally, our approach helps devel-

opers identify specific metrics that are improved by addressing certain code smells, further guiding their refactoring strategies.

Overall, our work empowers developers to systematically and objectively enhance software quality, moving beyond ad-hoc fixes to a more strategic and impactful approach to code maintenance. Moreover, it helps researchers to develop automated refactoring tools based on the smell impact.

Summary for RQ2. Mostly size related code smells such as *Long Method, Large Class,* etc. are highly impactful based on the correlation between the smells and software metrics. Structure or inheritance related smells such as *Refused Parent Bequest, Spaghetti Code,* etc. have modertae impact. Other smells have low impact on software quality. The general findings will help software practitioners to think more deeply about their impact on software systems.

5.4.3 Impact versus Frequency of Code Smells [RQ3]

Frequencies of various code smells are not uniform for a system [2], and hence all smells do not have the same importance. Although some code smells are highly impactful, their frequencies of occurrences in the systems might not be very high. The low frequency of a smell indicates that the number of the smell's occurrences is very few. Moreover, some smells might have higher frequencies but their impact might be very low. Therefore, these types of smells might not be very important for developers to focus on the maintenance phase. On the other hand, highly impactful code smells having higher frequencies might be the targeted smells which should be given higher priority in the maintenance. So, only impact analysis might not be enough for developers while performing prioritization and refactoring of the smells. These cases lead us to analyze the relationship between the frequency and impact of each code smell.

5.4.3.1 Data Analysis Process for RQ3

We have conducted a comprehensive analysis to measure the average smell frequency by examining the frequency of occurrences of a particular code smell across various software systems. The term 'average smell frequency' refers to the average number of times a particular code smell is found in each software system. To achieve this, first, we have identified the number of occurrences of the code smell in each of the 35 different software systems. After gathering this data, we have proceeded to calculate the average smell frequency for each code smell using the following Equation 5.2:

$$\textit{Average Smell Frequency, } ASF = \frac{\sum_{i=1}^{N} F_i}{N} \tag{5.2}$$

where F_i represents the frequency of the code smell in each individual system, and N represents the total number of systems, which in our study is 35.

By applying this formula, we have been able to calculate the average smell frequency for each of the 13 different code smells under consideration. Furthermore, similar to the methodology described in Sub-section 5.4.2.1, we have categorized the code smells into the three levels of frequency – *High, Moderate* and *Low*. This categorization is based on statistical quartiles, as described by Gupta [177]. We have used the first quartile (Q1), the second quartile (Q2), and the third quartile (Q3) to determine these levels of frequency. Specifically, we defined the frequency levels as follows:

- *High Frequency*: Metrics with ASF values greater than or equal to Q3, representing the top 25% of the ASF scores.
- *Low Frequency*: Metrics with ASF values less than or equal to Q1, representing the bottom 25% of the ASF scores.

- *Moderate Frequency*: Metrics with ASF values between Q1 and Q3, representing the middle 50% of the ASF scores.

This detailed approach ensures a precise and meaningful analysis of code smell frequencies across the software systems, allowing us to better understand the prevalence and distribution of the code smells.

5.4.3.2 Result of RQ3

1. Frequency level of code smells

The result based on the average frequency of code smells is shown in Table 5.10. According to the table, firstly, it is found that four code smells have high frequency – *Long Method, Complex Class, Long Parameter List* and *Large Class*. Secondly, the five moderate frequent smells are – *Class Data Should Be Private, Anti Singleton, Lazy Class, Blob* and *Base Class Should Be Abstract*. Finally, the four low frequent smells are – *Refused Parent Bequest, Spaghetti Code, Many Field Attributes But Not Complex* and *Speculative Generality*. The frequency category will help developers in understanding about which smells occur more or less.

Table 5.10: Average frequency level of code smell

Code Smell	**Average Smell Frequency (Per Project)**	**Frequency Level**
Long Method	733.06	**High**
Complex Class	565.06	
Long Parameter List	245.77	
Large Class	231.60	
Class Data Should Be Private	110.60	**Moderate**
Anti Singleton	86.34	
Lazy Class	79.03	
Blob	20.91	
Base Class Should Be Abstract	8.74	
Refused Parent Bequest	7.91	**Low**
Spaghetti Code	7.17	
Many Field Attributes But Not Complex	4.57	
Speculative Generality	3.14	

2. Relationship between frequency and impact of code smells

After identifying the frequency levels of the code smells, it is necessary to analyze whether frequency differs from impact or not. Therefore, we have compared our analysis between the smell frequency and impact for each smell. The comparative result of the analysis is summarized in the following Table 5.11. According to the table, firstly, it is observed that four high frequent smells, namely *Long Method, Complex Class, Large class* and *Long Parameter List* have high impact on software quality. So these smells should be carefully monitored while developing and maintaining a system. Only *Blob* and *Many Field Attributes But Not Complex* smells have both moderate and low impact and frequency level respectively. Secondly, we have observed several exceptions in the relationships. For instance, it has been seen that *Anti Singleton* smell has a moderate frequency level though its impact level is high. In addition, three smells – *RPB, SC* and *SG* have low frequency with moderate impact, whereas three other smells – *CDSBP, BCSBA* and *LzC* have moderate frequency with low impact.

Table 5.11: Impact versus frequency of code smell

		Frequency level		
		High	**Moderate**	**Low**
Impact	**High**	Long Method, Complex Class, Large Class, Long Parameter List	Anti Singleton	
	Moderate		Blob	Refused Parent Bequest, Spaghetti Code, Speculative Generality
	Low		Class Data Should Be Private, Base Class Should Be Abstract, LazyClass	Many Field Attributes But Not Complex

The exceptions discussed above have concluded that it is not always true that high, moderate and low frequency of code smells indicate high, moderate and low impact respectively.That is, not all of the smell frequency levels have the similar kind of effect on the correlation based impact. To make it clear, the correlations

show how the software metrics of a smell type will increase (or decrease) when the smell frequency increases (or decrease). On the other hand, frequency just tells their frequency of occurrences in the systems.

3. Implication of the result

This comparative analysis between the impact and frequency of code smells adds new contribution in the research field by providing meaningful insights to the developers and researchers. The result suggests them to focus on the high impactful smells having higher frequency level while developing a software and innovating refactoring tools. On the other hand, they can focus less on the lower impactful smells having any frequency levels as their impact can be negligible. Moreover, they can avoid low impactful smells with low frequency level to save development and maintenance time.

Summary for RQ3. One can view that high, moderate and low frequency of code smells mean high, moderate and low impact on software metrics respectively. These cases are not always hold for all the smell types. For example, *RPB, SC* and *SG* have moderate impact thpugh their frequency level is low, whereas *AS* has high impact with moderate frequency level.

5.5 Threats To Validity

The construct, external, internal and reliability threats to validity [179] of this empirical study are discussed in this section.

1. Threats to construct validity

The threats concern the relationship between theory and observation due to measurement errors in our study. The potential threats are related to the accuracy of the tools used in this study to detect code smells from software systems. The results of the tool might lead to a different phenomenon for different tools than our investigation. However, we have used the tool *DÉCOR* which has been evaluated

in previous studies as having good accuracy, and hence it might mitigate the threats to some extent.

2. Threats to external validity

Regarding the generalization of our findings, we limited our study based on only Java systems, due to limitations of the infrastructure we used (e.g., for a large number of code smells, detection tools only work on Java systems). Moreover, the choice of projects from Apache and Eclipse code-bases is also a threat to validity. However, these projects are well-known in this research domain and one can easily use them for replication and comparison purposes. In addition, the impact of code smells is based on the 25 internal software metrics. However, this is the largest study in terms of a number of software metrics (25), and considered code smell types (13) – on the impact analysis of code smells on software metrics. Moreover, we focused on open-source systems only, and we cannot speculate about how the results would be different for industrial systems. Since the frequency of code smells between open-source and industrial systems observed by Rahman et al. [2], these results can be generalized to some extent.

3. Threats to internal validity

The threats concern factors that could influence our observations. We are aware that we cannot claim a direct cause-effect relationship between the occurrence of code smells and software metrics. In particular, our observations may be influenced by the different factors related to development phases (e.g., experience of developers, workload, etc.). We focused that some smells are highly associated with some of the metrics, while some are not.

4. Threats to reliability validity

The potential threats concern to the possibility of replicating the study. However, we have provided all the necessary details including the synthesized dataset to replicate our findings (see Section 5.3).

5.6 Discussion

An empirical study on 35 Java open-source systems has been conducted with the purpose of understanding the impact of code smells on the systems and their relation with internal software metrics. The results highlighted the following findings:

1. **Relationship between code smell and software metric:** Most of the code smells have high correlation with size related metrics such as number of files, classes, methods, statements, etc. This means that when the occurrence of the smells increases, these metric scores also increase, and hence it decreases maintainability. Moreover, a code smell (e.g., *Long Method*) is not only associated with the metrics (e.g., *size*) by which it is defined, but also other metrics (such as *DIT, CBO, CC,* etc.) which are significant. On the other hand, same metric can be correlated with multiple smells, and hence developers should focus on these smells while optimizing the particular metric. For instance, if the goal of a system is to reduce coupling, the developers should focus on the code smells related to the *CBO* metric such as *Anti Singleton, Blob, Complex Class, Large Class, Long Method,* and *Long Parameter List*. Therefore, the relevant and impactful metrics might be useful for developers to optimize the metrics through smell refactoring according to the system's need to improve the maintainability. At the same time, the developers can recognize which metrics are going to be optimized while refactoring a particular smell. These findings help them in objective-based smell-refactoring activities as like objective-based maintenance activities [3].

2. **Impact of code smell:** The three categories of impact level – *High, Moderate* and *Low* for the code smells provide insights to developers about which code smells should be focused on while writing code and developing a system. For example, the developers should be careful about the high impactful smells such as - *Long Method, Anti Singleton, Complex Class, Large Class,*

and *Long Parameter List*. Therefore, the quality of the system will be improved from the beginning of the system's life cycle, which results in increasing maintainability in the long run. In addition, it is difficult for the developers to work with a large number of smells. So, the short list of impactful smells helps them to prioritize refactoring tasks for the smells based on their impact and the system's need.

3. **Relation between frequency and impact of code smell:** Since all code smells do not occur with similar frequencies, developers might not need to focus on all of these. For instance, smell with both low frequency and low impact such as *Many Field Attributes But Not Complex* can be avoided, as low frequency means that the overall impact of the smell on the system is not very significant. On the contrary, smells having both high frequency and high impact such as *Long Method, Complex Class, Large Class,* and *Long Parameter List* should be given higher priority due to their overall high impact on the system. Moreover, smells with moderate frequency but high impact like *Anti Singleton* should also be given high priority while maintaining the system.

5.7 Summary

The results of this chapter clearly show that code smells should be carefully monitored by developers, since these smells are related with a lot of software quality metrics such as size, coupling, complexity, etc. It is significant in a sense that, code smells are correlated with both smell definition and non-definition based metrics. This, in fact, helps developers to prioritize refactoring activities of the smells while optimizing a set of metrics of a system. The results also conclude that all the smells do not have the same impact on software metrics and some smells are highly impactful having a higher frequency. The shortlist of the most impactful

smells having higher frequency can help to write or reuse source code carefully without inducing these smells from the beginning of software development. Moreover, these findings not only help practitioners but also researchers to develop automated refactoring tools based on the impact of code smells and prioritize the smells.

CHAPTER

6

MAINTAINABILITY METRIC BASED IMPACT ANALYSIS OF CODE SMELLS

Code smells, which indicate potential design problems, have a negative impact on software maintainability in terms of two significant metrics – change-proneness and fault-proneness. Previous literature has explored the detrimental impacts of code smells on maintainability and has prioritized them based on system-specific characteristics. That is, prioritization of the smells is possible for a developed system and it varies from system to system. However, it remains unclear how each type of code smell impacts these metrics individually and how to prioritize them regardless of system dependencies. Moreover, it is a very difficult and time-

consuming task for developers to work with a long list of code smell types. Also, not all of those smell types have a similar impact on the systems, and hence highly impactful smell types should be refactored with high priority. To address this research gap, in this chapter, we conduct an empirical study to identify the impact of the code smell types on change- and fault-proneness metrics and prioritize them accordingly. Similar to the previous chapter, for this study, we analyze 13 code smell types in 35 open-source Java systems. Specifically we analyze these to (1) examine their relationship with the two maintainability metrics, and (2) prioritize the code smells based on the relationship. The findings of this chapter reveal that different types of code smells have varying levels of impact on maintainability such as *high, moderate* and *low* which are system independent. Our results show that among the 13 smell types, there exist 4 high prioritized smell types, such as *Anti Singleton, Blob, Class Data Should Be Private* and *Long Parameter List* which have a high impact on the systems. These findings not only guide developers in smell prioritization to enhance maintainability but also researchers to innovate refactoring tools that consider the impact of the smell types.

6.1 Introduction

Code smells have a significant impact on two important maintainability metrics – change-proneness and fault-proneness, which directly influence developers' activities [19]. Change-proneness refers to the ease with which a software system can be modified or adapted to meet evolving requirements. On the other hand, fault-proneness refers to the likelihood that a software system will contain defects or bugs. Code smells can make code more difficult to change as well as increase the likelihood of introducing new faults by making code harder to understand and maintain [10, 19]. Consequently, this can increase the time and effort required for making changes and resolving faults, leading to cost overruns and slowing down

the development and maintenance processes. Thus, code smells are important design factors that affect these two maintainability metrics of a software system.

Usually, developers introduce code smells due to less awareness of design and program comprehensibility [1]. During continuous software maintenance, developers often unintentionally introduce various code smells into a software system [49, 184, 185]. Indeed, it is very difficult for the developers to remove or work with all the code smells at a time especially from a developed system, as it is a huge time-consuming task. It is also not wise to refactor only known or frequent code smell types, as not all of the smell types have similar impact or importance to be removed. For instance, several smell types might have less impact on software maintainability, and also several ones occur with very low frequency, which developers can focus on with less priority, as lower frequency means lower effect on maintainability [2]. Therefore, it is necessary to identify such code smell types having higher impacts on the software maintainability to be refactored. In this study of the chapter, we define an *impactful code smell* as – *a code smell is impactful if it has an impact or effect on a software system, especially on software maintainability metrics – change- and fault-proneness.* In other words, a code smell is highly impactful if it is highly correlated with the metrics, and vice versa.

By identifying the potential impactful code smell types in advance, developers can proactively refactor those smells (if applicable) before reusing code from other systems [2]. Such type of activity will enhance the maintainability of the system in the long run. Having a concise list of impactful code smell types, instead of addressing all or only the well-known smell types, helps developers to focus on and prioritize only those smells during system development and maintenance. Additionally, when it comes to intuition-based refactoring, some developers may focus on refactoring higher impactful smell types, while others may focus on refactoring lesser impactful ones. This might not be the goal of objective-based

maintenance activity [3], where the objective is to improve code maintainability in lesser time through refactoring the impactful smell types. It is important to establish the relationships between the code smell types and the maintainability metrics to identify the impactful ones. This relationship also facilitates the transition from intuition-based to objective-based refactoring. For instance, optimizing *change-proneness* or *fault-proneness* of a system is a key objective of maintenance activities such as corrective, adaptive and perfective tasks [3]. Therefore, the developers need to refactor the impactful code smells to achieve these maintenance objectives effectively.

Previous studies have investigated the relationship between code smells and the two significant maintainability metrics – change- and fault-proneness. For instance, Khomh et al. [19] and Palomba et al. [10] investigated the impact of code smells on these two maintainability metrics. Particularly, they showed that smelly classes are highly change-prone and fault-prone than non-smelly classes. Abbes et al. [37] showed that multiple code smells affecting the same source code have a negative impact on program comprehensibility. Subjective factors such as developers' perceived criticality [**?**] and developers' relevance have been used to prioritize code smells [21, 22, 23].

We have identified a couple of research gaps in the existing literature. Firstly, code smell prioritization has been performed in the literature which is mostly system (or project) specific and/or requires developer's intervention. That is, prioritization of the smells is possible for a developed system and it varies from project to project. In other words, there is a lack of generic smell prioritization approaches that are system-independent. The system-independent smell prioritization helps new developers to recognize and work with the prioritized smell types in advance. It is significant especially while developing a new system to improve its maintainability in the long run. Secondly, while previous works have established

a significant relationship between code smells and maintainability measures, the specific impact of individual smell type on these metrics remains unclear. Specifically, the frequency of which code smell types are highly related to these metrics has not been thoroughly explored. To address these research gaps, we present an empirical study aimed at investigating the relationship (or impact) between the frequency of each individual code smell type and the two maintainability metrics. We then prioritize the code smell types based on their individual impact.

In order to conduct the analysis, an empirical study has been carried out on 35 open-source Java systems. From these systems, we have identified the frequencies of 13 types of code smells and two maintainability metrics – change-proneness and fault-proneness to further measure the relationship between them, and identify the impactful code smell types. In the context of this chapter, the 'impact' of a code smell type refers to the relationship based on the correlation between the code smell type and each of the maintainability metrics. In addition, based on the correlation analysis, we prioritize the code smell types which are independent of systems. Also, we analyze which impactful code smell types are highly frequent, that is, the higher occurrences in the systems.

In summary, this chapter makes the following contributions:

(i) The relationship between each of the code smell types and two software maintainability metrics – change-proneness and fault-proneness to quantify the impact of the smell types.

(ii) Three prioritization levels (*High, Moderate* and *Low*) of the code smell types which can be used to improve software maintainability.

Structure of the chapter: Section 6.2 describes the design of the empirical study. Section 6.3 presents and discusses the results of the study, while the threats to their validity are discussed in Section 6.4. Finally, Section 6.5 discusses its key findings of this chapter.

6.2 Empirical Study Design

The objective of the empirical study is to prioritize 13 types of code smell by evaluating their impact on the two significant software maintainability metrics – change-proneness and fault-proneness. By analyzing 35 open-source software systems, the study particularly tries to identify the impactful set of code smell types which have impacts on these two metrics. For this chapter, it is important to note that the term 'impactful' when associated with a type of code smell, refers to the relationship between each of the metrics and the code smell type. In fact, some smell types may have a strong relationship with the metrics but fewer occurrences in the systems, or vice versa. Consequently, frequency analysis of the code smell types along with their impacts is also another objective of the study.

6.2.1 Formulating Research Goal

To prioritize code smell types, we first need to determine their impact on the two important maintainability metrics – change- and fault-proneness individually. Then for each code smell type, we combine their impact that is being used in smell prioritization. More specifically, to achieve this goal, we answer the following research questions:

RQ1: ***How is the frequency of each code smell type related with change-proneness?***

This research question investigates the frequency of which code smell types have a relationship with change-proneness by analyzing their frequency of occurrences in software systems. Code smell types those have a higher correlation with the change-proneness are considered as highly 'impactful smells' for this RQ1. Therefore, a correlation-based analysis is performed to identify the relationship between them and to show whether smell frequency has a relationship with the change-proneness. Based on the relationship strength, the code smell types are

categorized into three impact levels – *High, Moderate* and *Low*. The answer to RQ1 will help software developers to focus on the smell types which have a higher impact on the change-proneness to minimize it.

RQ2: ***How is the frequency of each code smell type related with fault-proneness?***

This research question aims to explore the association between the frequency of different types of code smells and fault-proneness in software systems. Code smell types that show a higher correlation with fault-proneness are regarded as highly 'impactful smells' for the purpose of this RQ2. To determine the relationship between code smell frequency and fault-proneness, a correlation-based analysis is conducted. Additionally, the code smell types are categorized into three impact levels – *High, Moderate* and *Low* based on this relationship. The findings of this investigation, addressing RQ2, can assist software developers in prioritizing the identification and mitigation of code smells that have a significant impact on fault-proneness.

RQ3: ***How to prioritize code smell types based on their impact on both change- and fault-proneness?***

The research question focuses on prioritizing code smell types based on their impact on the two significant metrics – change-proneness and fault-proneness. By combining the correlations from RQ1 and RQ2, the aim is to determine a prioritization list of the code smell types. This list will assist developers in identifying impactful code smell types which need attention during refactoring activities, especially improving the maintainability of software systems.

6.2.2 Systems Under Study

In order to conduct the empirical study and answer the research questions, similar to Chapter 5, we have analyzed 35 open-source Java systems mostly belonging to

two major ecosystems: Apache[1] and Eclipse[2]. Table 6.1 summarizes the analyzed systems, the latest stable version of the systems up to this study, and their size in terms of number of files (NOFs), number of methods (NOMs), and lines of code (LOCs).

The systems have been selected because – (i) the systems are well-known in the research domain of code smells and refactorings [10, 21]; (ii) the systems are in different sizes (from 8,069 to 2,097,667 LOCs), large enough having an average NOFs of 3,581; NOMs of 38,450 and LOCs of 460,522; (iii) the selected systems cover a variety of application domains such as *ArgoUML* - a UML-based system modeling tool, *Eclipse* - an IDE, *Lucene* - a search manager, *Xerces* - an XML parser, etc.; and (iv) Many of these systems (* symbol in the table's Id No. column) exploit Bugzilla[3] or Jira[4] as an issue tracker for identifying fault-proneness related information. Therefore, we use these systems for analyzing the impact of fault-proneness as shown * in the table and all the systems for analyzing the impact of change-proneness in our study.

6.2.3 Selection and Detection of Code Smell Types

A list of 13 types of code smell has been selected to carry out the study, similar to the previous chapter. Table 6.2 also lists these 13 code smell types along with a brief description. For the purpose of detecting these smell types from the selected systems, a state-of-the-art tool called DÉCOR [82] has been employed because the tool detects a large number of code smell types (18 smell types), shows good accuracy, and has widely been used in the prior studies [175, 10, 19]. It is noted, in this study, we have excluded five code smell types out of 18 detected by DÉCOR – *Base Class Knows Derived Class, Functional Decomposition, Message Chain,*

[1] https://www.apache.org.
[2] https://www.eclipse.org.
[3] http://www.bugzilla.org.
[4] https://www.atlassian.com/software/jira.

Table 6.1: Software systems involved in the study

Id No.	Project	Version	NOFs	NOMs	LOCs
1*	Activemq	5.17.0	4,344	42,349	421,979
2*	Ant	1.9.0	1,256	13,768	138,391
3*	Ant-ivy	2.5.0	685	7,540	76,678
4	ArgoUML	0.35.1	1,969	17,955	177,402
5*	Cassandra	4.0.0	7,395	76,237	1,483,583
6*	Cayenne	4.1	3,195	41,339	485,781
7	CDT	10.6.0	3,763	26,883	267,165
8*	CXF	3.5.0	9,630	103,490	1,120,148
9	DeepLearning4j	1.0.0	7,795	59,725	698,622
10*	Drill	1.10.0	4,670	60,203	533,478
11	Eclipse Core	R4.9	2,376	32,977	305,762
12	ElasticSearch	8.3.0	16,250	154,096	2,097,667
13	Freemind-mmx	1.0.0	522	6,804	63,933
14*	Hadoop	3.3.0	11,041	143,785	1,762,363
15*	HBase	3.0.0	4,707	71,619	823,856
16*	Hive	3.1.0	6,556	107,711	1,257,002
17	Hsqldb	2.7.0	935	13,874	208,333
18*	Incubator-livy	0.7.0	100	811	8,069
19*	Jackrabbit	2.9.0	3,121	29,053	332,826
20	JBoss-modules	2.0.3	244	1,956	20,586
21*	Jena	4.5.0	6,084	66,138	576,575
22	JFreeChart	1.5.3	1,012	11,336	136,786
23	JHotDraw	9.0	671	7,630	80,440
24*	Karaf	4.4.0	1,607	10,332	126,431
25*	Lucene	9.3.0	5,465	52,968	827,867
26	Mahout	14.0	1,262	9,552	111,437
27*	Nutch	2.4	521	3,136	38,268
28	Opennlp	1.9.4	1,008	5,303	75,724
29*	Pig	0.10.0	1,563	15,433	231,371
30*	Poi	5.2.2	3,648	40,032	413,657
31*	Qpid	0.3	2,827	29,189	327,821
32*	Struts	6.0.0	2,366	18,640	186,595
33	Tomcat	10.0.23	2,620	33,433	355,081
34*	Wicket	7.2.0	3,293	20,678	215,286
35*	Xerces	2.9.0	825	9,809	131,326
Total		**-**	**125,326**	**1,345,784**	**16,118,289**
Average		**-**	**3,581**	**38,450**	**460,522**

Swiss Army Knife and *Traditional Breaker* as these smells do not occur in almost most of the systems [2].

Table 6.2: Code smell types involved in the study

Id No.	Code Smell Type	Description
1	Anti Singleton (AS)	A class that provides mutable class variables, which consequently could be used as global variables.
2	Base Class Should Be Abstract (BCSBA)	An inheritance tree contains roots that are not abstract - only the leaves should be concrete.
3	Blob	A class that is too large and not cohesive enough, that monopolises most of the processing of a system, takes most of the decisions, and is associated to data classes.
4	Class Data Should Be Private (CDSBP)	A class that exposes its fields, thus violating the principle of encapsulation or data hiding.
5	Complex Class (CC)	A class that contains (at least) one large and complex method, in terms of cyclomatic complexity and LOCs.
6	Large Class (LC)	A class that is large in size in terms of LOCs.
7	Lazy Class (LzC)	A class that has few fields and methods (with little complexity), and that does not do too much in the system.
8	Long Method (LM)	A method that is very large in size and complex.
9	Long Parameter List (LPL)	A method that contains a long list of parameters.
10	Many Field Attributes But Not Complex (MFABNC)	A class that is not complex but has too many fields those are public.
11	Refused Parent Bequest (RPB)	A class that inherits functionalities from its parent but never uses thses, thus violating polymorphism.
12	Spaghetti Code (SC)	A class without structure that declares long methods without parameters, thus prevents polymorphism.
13	Speculative Generality (SG)	A class that is defined as abstract but it has very few children, which does not make use of its methods.

We have chosen the 13 types of code smell for the study because – (1) these are representative of various object-oriented design issues such as inheritance, polymorphism, encapsulation, etc., (2) the frequencies of these smells in the systems are great in numbers to conduct the empirical study, and (3) their frequencies are almost similar between open-source and industrial systems [2].

6.2.4 Detection of Change-proneness

The change-proneness of a class is computed as the number of total changes between two successive releases of a system [10]. Both addition and deletion of statements are considered as changes to a class. So, the number of changes of a class C_i is calculated using equation (6.1).

$$\#changes(C_i) = added(C_i) + deleted(C_i) \tag{6.1}$$

Here, $added(C_i)$ refers to the number of added lines and $deleted(C_i)$ refers to the number of deleted lines in between two consecutive releases. To compute $added(C_i)$ and $deleted(C_i)$ of each class, commit history is analyzed from the version control system. It is noted that the successive two releases of a system are the version given in Table 6.1 and its previous stable version. For instance, change-proneness for the *Ant* system is calculated from the version 1.8.0 to 1.9.0.

For example, consider the changes between two successive releases in *AntFilterReader.java* class from *Ant* project as shown in Listing 6.1. There are four deleted lines (-, red) and two added lines (+, blue) in this code snippet. So, the number of total changes is six which is considered as the value of change-proneness for *AntFilterReader.java* class.

Listing 6.1: AntFilterReader.java

```
@@ -134,12 +135,10 @@ public final class AntFilterReader extends DataType {
        public Parameter[] getParams() {
                if (isReference()) {
-                       ((AntFilterReader) getCheckedRef()).getParams();
+                       getRef().getParams();
                }
                dieOnCircularReference();
-               Parameter[] params = new Parameter[parameters.size()];
-               parameters.copyInto(params);
-               return params;
+               return parameters.toArray(new Parameter[0]);
        }
```

Furthermore, we split change-proneness based on the classes of each individual type of the 13 code smells to have smell level change-proneness. For example, change-proneness of classes having *Anti Singleton* smell type are separated to have of that smelly change-proneness. In this way, we have identified 13 types of smelly change-proneness for the 13 code smell types. We call this as smelly change-proneness of the corresponding code smell type.

6.2.5 Detection of Fault-proneness

The fault-proneness of a class is measured as the number of bug-fixing changes in a class between two successive releases of a system [10]. It is because after fixing a fault, then it can be detected from the source code. It is noted that, similar to the previous sub-section, the successive two releases of a system are the version given in Table 6.1 and its previous stable version. However, these changes include both addition and deletion of statements in source code through commits which are referred as bug-fixing commits. Bug-fixing commits are identified by searching commits that contain *Issue ID* or *Bug ID* in commit messages.

The bug repository of each project uses a unique identifier to track bugs or issues that follows a specific pattern. As the identifier varies from project to project, a manual exploration is performed on the bug repository of each project to find the pattern of *Issue ID* or *Bug ID*. Then, the regular expressions are formatted for each project that are used in searching commit messages.

For example, the following commit message from *Cayenne* system [5] can be identified with the regular expression "*CAY-\\d+*":

"CAY-2732 Exception when creating ObjEntity from a DbEntity"

The complete list of bug repository URLs and the regular expressions (or bug patterns) used for each project can be found in the online appendix [6].

Using the *Issue ID* found in commit messages, the corresponding issue report is extracted from the issue tracker such as *Jira* and *Bugzilla*. Issues related to bugs are separated, that is, the type of issue is *bug*, excluding *enhancement* type issues. To exclude duplicated or false-positive bugs that can bias the result, only bugs that have the status *Closed* or *Resolved* and the resolution *Fixed* are considered.

[5] https://github.com/apache/cayenne

[6] https://bitbucket.org/maintainabilityvscodesmell/smellimpactstudy/src/master/

Finally, the number of bug-fixing changes of a class is calculated as the sum of added and deleted lines only through bug-fixing commits.

For example, Listing 6.2 shows the fixing changes through bug-fixing commits between two successive releases in *MessageDatabase.java* class from *ActiveMQ* project. There are total six changes, two deleted lines (-, red) and four added lines (+, blue) in this code snippet. So, the value of fault-proneness is considered to be six for *MessageDatabase.java* class.

Furthermore, we split fault-proneness based on the classes of each individual type of the 13 code smells to have smell level fault-proneness. For example, fault-proneness of classes having *Anti Singleton* smell type are separated to have of that smelly fault-proneness. In this way, we have identified 13 types of smelly fault-proneness for the 13 code smell types. We call this as smelly fault-proneness of the corresponding code smell type.

Listing 6.2: MessageDatabase.java

```
1   @@ -4254,8 +4254,10 @@ public abstract class MessageDatabase extends ServiceSupport
        implements BrokerServiceAware {
2
3       @Override
4           protected Class<?> resolveClass(ObjectStreamClass desc) throws
                IOException, ClassNotFoundException {
5   -           if (!(desc.getName().startsWith("java.lang.") ||
                    desc.getName().startsWith("java.util.")
6   -               || desc.getName().startsWith("org.apache.activemq."))){
7   +           if (!(desc.getName().startsWith("java.lang.")
8   +               || desc.getName().startsWith("com.thoughtworks.xstream")
9   +               || desc.getName().startsWith("java.util.")
10  +               || desc.getName().startsWith("org.apache.activemq."))) {
11  +           throw new InvalidClassException("Unauthorized deserialization
                    attempt", desc.getName());
12          }
13          return super.resolveClass(desc);
14      }
```

6.2.6 Definition of Several Metrics

The study is based on the frequency of 13 code smell types and two maintainability metrics – change- and fault-proneness in the systems. Therefore, before analysing

the relationship, we define several metrics for the study as shown in Table 6.3.

Table 6.3: Definition of metrics involved in the study

No.	Metric Name	Difinition
1	Smell Frequency (SF)	frequency of occurrences of a code smell type in a system.
2	Total Smell Frequency (TSF)	frequency of occurrences of a code smell type in all the analyzed systems.
3	Average Smell Frequency (ASF)	frequency of occurrences of a code smell type per system (TSF divided by the number of systems).
4	Change-proneness (CP)	number of changes of a class in a system. Specifically, we calculate smelly CP for a code smell type as the number of changes of a class having the code smell type in a system.
5	Total Change-proneness (TCP)	number of changes of classes in all the analyzed systems. Specifically, we calculate TCP for a code smell type as the number of changes of classes having the code smell type in all the systems.
6	Average Change-proneness (ACP)	number of changes of classes having a code smell type per system (TCP divided by the number of systems).
7	Fault-proneness (FP)	number of faults of a class in a system. Specifically, we calculate FP for a code smell type as the number of faults of a class havingthe code smell type in a system.
8	Total Fault-proneness (TFP)	number of faults of classes in all the analyzedsystems. Specifically, we calculate TFP for a code smell type as the number offaults of classes having the code smell type in all the systems.
9	Average Fault-proneness (AFP)	number of faults of classes having a code smelltype per system (TFP divided by the number of systems).

It is noted, the source code of change-proneness and fault-proneness detection and other dataset including metric scores used in the study have been given in public repository[6] for replication and further research purposes.

6.2.7 Data Analysis

In this sub-section, we explain how we analyze the data to answer the research questions. More specifically, we discuss how we calculate and analyze the metrics, categorize these metrics (particularly *ASF*, *ACP* and *AFP*), calculate the correlation coefficient and interpret it, and finally compare our findings with expert developers' opinion. The data analysis process is shown as a schematic diagram in Figure 6.1.

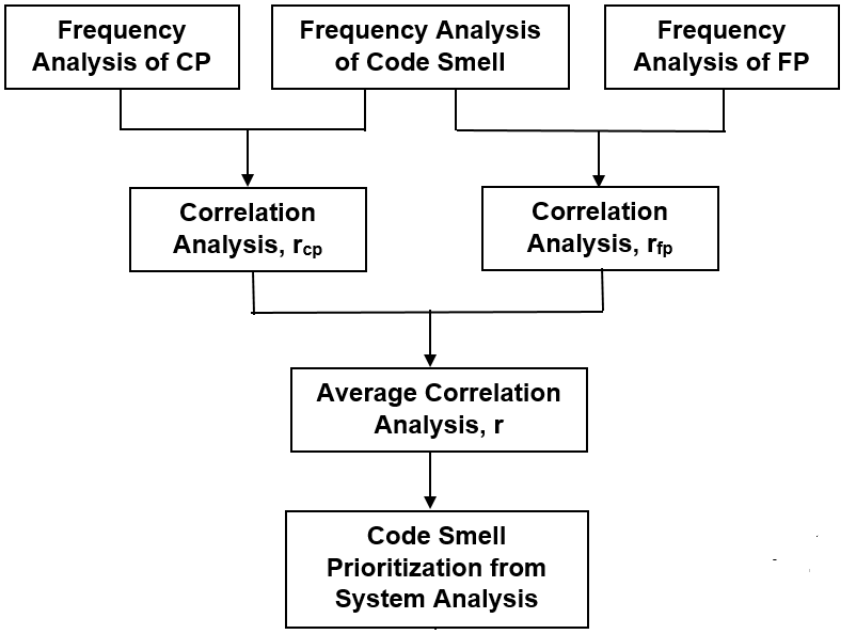


Figure 6.1: Schematic diagram of data analysis process

6.2.7.1 Frequency analysis

The frequency analysis process is shown in Figure 6.2 as a schematic diagram. At first, we have calculated the frequency of each type of code smell, and two maintainability metrics – change- and fault-proneness of the corresponding classes of each smell type in the analyzed systems. Thus for each code smell type, we have calculated *Smell Frequency (SF)*, *Change-proneness (CP)* and *Fault-proneness (FP)* for each of the systems. By summing up the *SF*, *CP* and *FP* for all the systems, we have calculated *Total Smell Frequency (TSF)*, *Total Change-proneness (TCP)* and *Total Fault-proneness (TFP)* respectively for each smell type. After that, we have measured *Average Smell Frequency (ASF)*, *Average Change-proneness (ACP)* and *Average Fault-proneness (AFP)* per system respectively.

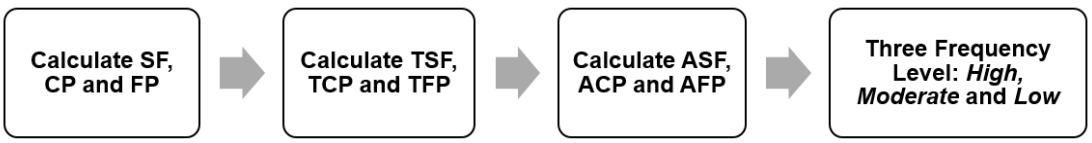


Figure 6.2: Schematic diagram of frequency analysis process

We have categorized the metric *ASF*, *ACP* and *AFP* values into three frequency levels – *High, Moderate* and *Low* based on the statistical quartiles [177] - quartile 1 (Q1), quartile 2 (Q2) and quartile 3 (Q3), where it is *High if the metric value* $\geq$ *Q3; Low if the metric value* $\leq$ *Q1; and Moderate if Q3* $>$ *the metric value* $>$ *Q1*. We have made this categorization because it actually provides insight to understand the frequency level of the corresponding metrics.

The total change-proneness (TCP) and total fault-proneness (TFP) scores differ from one code smell type to another. A more frequent smell type is more likely to have more change-proneness and fault-proneness than less frequent ones. So, the frequency of the smell types has been further studied using a normalized metric - inverse smell frequency, ISF_i *(metric)* score for a particular smell type i in terms of a maintainability metric – TCP or TFP as shown in Equation 6.2 [2]. We have been inspired to use this metric from the Information Retrieval technique, *inverse document frequency* calculation [186].

$$ISF_i(metric) = \frac{metric\ score_i}{smell\ frequency_i} \tag{6.2}$$

Here, $metric\ score_i$: is the metric value – TCP or TFP for a code smell type i. $smell\ frequency_i$: is the TSF value for a code smell type i, such as - *Long Method* smell frequency, or *Blob* smell frequency, or so on.

In particular, the $ISF_i(TCP)$ or $ISF_i(TFP)$ defines the average number of change-proneness (CP) or fault-proneness (FP) respectively against the one occurrence of a particular code smell type i. In other words, it indicates how many CP or FP are responsible for the per occurrence of that code smell type. So the higher value of the ISF_i score indicates the more occurrences of changes and faults for those smell types, and hence these smell types will be more critical for the systems. Also as like earlier, we have categorized the ISF metric score into the three levels – *High, Moderate* and *Low* by statistical quartile analysis.

6.2.7.2 Correlation coefficient analysis

To conduct the data analysis, we have used the Spearman's Rank Correlation Coefficient [177] between the smell frequencies (SF) and change-proneness (CP) [for RQ1], and between the smell frequencies (SF) and fault-proneness (FP) [for RQ2] of the corresponding smelly classes of all the systems. We have used the Spearman's correlation in this study because it is a non-parametric measure, and the datasets are ordinal and do not follow a normal distribution. For instance, we have taken the frequency of *Anti Singleton* smell type of each system as the x variable and change-proneness of the *Anti Singleton* smelly classes of the corresponding systems as the y variable to measure the correlation coefficient between them. Similarly, we have measured the correlation between the *Anti Singleton* smell type and its corresponding fault-proneness by considering their frequencies. Thus we have measured the correlation coefficient for the frequency of 13 smell types with the corresponding change- and fault-proneness.

Specifically, we have followed the guidelines provided by Cohen [178] for interpreting the correlation coefficient (r). It is considered that there is no correlation when $0 \leq r < 0.1$, small correlation when $0.1 \leq r < 0.3$, medium correlation when $0.3\ leq\ r < 0.5$, and strong correlation when $0.5 \leq r \leq 1$. Furthermore, based on the correlation analysis, we have categorized the 13 code smell types into three impact levels - *High, Moderate* and *Low*. Here, we have considered strong correlation as *High*, medium as *Moderate* and rest as *Low* impact level of the code smell types. Also, these impact levels are considered as the prioritization levels for the smell types in this research as higher impact means higher prioritized smell types.

It is noted that we have excluded four code smell types (*MFABNT, RPB, SC* and *SG*) from the correlation analysis because of satisfying the following two conditions based on the *spreadity* score metric [2] – (i) the spreadity score as

shown in Equation 6.3 of each of the four code smell types is less than 0.80, and (ii) the spreadity of change-proneness (CP) and fault-proneness (FP) as shown in Equation 6.3 for each smelly classes is less than 0.5 for the analyzed systems, that is, most of the systems (more than 50% of the systems) do not contain any changes and faults for those smelly classes, indicating no relationship exists. The spreadity score of all the smell types has been shown in Table 6.4. We consider the remaining nine smell types for the correlation analysis, as either spreadity score of the smell types is equal to or greater than 80%, or spreadity score of the corresponding smelly CP or FP is equal or greater than 50%.

$$spreadity(x) = \frac{n(x)}{N} \tag{6.3}$$

Here, x = attribute (x can be a code smell, or change-proneness, or fault-proneness of smelly classes for a code smell type), $n(x)$ = number of systems containing attribute x, and N = total number of systems. Higher value of the metric for an attribute indicates higher number of systems contain the attribute. The maximum value will be 1 if all the systems contain the attribute, and it will be zero if none of the systems contain it.

6.3 Result Analysis

For the prioritization of code smells, the impact of the smells on the two maintainability metrics – change- and fault-proneness individually has been identified and their combined impact has also been determined. The detailed result analysis is explained in this section.

Table 6.4: Spreadity score of code smell, and their corresponding CP and FP

	Spreadity Score		
Code Smell Type	**Code Smell**	**Corresponding Smelly CP**	**Corresponding Smelly FP**
AS	1.00	1.00	0.86
BCSBA	**0.74**	0.60	0.50
Blob	0.91	0.86	0.68
CDSBP	1.00	0.91	0.77
CC	1.00	1.00	1.00
LC	1.00	1.00	0.95
LzC	0.94	0.74	**0.36**
LM	1.00	1.00	1.00
LPL	1.00	1.00	0.91
MFABNC	**0.51**	**0.31**	**0.09**
RPB	**0.60**	**0.37**	**0.09**
SC	**0.77**	**0.49**	**0.41**
SG	**0.60**	**0.46**	**0.23**

[* Bold font in the values indicates low spreadity score]

6.3.1 Impact of Each Code Smell Type on Change-proneness [RQ1]

The result of RQ1 that the relationship based on the correlation analysis between each code smell type and its change-proneness is discussed in this sub-section. Based on the relationship, the impact of each smell type on the change-proneness is analyzed.

6.3.1.1 Frequency of the code smell types and change-proneness of the respective smelly classes

Table 6.5 (columns 2 to 5) shows the scores of smell and change-proneness related metrics such as *Total Smell Frequency (TSF), Total Change-proneness (TCP), Average Smell Frequency (ASF)* and *Average Change-proneness (ACP)* respectively. Also, Figure 6.3a and 6.3b show the *ASF* and *ACP* for each code smell type respectively in order to visualize their frequencies.

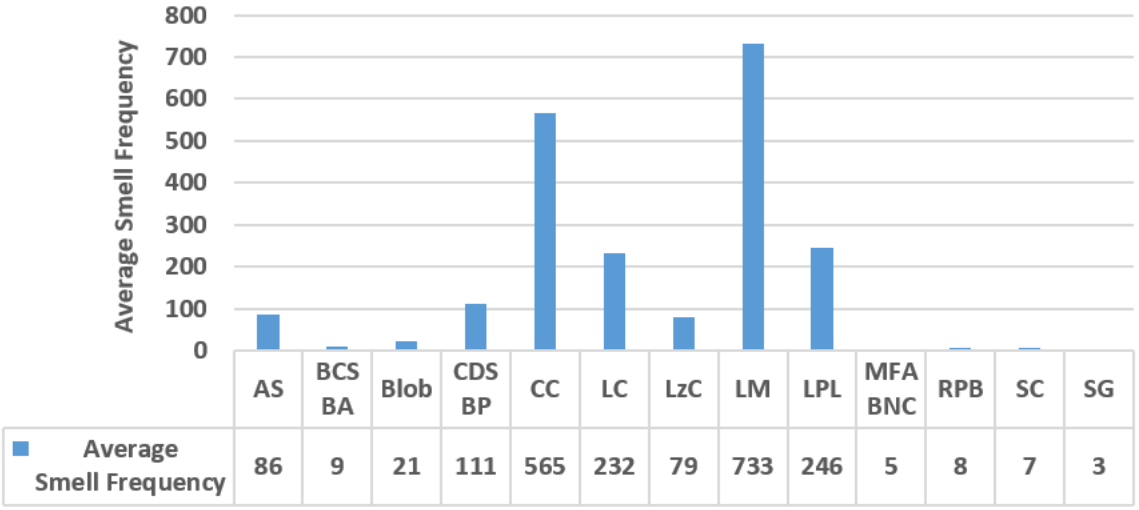
From these figures and based on the quartile analysis as discussed in the Sub-

Table 6.5: Impact of each code smell type and its corresponding change-proneness

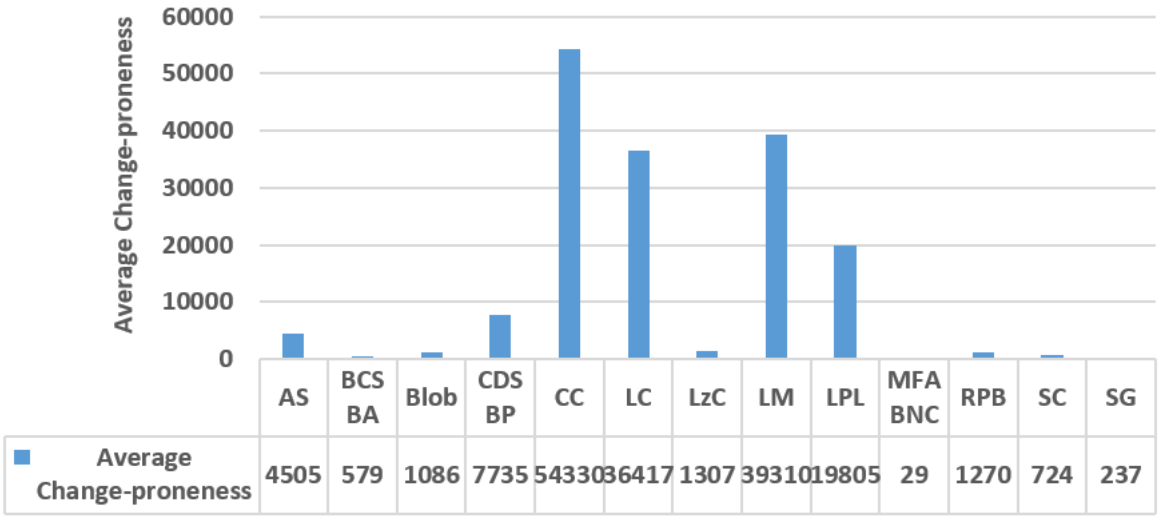
Code Smell	Total Smell Frequency, *TSF*	Total Change-proneness, *TCP*	Average Smell Frequency, *ASF*	Average Change-proneness, *ACP*	*ISF(TCP)*	Correlation, r_{cp}	p-value
AS	3022	157691	86	4505	52.18	0.57	0.000
BCSBA	306	20263	9	579	66.22	0.11	0.589
Blob	732	38006	21	1086	51.92	0.39	0.029
CDSBP	3871	270736	111	7735	69.94	0.57	0.000
CC	19777	1901561	565	54330	96.15	0.38	0.024
LC	8106	1274596	232	36417	157.24	0.28	0.102
LzC	2766	45735	79	1307	16.53	0.48	0.005
LM	25657	1375846	733	39310	53.62	0.40	0.018
LPL	8602	693188	246	19805	80.58	0.61	0.000
MFABNC	160	1015	5	29	NA	0 (NA)	NA
RPB	277	44435	8	1270	NA	0 (NA)	NA
SC	251	25333	7	724	NA	0 (NA)	NA
SG	110	8310	3	237	NA	0 (NA)	NA

section 6.2.7, we have observed that *Long Method, Complex Class, Long Parameter List* and *Large Class* code smell types have high frequency of occurrences, and at the same time these smelly classes have high change-proneness. On the other hand, smell types with low frequency such as *MFABNC, SC* and *SG* have a low number of changes. Also, smell types with moderate frequency such as *AS, Blob, CDSBP* and *LzC* have a moderate number of changes. Finally,*BCSBA* has moderate smell frequency but low change-proneness, and *RPB* has low smell frequency but moderate change-proneness. From this analysis and graphical representation, we conclude that *in most cases, a similar level of smell frequency has a similar level of change-proneness for each individual code smell type.*

We have further observed the *ISF(TCP)* score for each smell type in the sixth column of Table 6.5. This change-proneness related metric indicates how many changes are occurred for the single (or one) occurrence of a smell type. Therefore, the more the score for a smell type is, the more the smell type is critical for a system with respect to the change-proneness. For example, for the *Long Method (LM)* smell type, $ISF(TCP) \approx 157$ means that for one occurrence of *LM* smell, on an average 157 changes are occurred, whereas for *Lazy Class (LzC)* smell type the score is only 16.53. Here, we have excluded the last four smell types in this analysis for the same reason as discussed earlier in the Data Analysis sub-section (6.2.7). Based on the quartile analysis of the metric, firstly it is observed that



(a) Average smell frequency



(b) Average change-proneness of each code smell type

Figure 6.3: Average smell frequency and change-proneness of each code smell type

three smell types – *CC, LC* and *LPL* have high *ISF(TCP)* scores which are high frequent ones as well. Secondly, another three smell types – *BCSBA, CDSBP* and *LM* have moderate *ISF(TCP)* scores which are moderate frequent ones except *LM* (high frequent). Finally, the rest of the three moderate frequent smell types – *AS, Blob* and *LzC* have low *ISF(TCP)* scores. From this analysis, we conclude that *levels of smell frequency and ISF(TCP) are not similar, and in particular moderate frequent smell types have different levels of ISF(TCP) score.*

6.3.1.2 Correlation coefficient

From the above findings, it has been seen that frequencies of different code smell types have different levels of change-proneness according to the frequency analysis of the metrics. This leads to the correlation analysis to identify whether the frequency of the smell types really impacts the change-proneness and how the relationship exists between them.

The result of the correlation analysis between the frequency of each code smell type and its corresponding change-proneness, as well as the respective p-values have been shown in the last two columns of Table 6.5. It is noted that, as said earlier, we have excluded four code smell types (*MFABNT, RPB, SC* and *SG*) from the correlation analysis because most of the systems do not contain any changes for those smelly classes indicating no relationship exists. Therefore, the table shows *0 (Not Applicable, NA)* values for those smell types. We consider these four smell types including *BCSBA* and *LC* as *Low* impactful smells as their $r_{cp} < 0.3$, and hence developers might focus on these smells with lower priority. On the other hand, according to the discussion in Sub-section 6.2.7 (Data Analysis), *AS, CDSBP* and *LPL* are *High* impactful smell types as their $r_{cp} \geq 0.5$, and *Blob, CC, LM* and *LzC* are *Moderate* impactful smell types as their $0.3 \geq r_{cp} < 0.5$. So developers should focus on these smell types with higher priority while working on change-related maintenance activities. It has also been observed that the correlation coefficient is significant for the seven out of nine code smell types where *p-value* < 0.05 considering 5% level of significance. However, all of these seven smell types have high and moderate correlation with the change-proneness ($r_{cp} > 0.3$)

6.3.1.3 Smell frequency versus correlation coefficient

From the Table 6.5, we have found an interesting observation that *Anti Singleton, Class Data Should Be Private* and *Long Parameter List* have strong correlation, that is, high relationship with the change-proneness, while their average smell frequencies (*ASF*s) are not very high except for *Long Parameter List* smell type. It is possible that changes occur more in method levels where the number of parameters is higher, and therefore its change-proneness gets higher in accordance with its frequency. On the other hand, medium (moderate) correlated smell types such as - *Complex Class* and *Long Method*, and small (or low) correlated smell types such as - *Large Class* have higher smell frequency. In addition, *Base Class Should Be Abstract* has low correlation but moderate frequency. Only *Blob* and *Lazy Class* smell type have both moderate correlation and frequency. In addition, low correlated smell types such as *MFABNC, RPB, SC* and *SG* have low smell frequency, and it is expected that lower frequent smell types have lower change-proneness because of their lower number of classes in the systems. Therefore, for the research question, we have focused on the high and moderate correlated code smell types, and observed the general perception that higher frequent smell types might have higher correlation with the change-proneness which is not correct. Rather different levels of smell frequencies have different levels of correlation strengths. However, *in many cases while considering high and moderate correlated code smell types, smell frequencies are almost opposite of their relationship strength.*

From the frequency graphs in Figure 6.3, we observe that higher frequent smell types have higher change-proneness whereas their correlations do not follow it. To make it clear, the correlations show how the change-proneness of a smell type will increase (or decrease) when the smell frequency increases (or decreases). On the other hand, frequency just indicates their frequency of occurrences in the systems.

Summary for RQ1. While considering each code smell type individually, most of the smell types are correlated with the change-proneness except for the four smell types. Among these *AS, CDSBP* and *LPL* are highly correlated smell types. In addition, smell frequency has different impact levels on the change-proneness such as high and moderate correlated smells have moderate and high smell frequencies respectively.

6.3.2 Impact of Each Code Smell Type on Fault-proneness [RQ2]

Fault-proneness is another important maintainability measure of a software system and minimizing fault-proneness is one of the goals of software developers. In this sub-section, we separately discuss the impact of each type of code smells on fault-proneness, as like in the previous sub-section on change-proneness. More specifically, we discuss the result of RQ2 here.

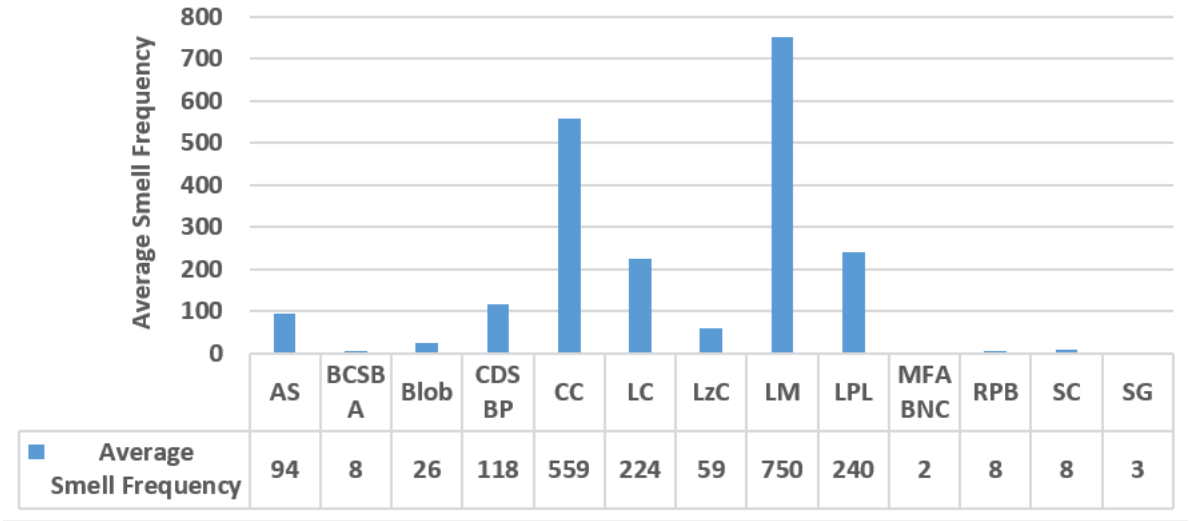
6.3.2.1 Frequency of the code smell types and fault-proneness of the respective smelly classes

Table 6.6 (columns 2 to 5) shows the scores of smell and fault-proneness related metrics such as *Total Smell Frequency (TSF), Total Fault-proneness (TFP), Average Smell Frequency (ASF)* and *Average Fault-proneness (AFP)* respectively. Also, Figure 6.4a and 6.4b show the *ASF* and *AFP* for each code smell type respectively to visualize their frequencies.

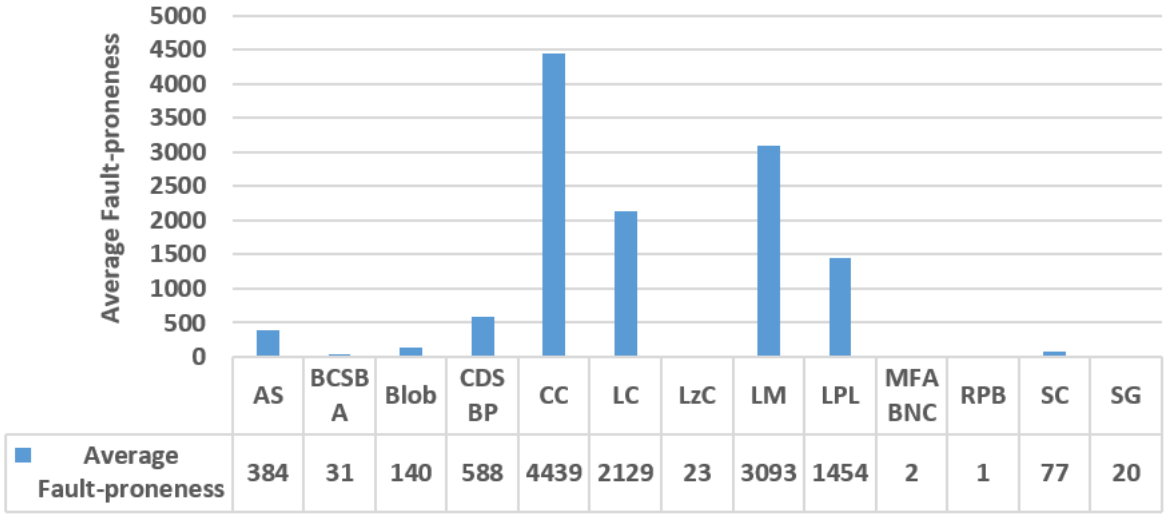
From these figures and based on the quartile analysis as discussed in the Subsection 6.2.7, we have observed that *Long Method, Complex Class, Large Class* and *Long Parameter List* code smell types have high frequency of occurrences, and at the same time these respective smelly classes have high fault-proneness. On the other hand, smell types with low frequency like *MFABNC, RPB* and *SG* have lower number of faults. Also, smell types with moderate frequency like *AS, Blob*

Table 6.6: Impact of each code smell type and its corresponding fault-proneness

Code Smell	Total Smell Frequency, TSF	Total Fault-proneness, TFP	Average Smell Frequency, ASF	Average Change-proneness, AFP	$ISF(FP)$	Correlation, r_{fp}	p-value
AS	2069	8449	94	384	4.08	0.71	0.000
BCSBA	169	689	8	31	4.08	0.28	0.213
Blob	569	3071	26	140	5.40	0.64	0.001
CDSBP	2590	12940	118	588	5.00	0.51	0.015
CC	12298	97652	559	4439	7.94	0.21	0.337
LC	4928	46828	224	2129	9.50	0.24	0.278
LzC	1302	497	59	23	0.38	0.46	0.030
LM	16492	68039	750	3093	4.13	0.35	0.109
LPL	5278	31981	240	1454	6.06	0.70	0.000
MFABNC	37	36	2	2	NA	0 (NA)	NA
RPB	168	14	8	1	NA	0 (NA)	NA
SC	182	1702	8	77	NA	0 (NA)	NA
SG	73	430	3	20	NA	0 (NA)	NA



(a) Average smell frequency



(b) Average fault-proneness of each smelly classes

Figure 6.4: Average smell frequency and fault-proneness of each code smell type

and *CDSBP* have moderate number of faults. Finally, *BCSBA* and *SC* smell types have low smell frequency but moderate fault-proneness, and Finally, *LzC* smell type has moderate smell frequency but low fault-proneness. From this analysis, we conclude that in most cases, similar level of smell frequency has similar level of fault-proneness for each individual code smell type.

We have further observed the *ISF(FP)* score for each smell type in the sixth column of Table 6.6 (here, *FP* is equivalent to *TFP*). This fault-proneness related metric indicates how many faults are occurred for the one occurrence of a smell type. Therefore, the more the score for a smell type is, the more the smell type is critical for a system with respect to the fault-proneness. For example, for the *Large Class (LC)* smell type, $ISF(FP) \approx 9$ means that for one occurrence of *LC* smell, on an average 9 faults are occurred, whereas, for *Anti Singleton (AS)* smell type, the score is approximately only 4. Here, we have excluded the last four smell types in this analysis for the same reason as discussed earlier in the Data Analysis subsection (6.2.7). Based on the quartile analysis of the metric, firstly it is observed that three smell types – *CC, LC* and *LPL* have high *ISF(FP)* score which are high frequent ones as well. Secondly, another three smell types – *Blob, CDSBP* and *LM* have moderate *ISF(FP)* score which are high frequent ones except *LM* (high frequent). Finally, three smell types having low *ISF(FP)* score – *AS, BCSBA* and *LzC* have moderate smell frequency except *BCSBA* (low frequent). From this analysis, we conclude that levels of smell frequency and *ISF(FP)* are not similar, that is, different levels of frequent smell types have different levels of *ISF(FP)* scores.

6.3.2.2 Correlation coefficient.

From the above findings, it has been seen that different types of smell frequencies have different levels of fault-proneness according to the frequency analysis of the metrics. This leads to the correlation analysis to identify whether the frequency of

the smell types really impacts the fault-proneness and how the relationship exists between them.

The result of the correlation analysis between the frequency of each code smell type and its corresponding fault-proneness, as well as the respective p-values have been shown in the last two columns of Table 6.6. It is noted that, as discussed earlier, we have excluded four code smell types (*MFABNT, RPB, SC* and *SG*) from the correlation analysis because most of the systems do not contain any faults for those smelly classes indicating no relationship exists. Therefore, the table shows *0 (Not Applicable, NA)* values for those smell types. Therefore, we consider these four smell types including *BCSBA, CC* and *LC* as *Low* impactful smells as their $r_{fp} < 0.3$, and hence developers might focus on these smells with lower priority. On the other hand, according to the discussion in Sub-section 6.2.7 (Data Analysis), *AS, Blob, CDSBP* and *LPL* are *High* impactful smell types as their $r_{fp} \geq 0.5$, and *LzC* and *LM* are *Moderate* impactful smell types as their $0.3 \leq r_{fp} < 0.5$. So developers should be careful about these smell types with higher priority during on the fault-related maintenance activities. It has also been observed that the correlation coefficient is significant for the five code smell types where $p - value < 0.05$ considering 5% level of significance. However, all of these smell types have strong correlation with the fault-proneness ($r_{fp} \geq 0.5$) except *LzC*.

6.3.2.3 Smell frequency versus correlation coefficient.

From Table 6.6, we have found an interesting observation that *Anti Singleton, Blob, Class Data Should Be Private* and *Long Parameter List* have strong correlation, that is, high relationship with the fault-proneness while their average smell frequencies (*ASF*s) are not very high except for *Long Parameter List*. It is possible that, as like changes, faults occur more in method levels where the number of parameters is higher, and therefore its fault-proneness gets higher in

accordance with its frequency. On the other hand, medium (moderate) correlated smell types such as *Lazy Class* and *Long Method* have moderate and high smell frequency respectively. *Complex Class* and *Large Class* smell types have low correlation but high smell frequency. In addition, low correlated smell types such as *BCSBA, MFABNC, RPB, SC* and *SG* have low smell frequency, and as it is expected that lower frequent smell types have lower fault-proneness because of their lower number of classes in the systems.

Therefore, for the research question, we have focused on the high and moderate correlated code smell types, and observed the general perception that higher frequent smell types might have higher correlation with the fault-proneness. This statement is not true, rather different levels of smell frequencies have different levels of correlation strengths. However, *in many cases while considering high and moderate correlated code smell types, smell frequencies are almost opposite of their relationship strength.*

From the frequency graphs in Figure 6.4, we have observed that higher frequent smell types have higher fault-proneness whereas their correlations do not follow it. To make it clear, the correlations show how the fault-proneness of a smell type will increase (or decrease) when the smell frequency increases (or decreases). On the other hand, frequency just indicates their frequency of occurrences in the systems.

Summary for RQ2. While considering each code smell type individually, most of the smell types are correlated with the fault-proneness except for the last four smell types. Among these *AS, LPL, Blob* and *CDSBP* are highly correlated smell types. Moreover, smell frequencies have different impact levels on the fault-proneness such as high and moderate correlated smell types have moderate and high smell frequencies respectively.

6.3.3 Impact on both Change- and Fault-proneness of Each Code Smell Type and its Prioritization [RQ3]

Before answering RQ3 of prioritizing code smell types, we first try to understand whether code smell types have more impact on change-proneness than fault-proneness or vice-versa. To do this, we perform the following hypothesis testing by comparing the correlation coefficient between change-proneness and fault-proneness with each of the smell types. Here we use statistical Z-test for the hypothesis testing [177].

- *Null hypothesis (H_0): There is no difference between the correlation coefficient of change- and fault-proneness with each of the code smell types. That is, for each code smell type, $r_{cp} = r_{fp}$.*
- *Alternative hypothesis (H_1): There is a difference between the correlation coefficient of change- and fault-proneness with each of the code smell types. That is, for each code smell type, $r_{cp} \neq r_{fp}$.*

The result of the hypotheses testing is shown in the fourth column of Table 6.7 as the p-value. The second and third columns again show the correlation coefficient for the two metrics – change- and fault-proneness for the readability purpose. It is noted that we have excluded the last four smell types from the testing as these coefficients contain *NA* (not applicable). However, the p-value is greater than 0.05 (considering 5% level of significance) for all the smell types which indicates that the null hypothesis H_0 can not be rejected. That is, the correlation does not differ from change-proneness to fault-proneness for each of the code smell types.

From the above analysis, we can see that impact of each code smell type on change- and fault-proneness are not different. So for this study, we combine the two correlation scores considering their same weight (i.e., average) for each of the smell types using the following Equation 6.4. This is because we put on equal

Table 6.7: Impact of code smell types on both change- and fault-proneness

Code Smell	**Correlation (CP),** r_{cp}	**Correlation (FP),** r_{fp}	**p-value (Hypothesis test)**	**Average Correlation,** r
LPL	0.61	0.71	0.245	**0.65**
AS	0.57	0.70	0.319	**0.64**
CDSBP	0.57	0.64	0.166	**0.54**
Blob	0.48	0.51	0.591	**0.51**
LzC	0.40	0.46	0.692	*0.47*
LM	0.39	0.35	0.548	*0.38*
CC	0.38	0.28	0.520	*0.30*
LC	0.28	0.24	0.564	0.26
BCSBA	0.11	0.21	0.330	0.19
MFABNC	0 (NA)	0 (NA)	NA	0 (NA)
RPB	0 (NA)	0 (NA)	NA	0 (NA)
SC	0 (NA)	0 (NA)	NA	0 (NA)
SG	0 (NA)	0 (NA)	NA	0 (NA)

[* Bold, italic and plain font in Average Correlation column indicates *high, moderate* and *low* impact respectively]

emphasis on these two maintainability metrics as maintenance activities mostly depend on faults and changes [3].

$$\textit{Average Correlation, } r = \frac{r_{cp} + r_{fp}}{2} \tag{6.4}$$

The result of the average correlation (r) scores is shown in the last column of Table 6.7. These r scores actually represent the total impact of code smell types on both metrics combinedly. So, the higher scores of the r are regarded as higher prioritized code smell types and lower scores represent the lower prioritized ones. Therefore, Table 6.7 shows the r scores in descending sorted order where top-to-bottom represent high-to-low prioritized smell types. Moreover, based on the r scores, we categorized the smell types as the following three prioritized (or impact) levels as discussed in Sub-section 6.2.7.

1. *High prioritized smells ($r \geq 0.5$):* Long Parameter List, Anti Singleton, Class Data Should Be Private and Blob. Mostly these smell types are related to data.

2. *Moderate prioritized smells ($0.3 \leq r < 0.5$):* Lazy Class, Long Method and

Complex Class. Mostly these smell types are related to blocks of codes like methods and classes.

3. *Low prioritized smells ($r < 0.3$):* Large Class, Base Class Should Be Abstract, Many Field Attributes But Not Complex, Refused Parent Bequest, Spaghetti Code and Speculative Generality.

Summary for RQ3. Different code smell types have different impacts on the two maintainability metrics – change- and fault-proneness combinedly. Therefore, Table 6.7 helps developers about which smell types they should focus on a priority basis in their maintenance as well as development activities.

6.3.4 Comparison with state-of-the-art approach.

Palomba et al. [10] investigated the impact of code smells on the two maintainability metrics – change- and fault-proneness based on the smell introduction and removal approach. In this approach, they compared change- and fault-proneness when a smell instance has been introduced with when it has been removed. They particularly showed that almost each of the smell types has limited, that is, low impact on fault-proneness while removing these smells from the systems. They interpreted it as faulty classes that will still be fault-prone in the future even if a smell was removed. Therefore, according to this finding, fault-proneness is smell independent. On the other hand, most of the smell types have high impact on change-proneness based on their removal impact. Considering all code smell types as a whole, they also showed that smelly classes are more change and fault-prone than non-smelly classes.

The approach discussed in this chapter is different from their approach which is based on correlation analysis of the frequency of code smell types, change- and fault-proneness. Our approach supports the findings of Palomba et al. that different smell types have different impact levels such as high, moderate and low

impact on these two metrics – fault- and change-proneness. However, it is interesting that, if we observe Table 6.7 carefully, we can see that most of the code smell types (six out of nine) have a higher correlation with the fault-proneness than with the change-proneness. It contrasts the finding of Palomba et al. that smell has no relationship with fault-proneness. Therefore, developers and researchers should focus on the code smells while working with both faults and changes.

In addition, according to Palomba et al. *CC, LM, SC, SG,* etc. are high impactful smell types which are different from our correlation based analysis (not high). On the other hand, according to our finding, *CDSBP, LPL,* etc. are high impactful smell types which are low based on Palomba et al. Moreover, expert developers' opinions support the impact variations of different code smell types on software systems (Table 7.3 of Chapter 7). However, according to their findings, *CC, SC, SG,* etc. are also not the high impactful smell types, rather *CDSBP, LC, LM,*, etc. are the high impactful ones. These findings including developers' perceptions make the thesis different from the existing ones which provide a new dimension to rethink about code smell impact on the change- and fault-proneness for the practitioners to maintain software systems and researchers to innovate smell-refactoring tools.

6.4 Threats To Validity

The construct, external and internal threats to validity of this empirical study are discussed in this section.

1. Threats to construct validity

As the threats concern the relationship between theory and observation due to measurement issues, there are two main considerations. Firstly, the potential threats are related to the accuracy of the tools used in this study to detect code smells automatically from software systems. The results of the tool might lead to a

different phenomenon for different tools than our investigation. However, we have used the tool DÉCOR which has been evaluated in previous studies having good accuracy. Additionally, we have manually verified the smells to avoid false positive results, and hence these might mitigate the threats to some extent. Secondly, the computation of the two measures – change- and fault-proneness might be influenced by these threats. However, we have mitigated it by implementing the computation techniques which have been used by previous studies.

2. Threats to external validity

Regarding the generalization of our findings, we have limited our study based on only Java systems, due to limitations of the infrastructure we have used (e.g., for a large number of code smells, the detection tool only works on Java systems). Moreover, we have focused on open-source systems only, and we cannot speculate about how the results would be different for industrial systems. Since the frequencies of code smells between open-source and industrial systems are not different as observed by Rahman et al. [2], these results might be generalized to some extent.

3. Threats to internal validity.

The threats concern factors that could influence our findings. We are aware that we cannot claim a direct cause-effect relationship between the occurrence of code smells and the two maintainability metrics – change- and fault-proneness. In particular, our findings may be influenced by the different factors related to software development processes such as the experience of developers, workload, etc. However, we have limited our focus on the software systems rather than development processes. Considering process related metrics is part of our future research agenda.

6.5 Discussion

The results clearly show that code smells have negative impacts on software maintainability with respect to change- and fault-proneness. Not all of the smell types have a similar impact, and certain types of smell such as *Anti Singleton, Blob, Class Data Should Be Private* and *Long Parameter List* have high priority due to their high impact on these two metrics. However, the shortlist of the most impactful set of smell types can aware developers to write or reuse source code carefully without inducing these smells from the beginning during software development. Moreover, these can help them to prioritize the smells to be refactored based on their impact while performing maintenance activities. Also, our findings help them to develop refactoring tools based on their impact.

In Summary, we recommend and highlight the following implications of the findings:

1. Code smells have different impacts on the two metrics – change- and fault-proneness based on their types. However, there has been almost similar level of impact on both of these two metrics for each code smell type. Therefore, developers should be careful about the smell impact while minimizing both faults and changes to maintain a software system.

2. From the frequency analysis, it is observed that high and moderate frequent code smell types have almost opposite impact on these two metrics. This observation has excluded the low frequent smell types because of their lower frequencies, change- and fault-proneness in the systems. Therefore, smell frequencies do not align with the smell impact levels and developers and researchers should focus more on the impactful smell types than frequent ones.

3. The *ISF* metric adds another dimension in the frequency analysis with

more significant information that helps developers and researchers to understand the criticality of a code smell type with respect to change- and fault-proneness.

6.6 Summary

In summary, this chapter highlights the varying impacts of different code smells on software maintainability, particularly their influence on change- and fault-proneness. Not all code smells have the same impact, with certain ones – such as *Anti Singleton, Blob, Class Data Should Be Private,* and *Long Parameter List* – proving especially significant in these areas. These findings offer developers valuable guidance for writing cleaner code from the beginning and help prioritize refactoring efforts where they will have the most impact. This study not only deepens understanding but also helps inform the development of refactoring tools that target the most critical code smells, ultimately leading to improved software quality and maintainability.

CHAPTER 7

PRIORITIZATION OF CODE SMELLS

Code smell prioritization involves determining the order in which the smells should be refactored to improve software quality. Since there exists various types of code smells, prioritization of these becomes essential to enhance software maintainability. In this chapter, code smells are prioritized according to the impact of the smells on the software quality and maintainability metrics as discussed in Chapter 5 and Chapter 6 respectively. By integrating both categories of impact – quality and maintainability – this chapter discusses a structured approach to prioritize these smells. The prioritization is further refined by incorporating developers' perceptions, comparing their subjective views with the empirical results derived from the metrics. This comparison will highlight the extent to which developers' perceptions align with the empirical analysis or reveal any discrepancies, providing insights into how intuitive judgments compare to data-driven results.

7.1 Introduction

Code smell prioritization is a crucial process that involves determining the order in which different code smells should be refactored to enhance the maintainability and overall quality of a software system. Since there is a vast number of code smells, each with varying levels of impact on the system, it becomes essential to distinguish between high, moderate and low priority smell types. Not all code smells carry the same weight; some may cause more significant issues, such as making the source code harder to maintain or understand, while others may have a more negligible effect.

Prioritizing the code smells helps developers to focus their efforts on the most critical areas, ensuring that the most harmful or disruptive issues are addressed first. Thus, developers can prevent potential long-term problems that could arise from neglecting severe code smells. Therefore, code smell prioritization plays a vital role in improving software maintainability and quality over time.

In this chapter, we will focus on the process of prioritizing code smells by considering the findings from Chapter 5, where the impact of code smells on software quality is analyzed, and Chapter 6, which discusses the impact of the smells on maintainability metrics. These combined analyses will provide a comprehensive framework for deciding which code smells should be refactored with the highest priority, thus ensuring the continuous improvement of the software quality and maintainability throughout its life cycle.

Furthermore, the prioritization process is enhanced by integrating developers' perceptions, allowing for a comparison between their subjective views and the empirical results obtained from quantitative metrics. This comparison not only emphasizes the degree to which developers' perceptions align with the empirical analysis but also uncovers any notable discrepancies between intuitive judgments

and data-driven outcomes. By examining these relationships, we can achieve valuable insights into the decision-making processes of developers and the potential biases that may influence their perceptions of code smells. Understanding these dynamics is crucial, as it helps to bridge the gap between qualitative assessments and quantitative measures, ultimately guiding developers towards more informed refactoring decisions. This comprehensive approach ensures that both subjective insights and objective data are taken into account, fostering a more holistic understanding of code maintainability and quality improvement efforts.

The particular contributions of this chapter are as follows:

(i) The prioritized list of code smell types that can be used to improve software quality and maintainability.

(ii) The perception of expert developers about the impact on software systems and prioritization of code smells.

(iii) The comparison of the developers' perception with the findings of the metric based system analysis to reveal how they are far behind from the empirical analysis.

Structure of the chapter: Section 7.2 describes prioritization process of code smells based on software quality and maintainability metrics. Section 7.3 discusses the developers' perception about impact of code smells and the prioritization process based on the perceptions. Section 7.4 compares the two prioritization results, and finally, Section 7.6 summarizes the chapter describing the key findings.

7.2 Prioritization of Code Smells Based on Software Quality and Maintainability Metrics

This section focuses on the prioritization of code smells using impact scores derived from two main metric categories. The first set of scores is calculated from software metrics, while the second set is based on maintainability metrics. The overall prioritization is determined by averaging these two impact scores, treating both equally, to produce a *Priority Score* for each code smell type. This scoring system, with its normalized range, enables the classification of code smells into high, moderate, and low priority groups, facilitating more effective maintenance and optimization strategies.

To prioritize code smells, firstly, impact scores (R_1) based on the software metrics from Table 5.9 of Chapter 5 have been used. Basically, these are the average scores of the square of the correlation coefficients (r^2) between the code smell frequency and each of the 25 software quality metrics. Secondly, impact scores (R_2) based on the maintainability metrics from Table 6.7 of Chapter 6 have also been used. Similarly, these are the average scores of the square of the correlation coefficients (r^2) between the code smell frequency and each of the two maintainability metrics – change- and fault-proneness. The r^2 score, which is formally known as coefficient of determination, has been chosen because it allows us to consider both the positive and negative impacts of the metrics. Also, the range of the score is normalized into [0, 1] which is useful for further calculation and understanding of their impacts. Thirdly and finally, the overall impact score is calculated using the simple average of the two impact scores R_1 and R_2 for each code smell type. Here, both datasets are treated as equal importance without favoring one over the other. In the study of this chapter, the overall impact score is referred to as the *Priority Score (P)* of the code smell types which is measured using Equation 7.1. These impact results including the *Priority Scores, P* are

shown in Table 7.1.

$$\textit{Priority Score, } P = W_1 \times R_1 + W_2 \times R_2 \tag{7.1}$$

Here, W_1 and W_2 are the wights for R_1 and R_2 respectively, where we assume equal weights, that is, $W_1 = W_2 = 0.5$.

Table 7.1: Prioritization of code smells based on both software and maintainability metrics

Code Smell	**Impact Score,** R_1	**Impact Score,** R_2	**Priority Score,** P
AS	0.85	0.42	**0.63**
LPL	0.83	0.43	**0.63**
CDSBP	0.77	0.29	**0.53**
Blob	0.79	0.28	**0.53**
LM	0.86	0.14	*0.50*
CC	0.85	0.09	*0.47*
LC	0.83	0.07	*0.45*
RPB	0.82	0 (NA)	*0.41*
SC	0.82	0 (NA)	*0.41*
SG	0.80	0 (NA)	0.40
MFABNC	0.76	0 (NA)	0.38
BCSBA	0.59	0.04	0.32
LzC	0.26	0.22	0.24

[* Bold, italic and plain font in Average Impact Score (R) column indicates *high, moderate* and *low* prioritized code smells respectively]

The P scores are used to prioritize the code smell types. That is, the high score of the P indicates the high impactful code smell type, and hence it is a high prioritized one. Similarly, the low score of the P indicates the low impactful code smell type, and hence it is a low prioritized one. Based on the statistical quartile analysis (Q_1, Q_2, Q_3) of the P scores (similar process as in Chapter 5), the code smell types are grouped into the following three prioritization levels. Table 7.1 shows the code smell types as the priority order where top one is the most prioritized and bottom one is the least prioritized code smell type.

1. *High prioritized code smells,* $P \geq 0.53(Q_3)$*:* Anti Singleton, Long Parame-

ter List, Class Data Should Be Private and Blob.

2. *Moderate prioritized code smells,* $0.40(Q_1) < P < 0.53(Q_3)$*:* Long Method, Complex Class, Large Class, Refused Parent Bequest and Spaghetti Code.

3. *Low prioritized code smells,* $P \leq 0.40(Q_1)$*:* Speculative Generality, Many Field Attributes But Not Complex, Base Class Should Be Abstract and Lazy Class.

In a nutshell, the approach of prioritizing code smells by combining software and maintainability metrics offers a comprehensive method for managing code quality. By calculating the *Priority Score (P)* through the equal weighting of *R1* and *R2* impact scores, this method ensures that both the software's structural integrity and its maintainability are considered. Grouping code smells into *high, moderate,* and *low* priority categories allows developers to focus on the most impactful issues first, promoting more efficient resource allocation and long-term improvement in software quality. This approach assists them with a data-driven framework to prioritize refactoring efforts for improving maintainability, and reducing change- and fault-proneness.

7.3 Developers' Perception about Impact of Code Smells on Software

Understanding developers' perception about the impact of code smell types is significant due to their active involvement in software development and maintenance. Therefore, the objective of this section is to investigate experienced developers' perceptions regarding the impact of code smell types on software systems and to uncover the reasons behind their judgments of certain code smell types as impactful while the others are not. This section particularly answer the following two research questions.

RQ1: ***What do developers perceive about the impact of code smell types on software systems?***

RQ2: ***How are the code smell types prioritized based on the developers' perception?***

7.3.1 Data Analysis Process about Developers' Perception

To identify developers' perception about code smell's impact (or prioritization) and compare it with the findings of the metric-based correlation analysis, we have taken opinions from 55 experienced developers from various software companies in 10 different countries. We call them expert developers or only experts in this study.

We have restricted the developers' experience above 3 years and they must know about the code smells so that we can collect insightful information from them for the analysis. More specifically, most of them have 5 to 25 years of experience in software development, whereas only two developers have three years and another two have four years of experience. They have also worked on a number of software projects ranging from 3 to 120. The details about the data from the developers are given in the repository[6].

We have asked them to provide the impact scores or levels (from 1 to 3 meaning – 1 means *High*, 2 means *Moderate* and 3 means *Low*) on the 13 code smell types based on their experiences on software maintenance activities such as – solving faults, changing codes or features, adding new features to existing software, comprehending programs, etc. These impact levels are also considered as the prioritization levels of the smell types. Moreover, we have also collected the reasoning behind the impact score (from 1 meaning highest to 13 meaning lowest impact) for each code smell type based on their intuition from three experienced developers. These three developers have experience: 41 years and 100 projects

(Expert-1), 10 years and 15 projects (Expert-2), and 8 years and 11 projects (Expert-3) in software development and maintenance respectively, particularly in Java programming language. The results of the data analysis are shown in the next sub-section.

7.3.2 Result Analysis

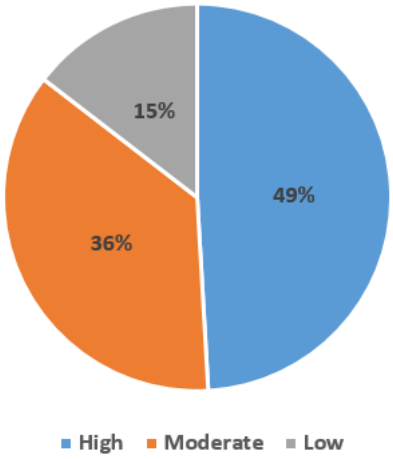
The detailed results about developers' perception about the impact of code smell types (RQ1) and their prioritization (RQ2) are explained in this section.

7.3.2.1 Developers' Perception about Code Smells [RQ1]

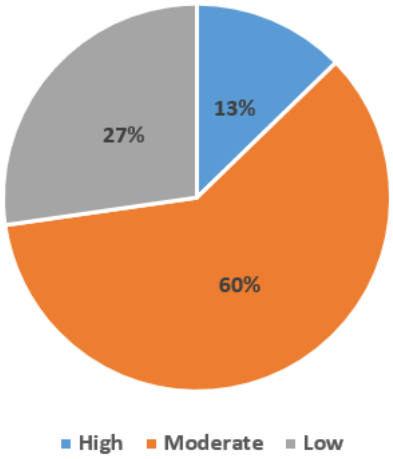
Developers' perceptions regarding the impact of code smells play a critical role in how they prioritize their refactoring efforts to enhance system maintainability. Insights gathered from 55 expert developers, as detailed in the Data Analysis Sub-section 7.3.1, provide valuable clarity on these perceptions. The developers' assessments of the impact levels of various code smell types are illustrated through pie charts in Figure 7.1.

For instance, with regard to the *Anti Singleton (AS)* code smell type, as depicted in Figure 7.1a, 49% of the developers believe it to be a highly impactful code smell. In contrast, 36% view it as moderately impactful, while a smaller portion, only 15%, consider it to have low impact. From this data, we can conclude that *Anti Singleton* is predominantly regarded as a high-impact code smell type.

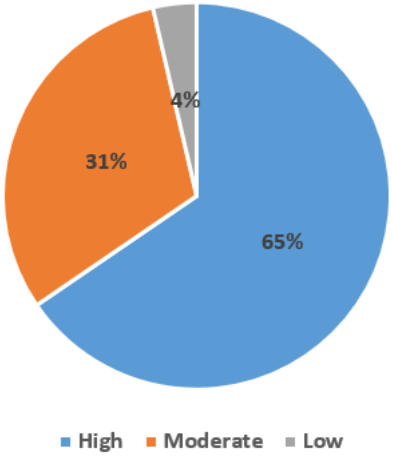
Thus, by aggregating the voting results from the developers, the 13 different types of code smells can be classified into the following three distinct impact levels based on their perceived severity:



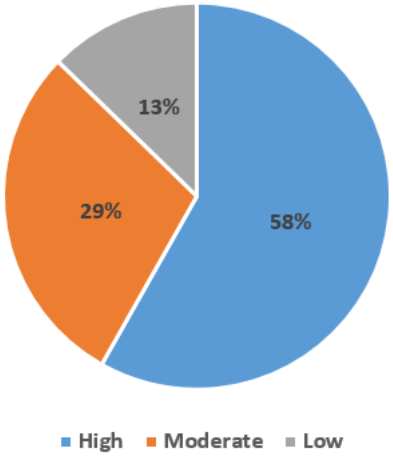
(a) Anti Singleton (AS)



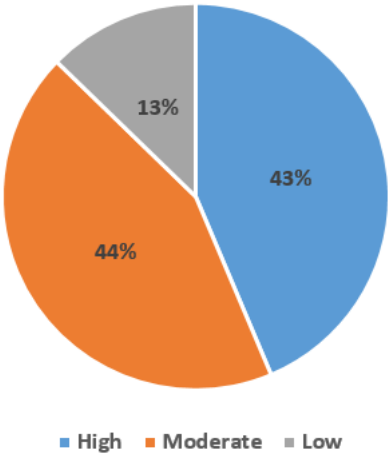
(b) Base Class Should Be Abstract (BCSBA)



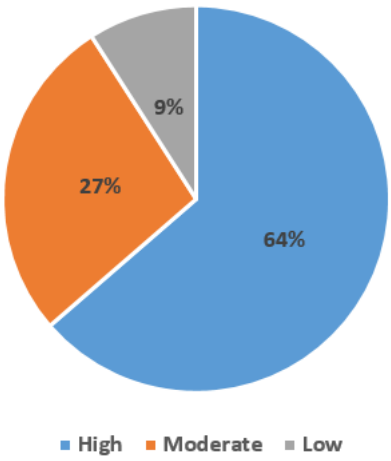
(c) Blob



(d) Class Data Should Be Private

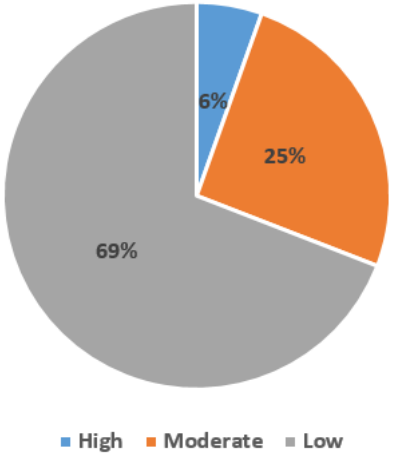


(e) Complex Class (CC)

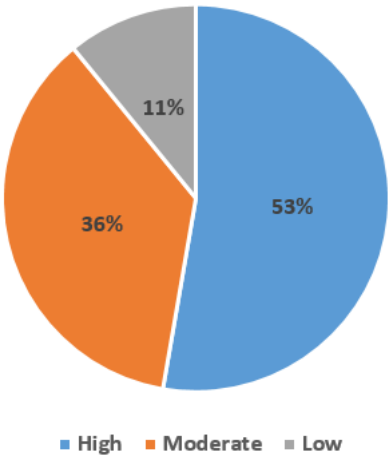


(f) God Class (GC)

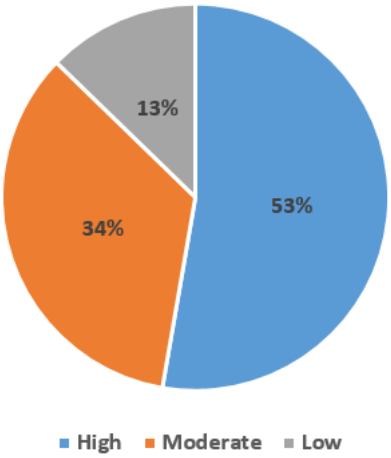
Figure 7.1: Developers' Perception about Code Smell Impact



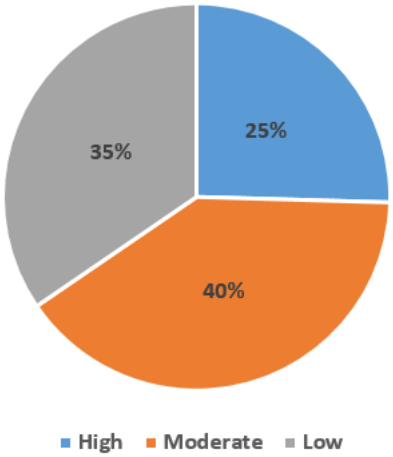
(g) Lazy Class (LzC)



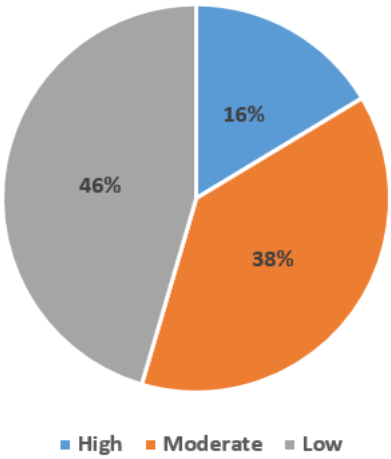
(h) Long Method (LM)



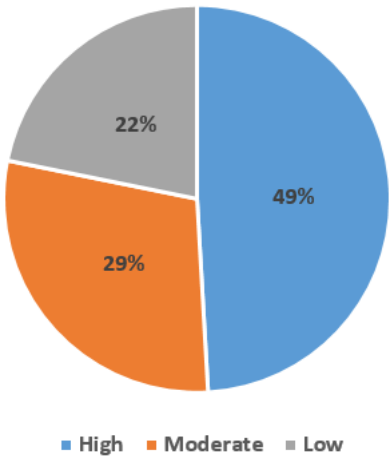
(i) Long Parameter List (LPL)



(j) Many Field Attributes But Not Complex (MFABNC)

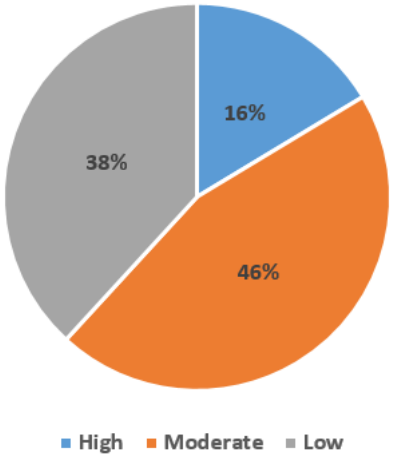


(k) Refused Parent Bequest (RPB)



(l) Spaghetti Code (SC)

Figure 7.1: Developers' Perception about Code Smell Impact (cont.)



(m) Speculative Generality (SG)

Figure 7.1: Developers' Perception about Code Smell Impact (cont.)

1. *High impactful code smells*: Seven types of the code smells fall into this group. These are – Anti Singleton, Blob, Class Data Should Be Private, God Class, Long Method, Long Parameter List and Spaghetti Code.

2. *Moderate impactful code smells*: Four types of the code smells fall into this group. These are – Base Class Should Be Abstract, Complex Class, Many Field Attribute But Not Complex and Speculative Generality.

3. *Low impactful code smells*: Only two code smell types fall into this group. These are – Lazy Class and Refused Parent Bequest.

This classification reflects what majority of the expert developers thinks about code smells' impact on the software maintainability.

7.3.2.2 Developers' Perception: Impact and Prioritization of Code Smells [RQ2]

The classification presented in the previous sub-section primarily reflects the majority opinion of expert developers. However, this approach does not account for

the full spectrum of individual opinions, as it overlooks the perspectives of every developer when assigning impact scores to various code smell types. As a result, the classification may miss subtle but important nuances in how different developers perceive the impact of each code smell.

To address this limitation, a more refined and comprehensive classification is introduced in this sub-section. Unlike the previous method, this approach takes into account the opinions of all developers, ensuring that each voice is represented when determining the impact level of every code smell type.

According to the 55 expert developers as discussed in the Data Analysis Subsection 7.3.1, the impact scores for each of the code smell types have been shown in Table 7.2. Here, low(1)-moderate(2)-high(3) score (which individual developer provides score to a code smell type) means the high-moderate-low impact of the smell types respectively. That is, 1 score from developer means that it is a high impactful code smell type, 2 means moderate and 3 means low impactful smell type. We have calculated the *Sum of Dev Score* (second column of the table) by summing up the individual developer's score for each code smell type and then calculated the *Average Dev Score* (third column). The lower of these metrics' score means higher impact of the smell type. Therefore, we have introduced a new and normalized metric *Inverse of Dev Score, IDS* (last column) which ranges from 0 to 1 making it easier to interpret. The *IDS* for each code smell type is calculated using Equation 7.2. The higher the score of the metric means the higher the impact of the smell type.

$$\textit{Inverse of Dev Score, IDS} = \frac{1}{\textit{Average Dev Score}} \tag{7.2}$$

The *Inverse of Dev Score (IDS)* metric is used to prioritize the code smell types, where the high-to-low score of the metric indicates high-to-low prioritization of the smell types. Therefore, Table 7.2 shows the impact of the code smell types

Table 7.2: Impact Score of code smell types according to developers' perception

Code Smell	Sum of Dev Score	Average Dev Score	Inverse of Dev Score, IDS
Blob	76	1.38	**0.72**
LC	80	1.45	**0.69**
CDSBP	85	1.55	**0.65**
LM	87	1.58	**0.63**
LPL	88	1.60	**0.63**
AS	91	1.65	*0.60*
CC	93	1.69	*0.59*
SC	95	1.73	*0.58*
MFABNC	115	2.09	*0.48*
BCSBA	118	2.15	0.47
SG	122	2.22	0.45
RPB	126	2.29	0.44
LzC	145	2.64	0.38

[* Bold, italic and plain font in Average Impact Score column indicates *high, moderate* and *low* impact respectively]

as a prioritized order according to their perceptions, which helps them especially new developers in their refactoring activities. Moreover, based on the statistical quartile analysis (Q_1, Q_2, Q_3) of the metric, we have categorized the smell types into the following three prioritized (or impact) levels.

1. *High prioritized code smells,* $IDS \geq 0.63(Q_3)$*:* Blob, Large Class, Class Data Should Be Private, Long Method and Long Parameter List.

2. *Moderate prioritized code smells,* $0.47(Q_1) < IDS < 0.63(Q_3)$*:* Anti Singleton, Complex Class, Spaghetti Code and Many Field Attributes But Not Complex.

3. *Low prioritized code smells,* $IDS \leq 0.47(Q_1)$*:* Base Class Should Be Abstract, Speculative Generality, Refused Parent Bequest and Lazy Class.

The new classification based on the *Inverse of Dev Score (IDS)* metric provides a more balanced and inclusive view, which ultimately leads to a more accurate reflection of the diverse perceptions within the developer community. This allows

for a more refined prioritization of refactoring efforts, ensuring that no significant viewpoint is overlooked.

Additionally, among the 55 developers, three developers have provided justification behind each impact score for each code smell type as shown in Table B.1, B.2 and B.3 in the Appendix section. To the best of our knowledge, we are the first to collect the detailed viewpoints with justification from the experts. For a better understanding of the smell impact, their justification might be helpful for the developers, especially for the new developers in software industries.

Summary for RQ1 and RQ2. Developers think code smells have different impacts based on their types. Size related smell types such as *Blob, Large Class, Long Method*, etc. have higher impact than complexity and data related smell types such as *Anti Singleton, Complex Class, Spaghetti Code*, etc. These perceptions help the developers especially new developers in their refactirng activities on priority basis.

7.4 Comparison of Code Smell Prioritizations

The findings of the previous section will actually provide insights about the current thoughts of developers and help to compare whether their thoughts are close to the results from the metric-based system analysis. So, we separately discuss this comparison in this section through the following research question (RQ3).

RQ3: ***What are the differences in code smell prioritization between the empirical analysis of the systems and perception of developers?***

This research question examines whether software developers are on the right track in their efforts to improve software maintainability by addressing code smells. It specifically aims to identify which code smell types developers should prioritize and which ones may be less significant. By conducting a comparative analysis, this research question provides valuable insights for both researchers and practitioners.

The findings can be used to enhance automated tools for code smell detection and refactoring prioritization, taking into account the analysis results and developers' preferences.

We have compared the prioritization results observed from the analysis of software systems (*Priority Score, P*) of Table 7.1 in Section 7.2 with the developers' perceptions (*Inverse of Dev Score, IDS*) of Table 7.2 in Sub-section 7.3.2.2. The comparative results have been shown in Table 7.3.

Table 7.3: Compare regarding code smell prioritization between metric-based findings and developers' perceptions

		Finding from System Analysis	**Developers' Perception**
Impact Level	**High**	AS, LPL, CDSBP, Blob	Blob, LC, CDSBP, LM, LPL
	Moderate	LM, CC, LC, RPB, SC	AS, CC, SC, MFABNC
	Low	SG, MFABNC, BCSBA, LzC	BCSBA, SG, RPB, LzC

From the table, we have identified the following findings:

1. For each of the three impact levels, we have calculated a metric – *Matched Score* using equation 7.3, which means that how many smell types found from developers' perception of an impact level are matched with the system analysis. According to the metric, three out of five *High* impactful smell types – *Blob*, *CDSBP* and *LPL* found from the developers' perception have been matched with the metric-based system analysis. Hence, the *matched score* for this level is $(3/5) * 100\% = 60\%$. Similarly, for the *Moderate*

impact level, smell type *CC* and *SC* have been matched and the score is $(2/4) * 100\% = 50\%$. And for the *Low* impact level, three out of four smell types – *BCSBA*, *RPB* and *LzC* have been matched and so the score is $(3/4) * 100\% = 75\%$. Overall, combining all impact levels the matched score is $(8/13) * 100\% \approx 61.54\%$. All cases, at least 50% of developers' perception has matched with the metric-based system analysis which shows the significance of the empirical analysis. However, from this comparative analysis, it is observed that developers' perception differs a lot from system analysis regarding the code smell's impact.

$$Matched\ Score = \frac{\mid D \cap S \mid}{\mid D \mid} \times 100\% \tag{7.3}$$

Here, D = a set of smell types identified from developers' perception.
S = a set of smell types identified from metric-based system analysis.

2. The two high prioritized smell types based on developers' perception – *LC* and *LM* differ from the findings of system analysis. According to the developers' justification, it is hard to comprehend and maintain projects due to the existence of *LC* and *LM* smells (see Table B.1, B.2 and B.3). Thus these smells decrease the productivity of the projects and so should get high priority. On the contrary, according to the system analysis, *LC* and *LM* moderately impacts on the system quality.

3. The two moderate prioritized smell types based on developers' perception – *AS* and *MFABNC* differ from the findings of system analysis. According to the system analysis, the data related smell types – *AS* can cause more degradation in code quality and thus get higher priority. Also, according to the developers, *AS* can increase coupling and cause error-prone codes (see Table B.1, B.2 and B.3). On the other hand, another data related smell type *MFABNC* is less harmful for the system though it decreases cohesion based

on developers' perceptions. Therefore, developers should be carefully think about these two smell types – which one should get higher priority. Since, *AS* is more related with systems quality and maintainability than *MFABNC*, they should give high priority on *AS* and less priority on *MFABNC*.

4. Only *RPB* is a low prioritized smell type based on the developer perception, while it is a moderate one according to the system analysis. It causes due to improper class hierarchy and increases maintenance effort. Thus it should be handled carefully and considered as a *moderate* prioritized code smell type.

5. It is observed that developers focus on the size related smell types (*High*) such as *Blob, LC* and *LM* as these increase complexity and decrease comprehensibility of the codes. They also focus on the data or variable related smell types (*High*) such as *CDSBP* and *LPL* as these in practical have more impact on the software quality and maintainability metrics based on the system analysis.

Summary for RQ3. For most of the code smell types (61.54%), developers' perception about smell prioritization matches with the system analysis. Whereas, 38.46% of the smell types of developers' perception does not match with the system analysis which is not a small portion. Therefore, developers should rethink their perception about smell prioritization while improving software quality as well as minimizing faults and changes in the systems.

7.5 Threats to validity

In this study, we have conducted a comparative analysis of empirical results alongside the perspectives of 55 expert software developers. Since individual viewpoints can vary from one developer to another based on their personal experiences, expertise, and working environments, this variation introduces a potential internal

validity threat to our findings. To mitigate this risk and enhance the credibility of our results, we carefully selected developers from a diverse range of software companies spanning ten different countries. By incorporating perspectives from professionals working in varied cultural, technological, and organizational settings, we aimed to ensure a more comprehensive and balanced representation of expert opinions, thereby strengthening the overall reliability and validity of our study.

7.6 Summary

The results of this chapter clearly show that code smells have different levels of impact on software systems. Based on the impact on the software quality and maintainability metrics, these code smells are prioritized into three levels – *High, Moderate* and *Low*. From these levels, *Anti Singleton, Long Parameter List, Class Data Should Be Private* and *Blob* are the most prioritized smell types. However, from developers' perspective, *Blob, Large Class, Class Data Should Be Private, Long Method* and *Long Parameter List* are the most significant ones. The comparative analysis with expert developers' perceptions makes this thesis significant in the sense that practitioners and researchers need to rethink about the smell impact. Since developers perceptions vary from the empirical results they should be more careful while prioritizing refactoring of the code smells. Also, these findings help them to develop refactoring tools based on their impact.

CHAPTER 8

IMPACT OF CODE SMELLS ON SOFTWARE MAINTENANCE COST

Program comprehensibility is a key factor in software maintainability. A clear understanding of a software system is essential for effective maintenance, reducing the risk of introducing defects and lowering overall costs. Conversely, the presence of code smells are design flaws that hinder program understanding, complicate the maintenance process and increase maintenance costs. The study in this chapter investigates the relationship between 13 different types of code smells and their impact on program comprehensibility by analyzing 35 open-source Java systems. The research aims to identify which code smells have the most detrimental effect on comprehensibility. Through two research questions, the study categorizes code smells based on their comprehensibility levels and analyzes the relationship be-

tween the impact of code smells and program comprehensibility. The findings will help developers to prioritize refactoring efforts to improve software quality and reduce maintenance costs by focusing on the most impactful code smells.

8.1 Introduction

Program comprehension refers to the process of understanding an existing software system in order to plan, design, code, and test modifications [3]. It is a significant activity that consumes approximately more than 50% of the total effort expended throughout the life cycle of a software system [187, 11, 3, 12]. Moreover, it is found that, on average, 70% of their time is spent on program comprehension activities [188, 189]. Without a thorough understanding of the system, making effective changes would be difficult, and the overall maintenance cost could rise. Thus, a good understanding of the system not only improves its quality but also enables efficient maintenance at a reduced cost [190].

The importance of program comprehension becomes even more evident during maintenance, which typically involves modifying parts of the system to fix bugs, improve functionality, or adapt to new requirements. Without a complete and accurate understanding of the system's architecture, components, and interactions, any changes made are likely to introduce new bugs, degrade system quality, or cause reliability issues [3]. In contrast, a deep understanding of the software allows developers to make targeted and effective modifications that preserve the integrity and efficiency of the system. This, in turn, leads to better-managed projects, and lower maintenance costs that can effectively meet changing requirements over time.

On the other hand, code smells refer to design flaws in a program that make it harder to comprehend and maintain [1]. These smells are generally recognized as patterns in the program that reduce comprehensibility and increase the likeli-

hood of errors [30]. If these issues are not addressed through refactoring early on, future changes can introduce more defects and increase the maintenance cost [3]. Therefore, code smells are a barrier to program comprehension, and there exists a negative relationship between them. Since there exist a large number of code smells, all of these do not have same relationship with program comprehensibility. In this study, we identify which code smells have more impact on program comprehensibility. Thus, the findings will help developers to refactor those impactful smells on priority basis to reduce maintenance cost of a software.

In summary, this chapter makes the following contributions:

(i) Measurement of program comprehensibility metric for each of the 13 code smell types.

(ii) Identification of the relationship between impact of the code smells and program comprehensibility.

Structure of the chapter: Section 8.2 describes the design of the empirical study. Section 8.3 presents and discusses the results of the study. Finally, Section 8.4 summarizes the chapter.

8.2 Empirical Study Design

The objective of the empirical study is to identify the relationship between the impact of the 13 code smell types and program comprehensibility. For this study, we, at first, identify how much smelly classes are comprehensible based on their types by analyzing 35 open-source Java systems. Second, we analyze the relationship between them. More specifically, to achieve this goal, we answer the following research questions:

RQ1: ***How much smelly classes are comprehensible based on their types?***

This research question identifies and quantifies the comprehensibility metric for each code smell type by analyzing the corresponding smelly classes of the software systems. Code smell types which have a lower comprehensibility score are considered as low 'comprehensible' for the smelly classes. Based on the score, the code smell types are categorized into three comprehensibility levels – *High, Moderate* and *Low*. The answer to RQ1 will help software developers to focus on the smell types which have a low comprehensibility score to make a software system more comprehensible and maintainable.

RQ2: ***What is the relationship between impact of code smell types and program comprehensibility?***

The research question identifies the relationship between the impact of code smell types and program comprehensibility. Code smell types which have higher impact on software systems might be lower comprehensible. This types of code smell are the most critical ones for software maintenance cost. Therefore, the findings of RQ2 will be able to provide developers a complete guideline about code smell types which they should focus on to increase comprehensibility, maintainability and decrease maintenance cost.

8.2.1 Systems Under Study

In order to conduct the empirical study and answer the research questions, we have analyzed 35 open-source Java systems. For details about the systems, see sub-section 5.2.2 in Chapter 5.

8.2.2 Code Smells Selection

To carry out the study, a list of 13 code smells has been selected similar to the earlier chapters. For details about the list of code smells, see sub-section 5.2.3 in Chapter 5.

8.2.3 Data Analysis

In this sub-section, we explain how we analyze the data to answer the research questions. More specifically, how we measure and analyze the comprehensibility of smelly classes, and the relationship between smell impact and program comprehensibility. Based on the analyses, we discuss how we categorize the code smells into three levels – *high, moderate* and *low*.

8.2.3.1 Comprehensibility Metric

In this study, we introduce the *comprehensibility* metric and show a way to measure it. For this, we manually check whether a class is comprehensible, and how much it is comprehensible. A class is regarded as comprehensible, if the entities used in this program is readable (or understandable). We refer these entities as a *bag of entities* of the class. The bag of entities consists of *class name, method name*, and *variable name* of the class. The keywords used in Java programming language are excluded in this process. The general loop variables such as *i, j* are also excluded here. The entities in the bag of a class are then categorized into following three levels:

1. ***Well Readable:*** If an entity is completely meaningful and helps to understand the program. For example, *calculateSalary()* is a method name which is completely understandable and well readable. The entities of this category carry 100% weight.

2. ***Moderate Readable:*** If an entity is not fully meaningful but helps to understand the program, that is, the entity is partially understandable. For example, *calSalary()* is partially understandable and moderate readable. The entities of this category carry 50% weight.

3. ***Non Readable:*** If an entity is neither meaningful nor understandable. For example, *cs()* is not understandable and non readable. The entities of this

category carry no (0%) weight.

After the categorization, we quantify the comprehensibility metric for a class using Equation 8.1. Here, W_1, W_2 and W_3 represent the weights for the entities of *Well Readable, Moderate Readable* and *Non Readable* level respectively. We weights these $1, 0.5$ and 0 respectively based on the definition as discussed earlier. F_1, F_2 and F_3 represent the frequency of the entities of the corresponding level of the class respectively. $N = F_1 + F_2 + F_3$ represents the total number of entities of the class. For example, if a class contains 10 entities: 5 of these are well readable, 3 moderate readable, and 2 non readable. So, *Comprehensibility Score* $= (5 \times 1 + 3 \times 0.5 + 2 \times 0) \times 100/10 = 0.65 = 65\%$, which means that the class is 65% comprehensible.

$$\textit{Comprehensibility Score} = \frac{W_1 \times F_1 + W_2 \times F_2 + W_3 \times F_3}{N} \times 100\% \tag{8.1}$$

8.2.3.2 Comprehensibility Metric Measurement Process

For the study, we measure *Comprehensibility Score* for the classes of each of the 13 code smell types for each project. To do so, in the first step, we take a project and split the classes into the 13 types of *smelly classes* having the corresponding code smell type. Thus we have 13 smelly directories for the 13 smell types for the project. There are various numbers of classes for each smelly directories. For instance, for *activemq* project, there are 332 *Anti Singleton* smelly classes and on the other hand, *Spaghetti Code* has 4 smelly classes. To manually read all the classes, it requires a huge amount of time. Therefore, we randomly observe 6 smelly classes on an average for each code smell type for the project. We then identify the entities and categorize these into the three levels – *Well Readable, Moderate Readable* and *Non Readable* for all the observed classes of each smell type as a whole. In this way, we analyze the entities and classes for each of the

35 projects. In particular, we observe 2,242 number of smelly classes in total combining 35 projects, approximately 64 (2242/35 = 64) classes per project, and 6 classes per smelly directory per project. For each class, it requires 2.5 minutes to comprehend, 160 minutes (2.5×64) for one project, and 5,605 minutes (2.5×2242) for all the classes of all projects. Table 8.1 summarizes the details of the classes observed and time required to comprehend the classes.

Table 8.1: Details about observed classes and time

	Observed Class#			**Minute**	**Hour**	**Day (5 working hours = 1 day)**
Average Class# Per Smell Type Per Project	6		**Average Estimated Time Per Class**	2.5		
Average Class# Per Project	64		**Estimated Time Per Project**	160	2.67	
Total Class# in Total Projects	2242		**Estimated Time for Total Projects**	5605	93.42	18.68

In the second step, we calculate the *Comprehensibility Score* for each smell type of the project using Equation 8.1. In this case, here, F_1, F_2 and F_3 represent the frequency of the entities of the corresponding level of the corresponding smelly classes cumulatively respectively. For example, for the *Anti Singleton* smell type of the *activemq* project, we observe 10 classes. Here, $F_1 = 537, F_2 = 12$ and $F_3 = 10$ for all the 10 classes cumulatively. So, *Comprehensibility Score* = $(537 \times 1 + 12 \times 0.5 + 10 \times 0) \times 100/(537 + 12 + 10) = 0.97 = 97\%$. In this way, we get 13 number of *Comprehensibility Scores* for the 13 individual code smell types for a project.

In the third step, following the above process, we calculate the *Comprehensibility Scores* for the 35 open source projects. Thus, we get 35 *Comprehensibility Scores* for each code smell type individually. We then calculate the *Average Comprehensibility Scores (ACS)* for each of the 13 code smell types.

In the fourth and final step, the *Average Comprehensibility Scores (ACS)* for

the classes of the smell types are categorized into three *Comprehensibility Levels – Low, Moderate* and *High*. This categorization is based on statistical quartiles, as described by Gupta [177]. We use the first quartile (Q1), the second quartile (Q2), and the third quartile (Q3) to determine these levels of comprehensibility. Specifically, we define the comprehensibility levels as follows:

1. *High Comprehensibility*: Metrics with ACS values greater than or equal to Q3, representing the top 25% of the ACS scores.

2. *Low Comprehensibility*: Metrics with ACS values less than or equal to Q1, representing the bottom 25% of the ACS scores.

3. *Moderate Comprehensibility*: Metrics with $ACS2$ values between Q1 and Q3, representing the middle 50% of the ACS scores.

We combine Q2 and Q3 [183] due to the relatively low number of scores in these quartiles and the minimal difference between the scores in these ranges. This adjustment ensures a more balanced and meaningful categorization. This categorization provides valuable insights for which code smell types affect mostly the comprehensibility. Thus, it helps to identify the key areas for potential improvements in software maintainability and reduction in maintenance cost.

8.3 Result Analysis

This section describes the results about program comprehensibility in respect with the 13 types of code smell. In particular, the answers to the research questions *RQ1* and *RQ2* are discussed here.

8.3.1 Comprehensibility of Smelly Classes [RQ1]

Table 8.2 shows the result *Average Comprehensibility Score (ACS)* of the corresponding smelly classes. These scores actually represent which code smells are how

much comprehensible. So, the lower scores of the *ACS* are regarded as the lower comprehensible of that smelly classes. Hence, these smell types developers should emphasis on while performing maintenance activities. In other words, they should give priority on these smell types while refactoring these to reduce maintenance cost. Therefore, Table 8.2 shows the *ACS* scores in ascending sorted order where top-to-bottom represents low-to-high comprehensibility as well as high-to-low prioritized smell types.

Table 8.2: Average comprehensibility of smelly classes

Code Smell	Average Comprehensibility Score (ACS)
LM	**84.22**
SC	**85.00**
RPB	**86.58**
AS	**88.95**
CDSBP	*89.25*
CC	*89.32*
LPL	*89.59*
LC	*91.43*
Blob	*91.74*
BCSBA	93.01
SG	94.21
MFABNC	94.74
LzC	96.52

[* Bold, italic and plain font in Average Comprehensibility Score (ACS) column indicates *low, moderate* and *high* Comprehensible of the corresponding smelly program respectively]

Moreover, based on the *ACS* scores, we categorized the smell types as the following three comprehensibility levels as discussed in the previous Data Analysis section.

1. *Low comprehensibility smell types*: Long Method, Spaghetti Code, Refused Parent Bequest and Anti Singleton.

2. *Moderate comprehensibility smell types*: Class Data Should Be Private, Complex Class, Long Parameter List, Large Class and Blob.

3. *High comprehensibility smell types*: Base Class Should Be Abstract, Speculative Generality, Many Field Attributes But Not Complex and Lazy Class.

From the table, it is observed that *Long Method* smell type gets the lowest *ACS* score. That is, classes having this type of code smell are likely to be less comprehensible. Thus, it will increase maintenance time and cost, and so developers should emphasis on and refactor this code smell on regular basis. Similarly, all other low comprehensibility smell types such as *SC, RPB* and *AS* should be emphasized during refactoring and maintenance activities. On the other hand, *Lazy Class* smell type gets the highest *ACS* score. Therefore, classes having this code smell are likely to be highly comprehensible, and can affect less on the maintenance cost. Developers might show less priority on such high comprehensibility smell types. Similarly, moderate comprehensibility smell types should get moderate priority or any priority based on the system needs.

8.3.2 Relationship between Smell Impact and Program Comprehensibility [RQ2]

Table 8.3 shows the relationship between smell impact and program comprehensibility by comparing them. The *Average Impact Score (AIS)* column in the table shows the score of combined software and maintainability metrics. Basically, these score are the *Priority Scores* collected from Table 7.1 of Chapter 7. These *AIS* scores are categorized into three impact levels – *High, Moderate* and *Low*. On the other hand, the *Average Comprehensibility Scores (ACS)* shown in Table 8.3 are collected from Table 8.2 of the above RQ1. These *ACS* scores are also categorized into three comprehensibility levels – *Low, Moderate* and *High*.

The hypothesis of this RQ2 is – *high impactful smelly classes have low comprehensibility, moderate impactful classes have moderate, and low impactful classes have high comprehensibility.*

Table 8.3: Impact of code smells versus program comprehensibility of code smells

Impact Level	**Code Smell**	**Average Impact Score (AIS)**		**Code Smell**	**Average Comprehensibility Score (ACS)**	**Comprehensibility Level**
High	AS	0.63		LM	84.22	**Low**
	LPL	0.63		SC	85.00	
	CDSBP	0.53		RPB	86.58	
	Blob	0.53		AS	88.95	
Moderate	LM	0.50		CDSBP	89.25	**Moderate**
	CC	0.47		CC	89.32	
	LC	0.45		LPL	89.59	
	RPB	0.41		LC	91.43	
	SC	0.41		Blob	91.74	
Low	SG	0.40		BCSBA	93.01	**High**
	MFABNC	0.38		SG	94.21	
	BCSBA	0.32		MFABNC	94.74	
	LzC	0.24		LzC	96.52	

From the table, it is observed that only one high impactful smell type out of four has low comprehensibility; two out of five moderate impactful smell types have moderate comprehensibility; and all of the four low impactful smell types have high comprehensibility. In total, out of the 13 smell types, 7 ($\approx 54\%$) have matched with the respective categories, whereas 6 ($\approx 46\%$) have not matched.

From developers' viewpoint as discussed in Chapter 7, it is difficult to differentiate between high and moderate impactful smell types. So, we observe the findings more deeply in another way. To do so, we combine *high* and *moderate* impactful smell types into one category as *strong* impactful and *low* ones into *weak* impactful. Similarly, *low* and *moderate* comprehensible smell types are considered as *weak* comprehensible and *high* ones are *strong* comprehensible. Now, we can conclude from the Table 8.3 that – *all the strong impactful smell types are weak comprehensible and all the weak impactful smell types are strong comprehensible.*

The average impact and comprehensibility scores are plot to generate a graph as shown in Figure 8.1. From this figure, we can say that – as impact score increases, comprehensibility score decreases. Therefore, there exists a negative

relationship between impact of code smells and program comprehensibility.

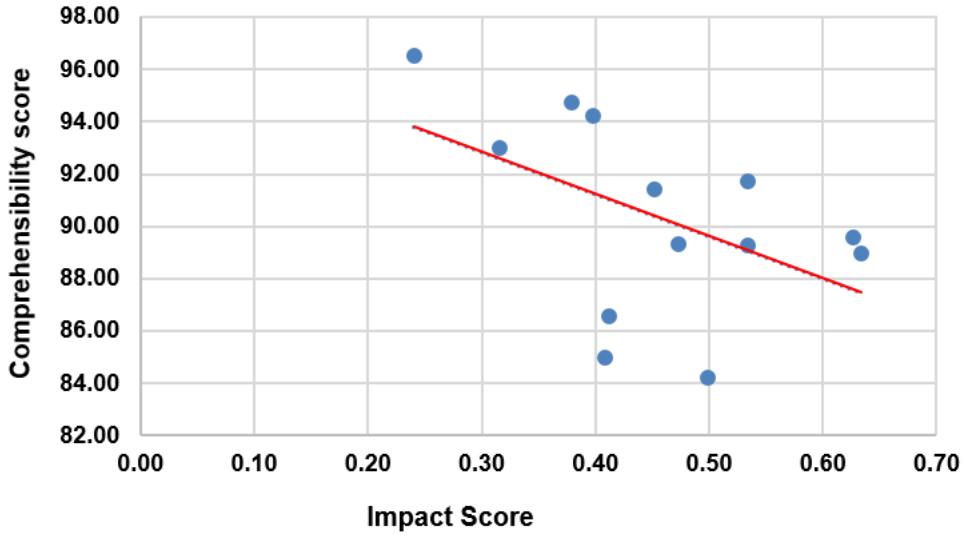


Figure 8.1: Impact of code smells versus program comprehensibility

To quantify the relationship, Spearman's Rank Correlation Coefficient [177] is measure between them. The Spearman's correlation is used in this study because it is a non-parametric measure, and the datasets are ordinal and do not follow a normal distribution. The correlation coefficient is -0.56 which indicates a negative but strong monotonic relationship according to Cohen's guideline [178]. It indicates that as impact score increases, comprehensibility tends to decrease. The $p - value = 0.049 < 0.05$ (5% level of significance) also validates that the relationship is statistically significant.

From these analyses, we can conclude that developers should focus on and prioritize the strong impactful smell types to refactor and enhance the comprehensibility of a software system. Thus, a software system can be developed and maintained at a lower cost.

8.4 Summary

The study of this chapter highlights the critical role of program comprehension in maintaining and improving software systems, with a particular focus on the negative impact of code smells. By analyzing 13 types of code smells, we identified varying levels of comprehensibility associated with each type. The findings show that certain code smells such as *LM, SC, AS,* etc. significantly hinder comprehension, making maintenance more challenging and costly. These insights provide a valuable guide for developers, allowing them to prioritize the refactoring of the most impactful code smells to improve program comprehensibility, enhance maintainability, and reduce long-term maintenance costs. Ultimately, addressing these critical design flaws will lead to more efficient software evolution and better overall system quality.

CHAPTER 9

CONCLUSION AND FUTURE DIRECTION

Code smells should be carefully monitored by developers, since these smells are related to a lot of software quality metrics such as size, coupling, complexity, etc. These smells are also correlated with the two key maintainability metrics – change-proneness and fault-proneness. However, all of the smells are not feasible to be refactored due to time, budget and effort constraints for developing and maintaining a software [2]. Therefore, it is required to prioritize the code smells based on their impact on software quality and maintainability. The results of the research conclude that all the smells do not have the same impact on software systems, and some smells are highly impactful, and hence these are the high prioritized types of code smells. The shortlist of the high prioritized code smells

helps to write or reuse source code carefully without inducing these smells from the beginning of software development. In this chapter, we summarize our main contributions of the empirical study, and recommend some possible future research directions.

9.1 Conclusion

A large scale empirical study has been conducted on 35 open-source Java systems, with the purpose of understanding the impact of code smells on software quality and maintainability metrics. The key findings of the research are highlighted in this section.

Software quality metrics are important factors to improve its design. Most of the code smells have high correlation with size-related software metrics such as the number of files, classes, methods, statements, etc. This means that when the occurrence of the smells increases, these metrics' scores also increase, and hence it decreases the quality of a software. Based on the software quality metrics, the high impactful code smells are – *Long Method, Anti Singleton, Complex Class, Large Class* and *Long Parameter List*. So, developers should emphasize most on these code smells while refactoring to improve software quality.

Software maintainability metrics such as change-proneness and fault-proneness are another key factors to improve its maintainability. Based on the software maintainability metrics, the high impactful code smells are – *Anti Singleton, Blob, Class Data Should Be Private* and *Long Parameter List*. So, developers should focus most on these code smells while refactoring to improve software maintainability.

According to the combined impact of code smells on both software quality and maintainability metrics, the code smells are finally prioritized into three levels – (i) High prioritized: *Anti Singleton, Long Parameter List, Class Data Should Be*

Private and *Blob*; (ii) Moderate prioritized: *Long Method, Complex Class, Large Class, Refused Parent Bequest* and *Spaghetti Code*; and (iii) Low prioritized: *Speculative Generality, Many Field Attributes But Not Complex, Base Class Should Be Abstract* and *Lazy Class*.

Program comprehensibility is a significant factor to reduce maintenance cost of a software. Code smells have a significant impact on the program comprehensibility metric. Code smells such as *Long Method, Spaghetti Code, Refused Parent Bequest* and *Anti Singleton* have the most impact on this metric, that is, these smells make code more difficult to comprehend. So, these code smells should be refactored to reduce maintenance cost through improving code comprehension.

The research also highlights that – what developers think about the impact of the code smells based on their intuition and expertise. Based on the developers' perception based metric (*Inverse of Dev Score, IDS*), the most high impactful code smells are – *Blob, Large Class, Class Data Should Be Private* and *Long Method*. The integration of the human (developers) perception makes the research different from state-of-the-art because these findings help them to rethink about smell prioritization to optimize maintenance cost.

Finally it is concluded that the findings of the research will assist developers to refactor code smells on priority basis that ultimately improves software quality, maintainability, comprehensibility and optimizes maintenance cost. Moreover, these findings not only help practitioners but also researchers to develop automated refactoring tools based on the impact of code smells.

9.2 Future Direction

In this thesis, we conducted a comprehensive study on the impact of code smells on software quality and maintainability. Our findings aid in prioritizing code smells,

ultimately lowering software maintenance costs. However, some concerns remain unaddressed and require further investigation.

- For future research, we will explore additional external software metrics, such as testability, reusability, etc. [7], to assess the impact of code smells and prioritize them accordingly. Additionally, we intend to analyze other influencing factors, including developers' experience and workload, that affect software quality metrics.

- Based on developers' experience, we will categorize the impact of code smells to analyze how varying levels of expertise influence their effects on software quality and maintainability. By examining the relationship between developers' experience and the severity or perception of code smells, we aim to gain deeper insights into how experienced and less-experienced developers approach, identify, and mitigate these issues. This analysis will help in refining prioritization strategies for code smells and tailoring recommendations based on developers' skill levels, ultimately contributing to more effective software maintenance practices.

- Future research can explore the occurrence and impact of code smells across different programming paradigms, such as object-oriented, functional, and procedural programming. This includes analyzing how each paradigm influences the presence of specific code smells, their effects on software maintainability, and the effectiveness of detection and refactoring strategies. Insights from this study can help tailor best practices and tool support for different programming approaches.

- Future research can involve conducting industry case studies to examine how organizations manage code smells in large-scale software projects. This includes analyzing the strategies, tools, and best practices used for detection, prioritization, and mitigation. Additionally, studying the challenges faced

by development teams and the impact of code smell management on software quality and maintenance costs can provide valuable insights for improving industry practices.

Impact of code smells is an important research domain that helps in developing and maintaining a robust software system. In this thesis, impact and prioritization of code smells are analyzed from various key perspectives such as software quality, maintainability, comprehensibility, and developers' perception. Future research about their impact can be further expanded by incorporating other external software and industrial factors discussed above.

Declaration of Generative AI and AI-assisted technologies in the writing process

During the preparation of this work, the author used ChatGPT in order to improve readability and language within some paragraphs of this thesis. After using the tool, the author reviewed and edited the content as needed and take full responsibility for the content of the thesis.

APPENDIX

A

RELATIONSHIP BETWEEN CODE SMELLS AND SOFTWARE METRICS

The correlation scores between each of the 25 software metrics and 13 code smell types have been shown in Table A.1.

Table A.1: Correlation between code smell and software metric

Smell vs Metric	AS	BCSBA	Blob	CDSBP	CC	LC	LzC	LM	LPL	MFABNC	RPB	SC	SG
NOM	0.95993	0.788256	0.937179	0.948287	0.951327	0.962673	0.483065	0.958263	0.933613	0.900372	0.955154	0.960633	0.913428
NDM	0.765655	0.56078	0.668236	0.747574	0.823843	0.799832	0.435835	0.797871	0.785629	0.326087	0.42563	0.731622	0.653798
NPriM	0.919742	0.744476	0.882708	0.840771	0.945931	0.952028	0.40654	0.912605	0.916247	0.700166	0.856722	0.852808	0.734058
NProM	0.910913	0.678148	0.876847	0.916398	0.939141	0.941242	0.409185	0.905246	0.929337	0.245335	0.800463	0.746629	0.770659
NPM	0.956974	0.768118	0.921834	0.911514	0.946285	0.952448	0.485103	0.953782	0.92465	0.890899	0.951375	0.954427	0.907678
NOS	0.951664	0.787198	0.91416	0.920269	0.942783	0.950067	0.528148	0.953782	0.919048	0.975648	0.956743	0.976299	0.936965
NDS	0.961572	0.800902	0.939539	0.955073	0.962673	0.962673	0.558332	0.972269	0.945378	0.975335	0.960973	0.98292	0.941023
NExS	0.936501	0.770957	0.895742	0.895425	0.926395	0.939141	0.452463	0.934454	0.89972	0.858979	0.953873	0.966899	0.890998
CC	0.931899	0.773449	0.91109	0.830962	0.9362	0.947826	0.463697	0.936415	0.90112	0.889365	0.952967	0.937418	0.87261
LCOM	0.987519	0.84827	0.96618	0.881416	0.960992	0.939422	0.356181	0.969188	0.968347	0.899632	0.9027	0.974163	0.916218
DIT	0.979196	0.829827	0.946988	0.975312	0.953985	0.890819	0.618322	0.964986	0.955182	0.99984	0.955363	0.966313	0.972103
IFANIN	0.969664	0.855465	0.94719	0.919987	0.939702	0.891379	0.597388	0.970588	0.968695	1	0.943133	0.960536	0.970592
CBO	0.924897	0.749326	0.913185	0.898012	0.940262	0.908747	0.418453	0.956229	0.940542	0.452768	0.727641	0.859731	0.803694
NOCh	0.86587	0.701314	0.654922	0.886507	0.941943	0.85269	0.538111	0.904799	0.905983	NA	0.959653	0.470353	0.798877
RFC	0.942085	0.75103	0.93477	0.844744	0.944184	0.938161	0.497541	0.938375	0.917367	0.904775	0.950547	0.937057	0.917815
NIM	0.952814	0.771167	0.936196	0.882401	0.948526	0.940822	0.509207	0.954902	0.929132	0.906863	0.941502	0.948242	0.921533
NIV	0.907424	0.791684	0.93181	0.897312	0.93634	0.945444	0.426782	0.938095	0.916247	0.791311	0.920292	0.892523	0.911426
WMC	0.958726	0.786196	0.937511	0.87879	0.953568	0.93676	0.501055	0.957143	0.936695	0.904775	0.955082	0.954929	0.920548
NOC	0.979853	0.850312	0.948671	0.978655	0.949366	0.95651	0.608104	0.97864	0.963233	0.99992	0.951519	0.966437	0.968896
NOF	0.999836	0.890923	0.991162	0.949048	0.99916	1	0.676597	0.99972	1	1	0.984073	0.999928	0.985118
NL	0.953088	0.768953	0.904951	0.844095	0.943483	0.943203	0.597782	0.947619	0.923529	0.982188	0.956743	0.965333	0.941628
BLOC	0.958999	0.756917	0.920624	0.941502	0.931158	0.943343	0.572091	0.947059	0.917367	0.965366	0.959462	0.961487	0.938923
LOC	0.948927	0.792108	0.904293	0.845628	0.941943	0.948106	0.544574	0.944258	0.922409	0.981551	0.959613	0.971884	0.944934
NCL	0.914714	0.759658	0.864442	0.889515	0.845437	0.841796	0.620946	0.884874	0.87535	0.97701	0.922606	0.915627	0.936669
RCTC	-0.25079	0.460177	0.176445	-0.21869	-0.37214	-0.34324	-0.22999	-0.33001	-0.26962	0.77801	0.696033	0.476276	0.723828

Table A.2: Impact of code smell based on software metrics

Code Smell	**Average Correlation Score,** r	**Impact Score,** r^2	**Impact Level**
Long Method	0.89	0.86	**High**
Anti Singleton	0.89	0.85	
Complex Class	0.89	0.85	
Large Class	0.88	0.83	
Long Parameter List	0.88	0.83	
Refused Parent Bequest	0.90	0.82	**Moderate**
Spaghetti Code	0.89	0.82	
Speculative Generality	0.89	0.80	
Blob	0.87	0.79	
Class Data Should Be Private	0.85	0.77	**Low**
Many Field Attributes But Not Complex	0.85	0.76	
Base Class Should Be Abstract	0.76	0.59	
Lazy Class	0.48	0.26	

APPENDIX

B

THREE EXPERT DEVELOPERS' PERCEPTION REGARDING SMELL IMPACT

The perceptions of the three expert developers' regarding the impact of each code smell type in details have been shown in Table B.1, B.2 and B.3 respectively.

Table B.1: Perception of Expert-1 regarding code smell impact

Code Smell	**Impact Score**	**Justification/ Why this score**
AS	8	Global variables must be avoided as much as possible. They are dependency magnets which lead to too much coupling.
BCSBA	13	Abstract classes have their use, but I don't find this very important.
Blob	4	Probably caused by bad design. Failed to decompose properly. Hard to maintain because of complexity.
CDSBP	1	This leads to direct coupling which makes changes very hard.
CC	5	Due to high complexity hard to understand and therefore hard to maintain. Must be broken down into multiple simple classes.
LC	3	See Blob.
LzC	9	More classes make system harder to understand. Functionality can probably be moved to other existing class(es).
LM	2	Hard to understand and thus to maintain. Headache for developers. Refactor into smaller functions.
LPL	7	Always bad. 1 parameter is fine, 2 parameters if you really must. 3 parameters start smelling. 4 or more parameters: Really?
MFABNC	10	Fields should never be public. See CDSBP.
RPB	11	Bad design. This class should probably not extend from that parent.
SC	6	Sloppiness or inexperienced developer. See LM. Refactor.
SG	12	Makes code harder to understand. Class can be merged or inlined.

Table B.2: Perception of Expert-2 regarding code smell impact

Code Smell	Impact Score	Justification/ Why this score
AS	7	Not a major issue in my opinion, in some cases, there may have valid use cases. One use case could be to manage a shared resource such as a database connection pool. But I would argue that it should be used judiciously and only when there is a clear need to restrict the number of instances of a class.
BCSBA	6	Making a base class abstract can help prevent issues and enforce good coding practices, but it also adds complexity and may limit flexibility. I don't see it as a major issue, it depends on case-by-case and the specific needs and requirements of the project. I would rely on the judgement of the author of the code.
Blob	1	I believe that having an excessive amount of code in one class is a serious issue that can cause difficulties in understanding the source code. This creates a high cognitive load for developers, leading to longer comprehension times and making changes more difficult. Ultimately, it affects the maintainability of the codebase and can make the refactoring process time-consuming. Therefore, I would recommend breaking down the functionality into smaller, more focused classes. Doing so would reduce the time it takes to make changes later on.
CDSBP	5	While not necessarily causing issues, ignoring best practices when it comes to the CDSBP code smell can impact the codebase's encapsulation, maintainability, and readability. Improving this code smell would require significant changes and could limit flexibility, so it should be carefully considered and balanced against other priorities and constraints. Ultimately, how to address this code smell should be discussed among team members and a resolution should be agreed upon for the team or specific project, but there shouldn't be a one-size-fits-all solution. As a general best practice, it is recommended to keep the affected code private.
CC	1	I consider this a major problem. This issue can make the class challenging to comprehend, maintain, and expand. It's essential to prevent this code smell, as it can lead to long-term maintainability problems, making it more time-consuming to understand and modify the code in the future.
LC	3	Like Complex class, if a class that has become too large and contains too many responsibilities, making it difficult to comprehend and maintain. As a result, it can be hard to add or change features in the class, leading to maintenance issues and decreased productivity.
LzC	5	Although it may not cause significant issues, it's also possible that it isn't contributing to the system's overall functionality. I would rate its impact as moderate. If a piece of code isn't necessary, I would rather remove it. Keeping code that doesn't contribute anything is unnecessary and can lead to maintenance problems. Even adding a single line of unnecessary code can create issues down the road.
LM	2	A long method is a method that has become too long, making it hard to read, understand and maintain. This can lead to maintenance issues and decreased productivity.
LPL	4	The code becomes difficult to read and maintain when it's excessively long, including the method's parameter list. Although there's no definitive limit on how long the list should be, it can become a subjective topic among developers. However, in general, if the list contains more than five parameters, it should be considered for limiting. One solution is to introduce a data class that only contains the required values.
MFABNC	8	Not a major issue and may have valid use cases. It should be considered carefully and balanced against other priorities and constraints.
RPB	9	Addressing the RPB code smell can improve the adherence to the Liskov Substitution Principle, the maintainability, and reusability of the codebase, but it may require significant refactoring and may have implications on the flexibility of the codebase. Therefore, it should be considered carefully and balanced against other priorities and constraints.
SC	1	This can lead to decreased productivity, increased development time, and higher maintenance costs.
SG	5	Although "Speculative Generality" is not usually a major issue, it's still a code smell that should be avoided as it adds unnecessary complexity to the codebase. The idea is that we'll cross the bridge when we come to it, which means that decisions will be taken later when we have more information. Personally, I feel that delaying decisions can be beneficial, but it's important to note that this code smell is not as significant compared to other code smells.

Table B.3: Perception of Expert-3 regarding code smell impact

Code Smell	**Impact Score**	**Justification/ Why this score**
AS	6	It can cause error-prone codes due to the ability to change fields and make debugging/testing difficult.
BCSBA	11	It can cause leaky abstraction and improper class hierarchy.
Blob	1	It requires significant efforts for development and maintenance due to low readability and difficulty to make changes.
CDSBP	9	It can cause unwanted side-effects and makes debugging difficult.
CC	4	It requires significant efforts for development and maintenance due to low readability and difficulty to make changes.
LC	2	It requires significant efforts for development and maintenance due to low readability and difficulty to make changes.
LzC	13	It can burden the maintenance efforts.
LM	8	It can be a sign of a method doing too much and can cause high development and maintenance efforts.
LPL	5	It can be a sign of a method doing too much and can create confusions for the callers, and can cause high development and maintenance efforts.
MFABNC	12	It can be a sign of low cohesion and requires significant efforts to trace how the fields are being used.
RPB	7	It can be a sign of improper class hierarchy and can cause bloated children, and thus, high maintenance efforts.
SC	3	It requires significant efforts for development and maintenance due to low readability and difficulty to make changes.
SG	10	It increases the maintenance efforts by confusing developers the purpose of the methods.

BIBLIOGRAPHY

[1] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: improving the design of existing code*. Addison-Wesley, 1999.

[2] M. M. Rahman, A. Satter, M. M. A. Joarder, and K. Sakib, "An empirical study on the occurrences of code smells in open source and industrial projects," in *Proceedings of the 16th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, 2022, pp. 289–294.

[3] P. Tripathy and K. Naik, *Software evolution and maintenance: a practitioner's approach*. John Wiley & Sons, 2014.

[4] R. Moser, P. Abrahamsson, W. Pedrycz, A. Sillitti, and G. Succi, "A case study on the impact of refactoring on quality and productivity in an agile team," in *Balancing Agility and Formalism in Software Engineering*. Springer, 2008, pp. 252–266.

[5] M. Arora, S. Sarangdevot, V. S. Rathore, J. Deegwal, and S. Arora, "Refactoring, way for software maintenance," *International Journal of Computer Science Issues (IJCSI)*, vol. 8, no. 2, p. 565, 2011.

[6] C. Vassallo, G. Grano, F. Palomba, H. C. Gall, and A. Bacchelli, "A large-scale empirical exploration on refactoring activities in open source software projects," *Science of Computer Programming*, vol. 180, pp. 1–15, 2019.

[7] M. Alshayeb, "Empirical investigation of refactoring effect on software quality," *Information and Software Technology*, vol. 51, no. 9, pp. 1319–1326, 2009.

[8] ISO/IEC, "Iec 9126-1: Software engineering-product quality-part 1: Quality model," *Geneva, Switzerland: International Organization for Standardization*, vol. 21, no. 9126-1, 2001.

[9] A. Kaur, "A systematic literature review on empirical analysis of the relationship between code smells and software quality attributes," *Archives of Computational Methods in Engineering*, vol. 27, no. 4, pp. 1267–1296, 2020.

[10] F. Palomba, G. Bavota, M. D. Penta, F. Fasano, R. Oliveto, and A. D. Lucia, "On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation," *Empirical Software Engineering*, vol. 23, no. 3, pp. 1188–1221, 2018.

[11] B. Cornelissen, A. Zaidman, A. Van Deursen, L. Moonen, and R. Koschke, "A systematic survey of program comprehension through dynamic analysis," *IEEE Transactions on Software Engineering*, vol. 35, no. 5, pp. 684–702, 2009.

[12] D. Oliveira, R. Santos, B. de Oliveira, M. Monperrus, F. Castor, and F. Madeiral, "Understanding code understandability improvements in code reviews," *IEEE Transactions on Software Engineering*, 2024.

[13] R. Baggen, J. P. Correia, K. Schill, and J. Visser, "Standardized code quality benchmarking for improving software maintainability," *Software Quality Journal*, vol. 20, no. 2, pp. 287–307, 2012.

[14] F. Zhang, A. Mockus, Y. Zou, F. Khomh, and A. E. Hassan, "How does context affect the distribution of software maintainability metrics?" in *2013 IEEE International Conference on Software Maintenance*. IEEE, 2013, pp. 350–359.

[15] H. Liu, Z. Ma, W. Shao, and Z. Niu, "Schedule of bad smell detection and resolution: A new way to save effort," *IEEE Transactions on Software Engineering*, vol. 38, no. 1, pp. 220–235, 2011.

[16] W. H. Brown, R. C. Malveau, H. W. S. McCormick, and T. J. Mowbray, *AntiPatterns: refactoring software, architectures, and projects in crisis*. John Wiley & Sons, 1998.

[17] R. Marinescu, "Detection strategies: Metrics-based rules for detecting design flaws," in *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.* IEEE, 2004, pp. 350–359.

[18] M. J. Munro, "Product metrics for automatic identification of" bad smell" design problems in java source-code," in *11th IEEE International Software Metrics Symposium (METRICS'05)*. IEEE, 2005, pp. 15–15.

[19] F. Khomh, M. D. Penta, Y.-G. Guéhéneuc, and G. Antoniol, "An exploratory study of the impact of antipatterns on class change-and fault-proneness," *Empirical Software Engineering*, vol. 17, no. 3, pp. 243–275, 2012.

[20] N. Sae-Lim, S. Hayashi, and M. Saeki, "How do developers select and prioritize code smells? a preliminary study," in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2017, pp. 484–488.

[21] F. Pecorelli, F. Palomba, F. Khomh, and A. De Lucia, "Developer-driven code smell prioritization," in *Proceedings of the 17th International Conference on Mining Software Repositories*, 2020, pp. 220–231.

[22] S. A. Vidal, C. Marcos, and J. A. Díaz-Pace, "An approach to prioritize code smells for refactoring," *Automated Software Engineering*, vol. 23, no. 3, pp. 501–532, 2016.

[23] N. Sae-Lim, S. Hayashi, and M. Saeki, "Context-based approach to prioritize code smells for prefactoring," *Journal of Software: Evolution and Process*, vol. 30, no. 6, p. e1886, 2018.

[24] D. Sjoberg, A. Yamashita, B. C. D. Anda, A. Mockus, T. Dyba *et al.*, "Quantifying the effect of code smells on maintenance effort," *IEEE Transactions on Software Engineering*, vol. 39, no. 8, pp. 1144–1156, 2013.

[25] L. J. Arthur, *Software evolution: the software maintenance challenge*. Wiley-Interscience, 1988.

[26] P. Marounek, "Simplified approach to effort estimation in software maintenance," *Journal of Systems Integration*, vol. 3, no. 3, pp. 3–10, 2012.

[27] S. M. H. Dehaghani and N. Hajrahimi, "Which factors affect software projects maintenance cost more?" *Acta Informatica Medica*, vol. 21, no. 1, p. 63, 2013.

[28] H. Van Vliet, *Software engineering: principles and practice*. John Wiley & Sons, 2008.

[29] L. Erlikh, "Leveraging legacy system dollars for e-business," *IT professional*, vol. 2, no. 3, pp. 17–23, 2000.

[30] T. Lewowski and L. Madeyski, "How far are we from reproducible research on code smell detection? a systematic literature review," *Information and Software Technology*, vol. 144, p. 106783, 2022.

[31] F. Sabir, F. Palma, G. Rasool, Y.-G. Guéhéneuc, and N. Moha, "A systematic literature review on the detection of smells and their evolution in object-oriented and service-oriented systems," *Software: Practice and Experience*, vol. 49, no. 1, pp. 3–39, 2019.

[32] W. Li and R. Shatnawi, "An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution," *Journal of systems and software*, vol. 80, no. 7, pp. 1120–1128, 2007.

[33] M. D'Ambros, A. Bacchelli, and M. Lanza, "On the impact of design flaws on software defects," in *2010 10th International Conference on Quality Software*. IEEE, 2010, pp. 23–31.

[34] C. Bird, "Clones: What is that smell?" *Empirical Software Engineering*, vol. 15, no. 5, pp. 503–530, 2010.

[35] I. Deligiannis, I. Stamelos, L. Angelis, M. Roumeliotis, and M. Shepperd, "A controlled experiment investigation of an object-oriented design heuristic for maintainability," *Journal of Systems and Software*, vol. 72, no. 2, pp. 129–143, 2004.

[36] A. Lozano and M. Wermelinger, "Assessing the effect of clones on changeability," in *2008 IEEE International Conference on Software Maintenance*. IEEE, 2008, pp. 227–236.

[37] M. Abbes, F. Khomh, Y.-G. Gueheneuc, and G. Antoniol, "An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension," in *2011 15th European Conference on Software Maintenance and Reengineering*. IEEE, 2011, pp. 181–190.

[38] F. Khomh, M. Di Penta, and Y.-G. Gueheneuc, "An exploratory study of the impact of code smells on software change-proneness," in *2009 16th Working Conference on Reverse Engineering*. IEEE, 2009, pp. 75–84.

[39] S. M. Olbrich, D. S. Cruzes, and D. I. Sjøberg, "Are all code smells harmful? a study of god classes and brain classes in the evolution of three open source systems," in *2010 IEEE International Conference on Software Maintenance*. IEEE, 2010, pp. 1–10.

[40] C. Zhu, X. Zhang, Y. Feng, and L. Chen, "An empirical study of the impact of code smell on file changes," in *2018 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 2018, pp. 238–248.

[41] A. Yamashita and L. Moonen, "Exploring the impact of inter-smell relations on software maintainability: An empirical study," in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 682–691.

[42] F. Palomba, G. Bavota, M. Di Penta, F. Fasano, R. Oliveto, and A. De Lucia, "A large-scale empirical study on the lifecycle of code smell co-occurrences," *Information and Software Technology*, vol. 99, pp. 1–10, 2018.

[43] I. Kádár, P. Hegedus, R. Ferenc, and T. Gyimóthy, "A code refactoring dataset and its assessment regarding software maintainability," in *2016 IEEE 23rd International conference on software analysis, Evolution, and Reengineering (SANER)*, vol. 1. IEEE, 2016, pp. 599–603.

[44] O. Chaparro, G. Bavota, A. Marcus, and M. Di Penta, "On the impact of refactoring operations on code quality metrics," in *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE, 2014, pp. 456–460.

[45] A. Murgia, R. Tonelli, M. Marchesi, G. Concas, S. Counsell, J. McFall, and S. Swift, "Refactoring and its relationship with fan-in and fan-out: An empirical study," in *2012 16th European Conference on Software Maintenance and Reengineering*. IEEE, 2012, pp. 63–72.

[46] Y. Kataoka, T. Imai, H. Andou, and T. Fukaya, "A quantitative evaluation of maintainability enhancement by refactoring," in *International Conference on Software Maintenance, 2002. Proceedings.* IEEE, 2002, pp. 576–585.

[47] B. Du Bois, S. Demeyer, and J. Verelst, "Refactoring-improving coupling and cohesion of existing code," in *11th working conference on reverse engineering*. IEEE, 2004, pp. 144–151.

[48] S. Vidal, H. Vazquez, J. A. Diaz-Pace, C. Marcos, A. Garcia, and W. Oizumi, "Jspirit: a flexible tool for the analysis of code smells," in *Proceedings of the 34th International Conference of the Chilean Computer Science Society (SCCC)*. IEEE, 2015, pp. 1–6.

[49] A. Ouni, M. Kessentini, S. Bechikh, and H. Sahraoui, "Prioritizing code-smells correction tasks using chemical reaction optimization," *Software Quality Journal*, vol. 23, no. 2, pp. 323–361, 2015.

[50] A. Chatzigeorgiou and A. Manakos, "Investigating the evolution of bad smells in object-oriented code," in *2010 Seventh International Conference on the Quality of Information and Communications Technology*. IEEE, 2010, pp. 106–115.

[51] ——, "Investigating the evolution of code smells in object-oriented systems," *Innovations in Systems and Software Engineering*, vol. 10, no. 1, pp. 3–18, 2014.

[52] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, and D. Poshyvanyk, "When and why your code starts to smell bad (and whether the smells go away)," *IEEE Transactions on Software Engineering*, vol. 43, no. 11, pp. 1063–1088, 2017.

[53] A. Rani and J. K. Chhabra, "Evolution of code smells over multiple versions of softwares: An empirical investigation," in *2017 2nd International Conference for Convergence in Technology (I2CT)*. IEEE, 2017, pp. 1093–1098.

[54] I. Griffith, S. Wahl, and C. Izurieta, "Truerefactor: An automated refactoring tool to improve legacy system and application comprehensibility," in *Proceedings of the 24th International Conference on Computer Applications in Industry and Engineering (CAINE)*, vol. 1. ISCA, 2011.

[55] N. Tsantalis, T. Chaikalis, and A. Chatzigeorgiou, "Ten years of jdeodorant: Lessons learned from the hunt for smells," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2018, pp. 4–14.

[56] D. Silva and M. T. Valente, "Refdiff: detecting refactorings in version histories," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 2017, pp. 269–279.

[57] F. Palomba, A. Zaidman, R. Oliveto, and A. De Lucia, "An exploratory study on the relationship between changes and refactoring," in *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*. IEEE, 2017, pp. 176–185.

[58] A. Brito, A. Hora, and M. T. Valente, "Refactoring graphs: Assessing refactoring over time," in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2020, pp. 367–377.

[59] E. A. AlOmar, M. W. Mkaouer, A. Ouni, and M. Kessentini, "On the impact of refactoring on the relationship between quality attributes and design metrics," in *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 2019, pp. 1–11.

[60] C. Tavares, M. Bigonha, and E. Figueiredo, "Analyzing the impact of refactoring on bad smells," in *Proceedings of the 34th Brazilian Symposium on Software Engineering*. ACM, 2020, pp. 97–101.

[61] Z. Soh, A. Yamashita, F. Khomh, and Y.-G. Guéhéneuc, "Do code smells impact the effort of different maintenance programming activities?" in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering*, vol. 1. IEEE, 2016, pp. 393–402.

[62] V. Garousi and B. Küçük, "Smells in software test code: A survey of knowledge in industry and academia," *Journal of systems and software*, vol. 138, pp. 52–81, 2018.

[63] F. A. Fontana, V. Ferme, and M. Zanoni, "Towards assessing software architecture quality by exploiting code smell relations," in *2015 IEEE/ACM 2nd International Workshop on Software Architecture and Metrics*. IEEE, 2015, pp. 1–7.

[64] F. Palomba, D. A. Tamburri, F. A. Fontana, R. Oliveto, A. Zaidman, and A. Serebrenik, "Beyond technical aspects: How do community smells influence the intensity of code smells?" *IEEE transactions on software engineering*, vol. 47, no. 1, pp. 108–129, 2018.

[65] E. Van Emden and L. Moonen, "Java quality assurance by detecting code smells," in *Reverse Engineering, 2002. Proceedings. Ninth Working Conference on*. IEEE, 2002, pp. 97–106.

[66] F. A. Fontana, P. Braione, and M. Zanoni, "Automatic detection of bad smells in code: An experimental assessment." *Journal of Object Technology*, vol. 11, no. 2, pp. 5–1, 2012.

[67] F. Jaafar, Y.-G. Guéhéneuc, S. Hamel, and F. Khomh, "Mining the relationship between anti-patterns dependencies and fault-proneness," in *2013 20th Working Conference on Reverse Engineering (WCRE)*, 2013, pp. 351–360.

[68] S. Boutaib, S. Bechikh, F. Palomba, M. Elarbi, M. Makhlouf, and L. B. Said, "Code smell detection and identification in imbalanced environments," *Expert Systems with Applications*, vol. 166, p. 114076, 2021.

[69] W. F. Opdyke, *Refactoring object-oriented frameworks*. University of Illinois at Urbana-Champaign, 1992, phD Thesis.

[70] A. Kaur and M. Kaur, "Analysis of code refactoring impact on software quality," in *MATEC Web of Conferences*, vol. 57. EDP Sciences, 2016, p. 02012.

[71] J. Al Dallal and A. Abdin, "Empirical evaluation of the impact of object-oriented code refactoring on quality attributes: A systematic literature review," *IEEE Transactions on Software Engineering*, vol. 44, no. 1, pp. 44–69, 2017.

[72] B. F. dos Santos Neto, M. Ribeiro, V. T. Da Silva, C. Braga, C. J. P. De Lucena, and E. de Barros Costa, "Autorefactoring: A platform to build refactoring agents," *Expert Systems with Applications*, vol. 42, no. 3, pp. 1652–1664, 2015.

[73] A. Peruma, S. Simmons, E. A. AlOmar, C. D. Newman, M. W. Mkaouer, and A. Ouni, "How do i refactor this? an empirical study on refactoring trends and topics in stack overflow," *Empirical Software Engineering*, vol. 27, no. 1, p. 11, 2022.

[74] M. O'Keeffe and M. O. Cinnéide, "Search-based refactoring for software maintenance," *Journal of Systems and Software*, vol. 81, no. 4, pp. 502–516, 2008.

[75] A. Chugh, S. Manchanda, and P. Khosla, "Improving software maintainability through refactoring-an empirical study," *International Journal of Advances in Electronics and Computer Science*, vol. 2, no. 4, pp. 88–93, 2015.

[76] A. Chug and M. Gupta, "A quality enhancement through defect reduction using refactoring operation," in *2017 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*. IEEE, 2017, pp. 1869–1875.

[77] B. Geppert, A. Mockus, and F. Robler, "Refactoring for changeability: A way to go?" in *11th IEEE International Software Metrics Symposium (METRICS'05)*. IEEE, 2005, pp. 10–pp.

[78] T. Mens and T. Tourwé, "A survey of software refactoring," *IEEE Transactions on software engineering*, vol. 30, no. 2, pp. 126–139, 2004.

[79] R. Haas and B. Hummel, "Deriving extract method refactoring suggestions for long methods," in *International Conference on Software Quality*. Springer, 2015, pp. 144–155.

[80] M. Shahidi, M. Ashtiani, and M. Zakeri-Nasrabadi, "An automated extract method refactoring approach to correct the long method code smell," *Journal of Systems and Software*, vol. 187, p. 111221, 2022.

[81] R. C. Martin, *Agile Software Development: Principles, Patterns, and Practices*. USA: Prentice Hall PTR, 2003.

[82] N. Moha, Y.-G. Guéhéneuc, L. Duchien, and A.-F. Le Meur, "Decor: A method for the specification and detection of code and design smells," *IEEE Transactions on Software Engineering*, vol. 36, no. 1, pp. 20–36, 2009.

[83] T. Sharma and D. Spinellis, "A survey on software smells," *Journal of Systems and Software*, vol. 138, pp. 158–173, 2018.

[84] G. Suryanarayana, G. Samarthyam, and T. Sharma, *Refactoring for software design smells: managing technical debt*. Morgan Kaufmann, 2014.

[85] A. Martini, J. Bosch, and M. Chaudron, "Architecture technical debt: Understanding causes and a qualitative model," in *2014 40th EUROMICRO Conference on Software Engineering and Advanced Applications*. IEEE, 2014, pp. 85–92.

[86] M. Lavallée and P. N. Robillard, "Why good developers write bad code: An observational case study of the impacts of organizational factors on software quality," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. IEEE, 2015, pp. 677–687.

[87] K. Curcio, A. Malucelli, S. Reinehr, and M. A. Paludo, "An analysis of the factors determining software product quality: A comparative study," *Computer Standards & Interfaces*, vol. 48, pp. 10–18, 2016.

[88] E. Tom, A. Aurum, and R. Vidgen, "An exploration of technical debt," *Journal of Systems and Software*, vol. 86, no. 6, pp. 1498–1516, 2013.

[89] S. Acuna, M. Gómez, and N. Juristo, "Towards understanding the relationship between team climate and software quality–a quasi-experimental study," *Empirical Software Engineering*, vol. 13, no. 4, pp. 339–342, 2008.

[90] M. S. Haque, J. Carver, and T. Atkison, "Causes, impacts, and detection approaches of code smell: a survey," in *Proceedings of the ACMSE 2018 Conference*, 2018, pp. 1–8.

[91] R. C. Martin, *Clean code: a handbook of agile software craftsmanship*, 1st ed. Upper Saddle River, NJ: Pearson Education, 2009.

[92] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*, 1st ed. Upper Saddle River, NJ: Pearson Deutschland GmbH, 1995.

[93] D. Das, A. A. Maruf, R. Islam, N. Lambaria, S. Kim, A. S. Abdelfattah, T. Cerny, K. Frajtak, M. Bures, and P. Tisnovsky, "Technical debt resulting from architectural degradation and code smells: a systematic mapping study," *ACM SIGAPP Applied Computing Review*, vol. 21, no. 4, pp. 20–36, 2022.

[94] B. W. Boehm, J. R. Brown, and M. Lipow, "Quantitative evaluation of software quality," in *Proceedings of the 2nd international conference on Software engineering*. IEEE Computer Society, 1976, pp. 592–605.

[95] K. Stroggylos and D. Spinellis, "Refactoring–does it improve software quality?" in *Fifth International Workshop on Software Quality (WoSQ'07: ICSE Workshops 2007)*. IEEE, 2007, pp. 10–10.

[96] R. Jabangwe, J. Börstler, D. Šmite, and C. Wohlin, "Empirical evidence on the link between object-oriented measures and external quality attributes: a systematic literature review," *Empirical Software Engineering*, vol. 20, pp. 640–693, 2015.

[97] International Organization for Standardization, *Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — Product quality model*, ISO Std. ISO/IEC 25 010:2023, 2011.

[98] G. Lacerda, F. Petrillo, M. Pimenta, and Y. G. Guéhéneuc, "Code smells and refactoring: A tertiary systematic review of challenges and observations," *Journal of Systems and Software*, vol. 167, p. 110610, 2020.

[99] M. Shatnawi, "Predicting software change-proneness from software evolution using machine learning," *International Journal of Information and Knowledge Management*, vol. 18, pp. 769–790, 2023.

[100] N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," *Proceedings of the 27th international conference on Software engineering*, pp. 284–292, 2005.

[101] A. Trifu and U. Reupke, "Towards automated restructuring of object oriented systems," in *11th European Conference on Software Maintenance and Reengineering*. IEEE, 2007, pp. 39–48.

[102] F. Pecorelli, S. Lujan, V. Lenarduzzi, F. Palomba, and A. De Lucia, "On the adequacy of static analysis warnings with respect to code smell prediction," *Empirical Software Engineering*, vol. 27, no. 3, p. 64, 2022.

[103] R. Verma, K. Kumar, and H. K. Verma, "Code smell prioritization in object-oriented software systems: A systematic literature review," *Journal of Software: Evolution and Process*, vol. 35, no. 12, p. e2536, 2023.

[104] R. Alfayez, R. Winn, W. Alwehaibi, E. Venson, and B. Boehm, "How sonarqube-identified technical debt is prioritized: An exploratory case study," *Information and Software Technology*, vol. 156, p. 107147, 2023.

[105] A. Kovačević, J. Slivka, D. Vidaković, K.-G. Grujić, N. Luburić, S. Prokić, and G. Sladić, "Automatic detection of long method and god class code smells through neural source code embeddings," *Expert Systems with Applications*, vol. 204, p. 117607, 2022.

[106] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, D. Poshyvanyk, and A. De Lucia, "Mining version histories for detecting code smells," *IEEE Transactions on Software Engineering*, vol. 41, no. 5, pp. 462–489, 2014.

[107] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk, "Detecting bad smells in source code using change history information," in *2013 IEEE/ACM 28th International Conference on Automated Software Engineering*. IEEE, 2013, pp. 268–278.

[108] M. M. Lehman, "Laws of software evolution revisited," in *European workshop on software process technology*. Springer, 1996, pp. 108–124.

[109] A. S. Cairo, G. d. F. Carneiro, and M. P. Monteiro, "The impact of code smells on software bugs: A systematic literature review," *Information*, vol. 9, no. 11, p. 273, 2018.

[110] M. Fokaefs, N. Tsantalis, and A. Chatzigeorgiou, "Jdeodorant: Identification and removal of feature envy bad smells," in *IEEE International Conference on Software Maintenance*. IEEE, 2007, pp. 519–520.

[111] N. Tsantalis, T. Chaikalis, and A. Chatzigeorgiou, "Jdeodorant: Identification and removal of type-checking bad smells," in *12th European Conference on Software Maintenance and Reengineering*. IEEE, 2008, pp. 329–331.

[112] M. A. Bigonha, K. Ferreira, P. Souza, B. Sousa, M. Januário, and D. Lima, "The usefulness of software metric thresholds for detection of bad smells and fault prediction," *Information and Software Technology*, vol. 115, pp. 79–92, 2019.

[113] R. Marinescu, G. Ganea, and I. Verebi, "Incode: Continuous quality assessment and improvement," in *2010 14th European Conference on Software Maintenance and Reengineering*. IEEE, 2010, pp. 274–275.

[114] B. L. Sousa, M. A. Bigonha, and K. A. Ferreira, "An exploratory study on cooccurrence of design patterns and bad smells using software metrics," *Software: Practice and Experience*, vol. 49, no. 7, pp. 1079–1113, 2019.

[115] M. I. Azeem, F. Palomba, L. Shi, and Q. Wang, "Machine learning techniques for code smell detection: A systematic literature review and meta-analysis," *Information and Software Technology*, vol. 108, pp. 115–138, 2019.

[116] J. Martins, C. Bezerra, A. Uchôa, and A. Garcia, "How do code smell co-occurrences removal impact internal quality attributes? a developers' perspective," in *Proceedings of the XXXV Brazilian Symposium on Software Engineering*, 2021, pp. 54–63.

[117] F. A. Fontana, M. V. Mäntylä, M. Zanoni, and A. Marino, "Comparing and experimenting machine learning techniques for code smell detection," *Empirical Software Engineering*, vol. 21, no. 3, pp. 1143–1191, 2016.

[118] A. Ouni, M. Kessentini, H. Sahraoui, and M. S. Hamdi, "Search-based refactoring: Towards semantics preservation," in *2012 IEEE International Conference on Software Maintenance*. IEEE, 2012, pp. 347–356.

[119] A. Ouni, M. Kessentini, H. Sahraoui, and M. Boukadoum, "Maintainability defects detection and correction: a multi-objective approach," *Automated Software Engineering*, vol. 20, pp. 47–79, 2013.

[120] M. Kessentini, R. Mahaouachi, and K. Ghedira, "What you like in design use to correct bad-smells," *Software Quality Journal*, vol. 21, pp. 551–571, 2013.

[121] A. Ouni, M. Kessentini, H. Sahraoui, K. Inoue, and M. S. Hamdi, "Improving multi-objective code-smells correction using development history," *Journal of Systems and Software*, vol. 105, pp. 18–39, 2015.

[122] M. Y. Mhawish and M. Gupta, "Predicting code smells and analysis of predictions: using machine learning techniques and software metrics," *Journal of Computer Science and Technology*, vol. 35, no. 6, pp. 1428–1445, 2020.

[123] B. Bafandeh Mayvan, A. Rasoolzadegan, and A. Javan Jafari, "Bad smell detection using quality metrics and refactoring opportunities," *Journal of Software: Evolution and Process*, vol. 32, no. 8, p. e2255, 2020.

[124] M. Mäntylä, J. Vanhanen, and C. Lassenius, "A taxonomy and an initial empirical study of bad smells in code," in *International Conference on Software Maintenance*. IEEE, 2003, pp. 381–384.

[125] A. Yamashita and L. Moonen, "Do code smells reflect important maintainability aspects?" in *2012 28th IEEE International Conference on Software Maintenance*. IEEE, 2012, pp. 306–315.

[126] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, and D. Poshyvanyk, "When and why your code starts to smell bad," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. IEEE, 2015, pp. 403–414.

[127] F. Palma, L. An, F. Khomh, N. Moha, and Y.-G. Guéhéneuc, "Investigating the change-proneness of service patterns and antipatterns," in *2014 IEEE 7th International Conference on Service-Oriented Computing and Applications*. IEEE, 2014, pp. 1–8.

[128] R. Nascimento and C. Sant'Anna, "Investigating the relationship between bad smells and bugs in software systems," in *Proceedings of the 11th Brazilian Symposium on Software Components, Architectures, and Reuse*, 2017, pp. 1–10.

[129] T. Hall, M. Zhang, D. Bowes, and Y. Sun, "Some code smells have a significant but small effect on faults," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 23, no. 4, pp. 1–39, 2014.

[130] A. Saboury, P. Musavi, F. Khomh, and G. Antoniol, "An empirical study of code smells in javascript projects," in *Proceedings of the 24th International conference on software analysis, evolution and reengineering*. IEEE, 2017, pp. 294–305.

[131] Y. Zhong, M. Shi, J. He, C. Fang, and Z. Chen, "Security-based code smell definition, detection, and impact quantification in android," *Software: Practice and Experience*, vol. 53, no. 11, pp. 2296–2321, 2023.

[132] I. Ahmed, C. Brindescu, U. A. Mannan, C. Jensen, and A. Sarma, "An empirical examination of the relationship between code smells and merge conflicts," in *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 2017, pp. 58–67.

[133] M. M. Rahman, T. Ahammed, M. M. A. Joarder, and K. Sakib, "Does code smell frequency have a relationship with fault-proneness?" in *Proceedings of the 27th International Conference on Evaluation and Assessment in Software Engineering*, 2023, pp. 261–262.

[134] M. M. Rahman, R. R. Riyadh, and M. R. Rahman, "Recommendation of move method refactorings using coupling, cohesion and contextual similarity," in *2017 IEEE International Conference on Imaging, Vision & Pattern Recognition (icIVPR)*. IEEE, 2017, pp. 1–6.

[135] M. M. Rahman, R. R. Riyadh, S. M. Khaled, A. Satter, and M. R. Rahman, "Mmruc3: A recommendation approach of move method refactoring using coupling, cohesion, and contextual similarity to enhance software design," *Software: Practice and Experience*, vol. 48, no. 9, pp. 1560–1587, 2018.

[136] A. Yamashita, M. Zanoni, F. A. Fontana, and B. Walter, "Inter-smell relations in industrial and open source systems: A replication and comparative analysis," in *2015 IEEE International Conference on Software Maintenance and Evolution*. IEEE, 2015, pp. 121–130.

[137] J. Martins, C. Bezerra, A. Uchôa, and A. Garcia, "Are code smell co-occurrences harmful to internal quality attributes? a mixed-method study," in *Proceedings of the 34th Brazilian Symposium on Software Engineering*, 2020, pp. 52–61.

[138] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, and A. De Lucia, "Do they really smell bad? a study on developers' perception of bad code smells," in *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE, 2014, pp. 101–110.

[139] D. Taibi, A. Janes, and V. Lenarduzzi, "How developers perceive smells in source code: A replicated study," *Information and Software Technology*, vol. 92, pp. 223–235, 2017.

[140] G. Bavota, A. De Lucia, M. Di Penta, R. Oliveto, and F. Palomba, "An experimental investigation on the innate relationship between quality and refactoring," *Journal of Systems and Software*, vol. 107, pp. 1–14, 2015.

[141] E. Murphy-Hill, C. Parnin, and A. P. Black, "How we refactor, and how we know it," *IEEE Transactions on Software Engineering*, vol. 38, no. 1, pp. 5–18, 2011.

[142] G. Hecht, N. Moha, and R. Rouvoy, "An empirical study of the performance impacts of android code smells," in *Proceedings of the international conference on mobile software engineering and systems*, 2016, pp. 59–69.

[143] M. A. Alkandari, A. Kelkawi, and M. O. Elish, "An empirical investigation on the effect of code smells on resource usage of android mobile applications," *IEEE Access*, vol. 9, pp. 61 853–61 863, 2021.

[144] R. Singh, A. Bindal, and A. Kumar, "Reducing maintenance efforts of developers by prioritizing different code smells," *International Journal of Innovative Technology and Exploring Engineering (IJITEE)*, vol. 8, no. 8S3, pp. 2223–2232, 2019.

[145] S. Vidal, E. Guimaraes, W. Oizumi, A. Garcia, A. D. Pace, and C. Marcos, "Identifying architectural problems through prioritization of code smells," in *2016 X Brazilian Symposium on Software Components, Architectures and Reuse (SBCARS)*. IEEE, 2016, pp. 41–50.

[146] N. Sae-Lim, S. Hayashi, and M. Saeki, "An investigative study on how developers filter and prioritize code smells," *IEICE Transactions on Information and Systems*, vol. 101, no. 7, pp. 1733–1742, 2018.

[147] ——, "Context-based code smells prioritization for prefactoring," in *Proceedings of the 24th International Conference on Program Comprehension*. IEEE, 2016, pp. 1–10.

[148] M. R. Islam, A. Al Maruf, and T. Cerny, "Code smell prioritization with business process mining and static code analysis: A case study," *Electronics*, vol. 11, no. 12, p. 1880, 2022.

[149] T. Guggulothu and S. A. Moiz, "An approach to suggest code smell order for refactoring," in *International Conference on Emerging Technologies in Computer Engineering*. Springer, 2019, pp. 250–260.

[150] ——, "Prioritize the code smells based on design quality impact," in *ICICCT 2019–System Reliability, Quality Control, Safety, Maintenance and Management*. Springer, 2020, pp. 406–415.

[151] F. A. Fontana and M. Zanoni, "Code smell severity classification using machine learning techniques," *Knowledge-Based Systems*, vol. 128, pp. 43–58, 2017.

[152] J. Nanda and J. K. Chhabra, "Sshm: Smote-stacked hybrid model for improving severity classification of code smell," *International Journal of Information Technology*, vol. 14, no. 5, pp. 2701–2707, 2022.

[153] A. Gupta and N. K. Chauhan, "A severity-based classification assessment of code smells in kotlin and java application," *Arabian Journal for Science and Engineering*, vol. 47, no. 2, pp. 1831–1848, 2022.

[154] A. Oliveira, L. Sousa, W. Oizumi, and A. Garcia, "On the prioritization of design-relevant smelly elements: A mixed-method, multi-project study," in *Proceedings of the 13th Brazilian Symposium on Software Components, Architectures, and Reuse*, 2019, pp. 83–92.

[155] T. Alshammari and M. Alshayeb, "Toward a software bad smell prioritization model for software maintainability," *Arabian Journal for Science and Engineering*, vol. 46, no. 9, pp. 9157–9177, 2021.

[156] E. Guimaraes, S. Vidal, A. Garcia, J. Diaz Pace, and C. Marcos, "Exploring architecture blueprints for prioritizing critical code anomalies: Experiences and tool support," *Software: Practice and Experience*, vol. 48, no. 5, pp. 1077–1106, 2018.

[157] R. Arcoverde, E. Guimarães, I. Macía, A. Garcia, and Y. Cai, "Prioritization of code anomalies based on architecture sensitiveness," in *2013 27th Brazilian Symposium on Software Engineering*. IEEE, 2013, pp. 69–78.

[158] S. Vidal, W. Oizumi, A. Garcia, A. D. Pace, and C. Marcos, "Ranking architecturally critical agglomerations of code smells," *Science of Computer Programming*, vol. 182, pp. 64–85, 2019.

[159] R. Verma, K. Kumar, and H. K. Verma, "Prioritizing god class code smells in object-oriented software using fuzzy inference system," *Arabian Journal for Science and Engineering*, pp. 1–28, 2024.

[160] A. Rani and J. K. Chhabra, "Prioritization of smelly classes: A two phase approach (reducing refactoring efforts)," in *2017 3rd International Conference on Computational Intelligence & Communication Technology (CICT)*. IEEE, 2017, pp. 1–6.

[161] Y. Mehta, P. Singh, and A. Sureka, "Analyzing code smell removal sequences for enhanced software maintainability," in *2018 Conference on Information and Communication Technology (CICT)*. IEEE, 2018, pp. 1–6.

[162] R. Malhotra, A. Chug, and P. Khosla, "Prioritization of classes for refactoring: A step towards improvement in software quality," in *Proceedings of the Third International Symposium on Women in Computing and Informatics*, 2015, pp. 228–234.

[163] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 476–493, 1994.

[164] M. Zhang, N. Baddoo, P. Wernick, and T. Hall, "Prioritising refactoring using code bad smells," in *2011 IEEE fourth international conference on software testing, verification and validation workshops*. IEEE, 2011, pp. 458–464.

[165] A. Chug and S. Tarwani, "Determination of optimum refactoring sequence using a* algorithm after prioritization of classes," in *2017 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*. IEEE, 2017, pp. 1624–1630.

[166] N. Tsantalis and A. Chatzigeorgiou, "Ranking refactoring suggestions based on historical volatility," in *2011 15th European Conference on Software Maintenance and Reengineering*. IEEE, 2011, pp. 25–34.

[167] K. Alkharabsheh, S. Alawadi, K. Ignaim, N. Zanoon, Y. Crespo, E. Manso, and J. A. Taboada, "Prioritization of god class design smell: A multi-criteria based approach," *Journal of King Saud University-Computer and Information Sciences*, vol. 34, no. 10, pp. 9332–9342, 2022.

[168] R. Singh, A. Bindal, and A. Kumar, "Software engineering paradigm for real-time accurate decision making for code smell prioritization," in *Data Science and Innovations for Intelligent Systems*. CRC Press, 2021, pp. 67–93.

[169] I. Griffith, S. Wahl, and C. Izurieta, "Evolution of legacy system comprehensibility through automated refactoring," in *Proceedings of the International Workshop on Machine Learning Technologies in Software Engineering*, 2011, pp. 35–42.

[170] V. Lenarduzzi, T. Besker, D. Taibi, A. Martini, and F. A. Fontana, "A systematic literature review on technical debt prioritization: Strategies, processes, factors, and tools," *Journal of Systems and Software*, vol. 171, p. 110827, 2021.

[171] D. Badampudi, C. Wohlin, and K. Petersen, "Software component decision-making: In-house, oss, cots or outsourcing-a systematic literature review," *Journal of Systems and Software*, vol. 121, pp. 105–124, 2016.

[172] J. Robbins, "Adopting open source software engineering (osse) practices by adopting osse tools," 2005.

[173] K.-J. Stol, M. A. Babar, P. Avgeriou, and B. Fitzgerald, "A comparative study of challenges in integrating open source software and inner source software," *Information and Software Technology*, vol. 53, no. 12, pp. 1319–1336, 2011.

[174] D. I. Sjøberg, A. Yamashita, B. C. Anda, A. Mockus, and T. Dybå, "Quantifying the effect of code smells on maintenance effort," *IEEE Transactions on Software Engineering*, vol. 39, no. 8, pp. 1144–1156, 2012.

[175] F. Pecorelli, F. Palomba, D. Di Nucci, and A. De Lucia, "Comparing heuristic and machine learning approaches for metric-based code smell detection," in *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. IEEE, 2019, pp. 93–104.

[176] C. Manning, P. Raghavan, and H. Schütze, *Introduction to information retrieval*. Cambridge University Press, 2010.

[177] S. Gupta and V. Kapoor, *Fundamentals of mathematical statistics*, 11th ed. Sultan Chand & Sons, 2020.

[178] J. Cohen, *Statistical power analysis for the behavioral sciences*. Routledge, 2013.

[179] P. Runeson, M. Host, A. Rainer, and B. Regnell, *Case study research in software engineering: Guidelines and examples*. John Wiley & Sons, 2012.

[180] SciTools, "Scitools – software intelligence for developers," 2025, accessed: 2025-04-19. [Online]. Available: https://scitools.com/

[181] M. J. Crawley, *The R book*, 2nd ed. John Wiley & Sons, 2012.

[182] P. Schober, C. Boer, and L. A. Schwarte, "Correlation coefficients: appropriate use and interpretation," *Anesthesia & Analgesia*, vol. 126, no. 5, pp. 1763–1768, 2018.

[183] K. K. Arkema, G. Guannel, G. Verutes, S. A. Wood, A. Guerry, M. Ruckelshaus, P. Kareiva, M. Lacayo, and J. M. Silver, "Coastal habitats shield people and property from sea-level rise and storms," *Nature climate change*, vol. 3, no. 10, pp. 913–918, 2013.

[184] M. W. Mkaouer, M. Kessentini, S. Bechikh, K. Deb, and M. Ó Cinnéide, "High dimensional search-based software engineering: finding tradeoffs among 15 objectives for automating software refactoring using nsga-iii," in *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation*. ACM, 2014, pp. 1263–1270.

[185] S. Kaur and P. Singh, "How does object-oriented code refactoring influence software quality? research landscape and challenges," *Journal of Systems and Software*, vol. 157, p. 110394, 2019.

[186] H. Schütze, C. D. Manning, and P. Raghavan, *Introduction to information retrieval*. Cambridge University Press, 2008.

[187] T. A. Corbi, "Program understanding: Challenge for the 1990s," *IBM Systems Journal*, vol. 28, no. 2, pp. 294–306, 1989.

[188] R. Minelli, A. Mocci, and M. Lanza, "I know what you did last summer: An investigation of how developers spend their time," in *Proceedings of the 23rd IEEE International Conference on Program Comprehension (ICPC)*. IEEE, 2015, pp. 25–35.

[189] X. Xia, D. Lo, A. E. Hassan, and X. Wang, "Measuring program comprehension: A large-scale field study with professionals," *IEEE Transactions on Software Engineering*, vol. 44, no. 10, pp. 951–976, 2017.

[190] D. Hendrix, J. H. Cross, and S. Maghsoodloo, "The effectiveness of control structure diagrams in source code comprehension activities," *IEEE Transactions on Software Engineering*, vol. 28, no. 5, pp. 463–477, 2002.