IMPACT OF COMMUNITY SMELLS ON SOFTWARE MAINTAINABILITY

TOUKIR AHAMMED
Exam Roll: 60802
Session: 2020-2021
Registration Number: 2015-318-000

A Thesis
Submitted to the Master of Science in Software Engineering Program Office
of the Institute of Information Technology, University of Dhaka
in Partial Fulfillment of the
Requirements for the Degree

MASTER OF SCIENCE IN SOFTWARE ENGINEERING

Institute of Information Technology
University of Dhaka
DHAKA, BANGLADESH

IMPACT OF COMMUNITY SMELLS ON SOFTWARE MAINTAINABILITY

TOUKIR AHAMMED

Approved:

Signature *Date*

Supervisor: Dr. Kazi Muheymin-Us-Sakib

To *my parents*,
who have always been there for me and inspired me

Abstract

Community smells are communication and collaboration issues among developers in a software development community that make software projects difficult to maintain. These social and organizational anti-patterns create development knowledge gap, lack of awareness about each other's work and mistrust among developers. Thus, developers' maintenance decisions and activities such as bug fixing, source code refactoring, etc., can be influenced by community smells. Although the relationship between community smells and a few technical aspects of source code, such as bug and code smell, has been explored in the existing studies, the impact of community smells on maintainability is yet to be investigated.

As the first step, this research investigates whether developers' involvement in community smells relate to software maintainability in terms of their contribution and bug introduction. For this purpose, a prevalent community smell named missing link smell is considered, which occurs when two developers contribute to the same source code without mutual communication. For analysis, open-source projects from GitHub are used to identify the involved developers in missing link. From these, the relationship between the number of contributions and the number of involvements in missing link of a developer is explored. The correlation is measured between the number of commits from involved developers in missing link and Fix-Inducing Changes (FIC), changes that introduce bugs. Both (i) the contribution of developers and the involvement in missing link and (ii) the number of commits from involved developers and the number of FIC commits, are found

positively correlated. Furthermore, bugs introduced by developers involved in missing link are mostly major type in terms of severity.

Based on these results, an empirical study is conducted to investigate whether the maintainability differs between classes affected by community smells and those which are not. After identifying all community smells along with involved developers by communication and collaboration analysis, classes modified by any of those developers are categorized as smelly classes otherwise non-smelly. To assess maintainability, change-proneness, fault-proneness, code smells and five quality attributes of maintainability, such as modularity, reusability, analyzability, modifiability, and testability are considered. These metrics are computed by analysing project artifacts such as source code, commits, issue reports, etc.

The distributions of these metrics are compared between smelly and non-smelly classes using statistical tools. The result shows that smelly classes are about 15 times more change-prone, 19 times more fault-prone as well as 1.7 times more likely to contain code smells than non-smelly classes. In terms of object-oriented metrics, smelly classes are 56% more complex, 37% more coupled, and 28% less documented on average. The findings suggest that community smells have an adverse impact on software maintainability. A smelly class is less maintainable than a non-smelly class and thus requires special attention.

Acknowledgments

"*All praises are due to Allah alone*"

First of all, I am grateful to Almighty Allah for giving me the opportunity and granting me the ability to complete my research work properly.

I would like to express my heartfelt gratitude and respect to my supervisor, Professor Dr. Kazi Muheymin-Us-Sakib, Institute of Information Technology, University of Dhaka, for his constant support, guidance and inspiration. His support and help have kept me motivated throughout this thesis.

I am thankful to the faculty and thesis committee members of Institute of Information Technology, University of Dhaka for their valuable feedback and suggestions, which have helped me to improve my thesis.

I am greatly indebted to my parents who have always supported and motivated me to reach my goal. I would also like to appreciate my classmates for their support. I extend my gratefulness towards the members of Distributed Systems and Software Engineering (DSSE) research group for their knowledgeable insights and constructive criticism on my work.

I am also thankful to the Ministry of Posts, Telecommunications and Information Technology, Government of the Peoples Republic of Bangladesh for granting me ICT Fellowship No - 56.00.0000.028.33.006.20-84; Dated 13.04.2021. Last but not least, my special thanks to the Bangladesh Research and Education Network (BdREN) for providing me with virtual machine facilities to carry out my thesis.

List of Publications

1. "Understanding the Involvement of Developers in Missing Link Community Smell: An exploratory Study on Apache Projects" in *Proceedings of the 8th International Workshop on Quantitative Approaches to Software Quality co-located with APSEC*, pp. 64-70, 2020.

2. "Understanding the Relationship between Missing Link Community Smell and Fix-inducing Changes" in *Proceedings of the 16th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE)*, pp. 469-475, 2021.

3. "Impact of Community Smells on Software Maintainability" in *16th International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2022. (Submitted)

Contents

List of Tables

List of Figures

Chapter 1

Introduction

A software product can be thought of a combined effort of members who are involved with its development. These members form a community which can be defined as a software development community. In the development community, developers communicate about the software development in the defined communication channel such as mailing list. For collaboration in the source code, they use version control system like *GitHub*. While developing a software, communication and collaboration issues can arise among developers. These may lead to the unforeseen project cost which is known as social debt [1]. These social and organizational anti-patterns that have the potential of emerging the social debt are defined as community smells [2]. For example, when a developer takes decisions regardless of the opinion or suggestion from his peers, it is called missing link smell. In this situation, developers work in the same source code but do not communicate with each other. When such developers leave, it will be difficult to accumulate required knowledge for a new developer to maintain that source code.

This chapter presents the motivation of the research. From this, the research questions are formulated and how these questions can be addressed are discussed. Next, the contribution and achievement of this work are described. Lastly, the organization of this thesis is provided as a guideline to the readers.

1.1 Motivation

In recent times, community smell related studies have gained a lot of attention in software engineering research [3, 4, 5, 6]. This field of study incorporates organizational and social aspects of the software development community with technical aspects. The way in which developers interact with source code not only depends on technical factors but also on inter-personal issues [4]. Thus, software maintainability which is defined as the modification capability of a software [7], may be affected when there are gaps in communication and collaboration among developers.

Community smells are perceived harmful by developers for software development and evolution [3]. Although these smells may not be an immediate obstacle for software development, it can create problems in the long run. For example, the knowledge gap created among developers due to community smells can make it difficult to maintain the source code. Sometimes, to avoid community related problems, developers prefer not to do work like refactoring code smells which can improve maintainability [4].

As maintenance is a major part of the Software Development Life Cycle (SDLC) and consumes more than half of the project costs [8], keeping the source code of a software maintainable is necessary. For this reason, understanding how and to which extent community smells have impact on software maintainability is important for ensuring better maintainability. To identify the impact, community smells needs to be related with metrics that represents the maintainability of the source code. Therefore, observing the maintainability of source code that are affected by community smells can yield an understanding of the impact of community smells on software maintainability.

Existing researches have been studying community smells from different perspectives, such as definition [1, 2], detection [3, 9] and impact analysis [4, 10].

Followed by the studies of definition, researchers have studied how to detect those from project artifacts [3, 9, 11]. A few studies established the relationship between community smells and technical factors by predicting code smell intensity [4] and bug [10] from community smells. The impact of community smells on software maintainability as a whole is yet to be investigated. This thesis aims at investigating the impact by relating developers' involvement in community smell with their contribution and bug introduction at first. Then, a detailed empirical investigation is conducted by exploring how maintainability metrics differ in the presence and absence of community smells.

1.2 Research Questions

The following research questions are formulated based on the above discussions:

- **RQ**: *How do community smells impact software maintainability?*

  This research question will be addressed by answering the following sub-research questions:

  - **SQ1**: *How does developers' involvement in community smells relate to their contribution in the project?*

    For answering this question, the involvement of developers in missing link smells need to be identified by detecting this smell in the community. For assessing the contribution of a developer, a measurement has to be defined. To identify the impact, how the number of involvements in missing link smell relate with the number of contribution can be investigated. The quality of contribution, that is, whether the contribution introduces bugs needs to be verified. To identify this relationship, the number of contribution of involved developers and the number of their bug introductions can be analysed.

– **SQ2**: *How do software maintainability metrics differ between community smells affected and unaffected classes?*

To address this question, community smells need to be detected first. Then, the measurement for software maintainability has to be defined. Next, the relationship needs to be established between community smells and maintainability metrics. The values of maintainability metrics have to be calculated from the source code for both affected and unaffected software artifact, such as class. To determine which classes are affected by community smells, the contribution activities of developers involved in community smells can be analysed. The change histories from version control system can be used in this case [12]. To understand whether the maintainability differs in the presence and absence of community smells, the value of maintainability metrics need to be compared using statistical analysis.

1.3 Contribution and Achievement

This research investigates the impact of community smells on software maintainability through developer contribution, bug introduction and maintainability metrics. There are two major contributions of this work. The first contribution is that it finds how developers' involvement in community smells is related with software maintainability, such as bug introduction and severity of bugs. The second contribution is that it identifies classes affected by community smells are less maintainable compared to those that are not. The overall contributions of this research are summarised as following:

1. **Developers' involvement in missing link smell negatively affects software maintainability**: Missing link community smells are detected from project repositories and mailing lists. Developers are divided into two

categories, such as smelly and non-smelly developers. The developers involved in any missing link smell are identified as smelly developers, otherwise considered as non-smelly developers. The contribution of developers is measured as the number of commits. Then, the correlation between the number of involvements in smell and the number of contribution is analysed. The percentage of developers who are involved in missing link smells is also computed. Fix-Inducing Changes (FIC), changes that introduce bugs [13], are used as the measure to examine whether developers involved in missing link introduce bugs in the system. Furthermore, the correlation analysis is performed between the number of smelly commits and the number of FIC commits. The severity of bugs introduced by smelly developers are analysed from bug repository.

The number of contribution is found positively correlated with the number of involvements in missing link smells. It indicates that a developer who contributes more in a project tends to have more missing link smells. The number of FIC commits is also found positively correlated with the number of smelly commits. It is evident that the number of bug introduction tends to increase with the increase of involvements in missing link smell. Moreover, the bugs introduced by these smelly developers are severe enough to cause major type of functionality loss in the project. This is inferred that involvement of developers in missing link smell can affect software maintainability negatively in the form of bug introduction. Chapter 4 describes this study in detail.

2. **Classes affected by community smells are less maintainable than unaffected classes**: Community smells are identified along with involved developers analysing source code repository and mailing list archive. Based on the involvement in these smells, developers are divided into smelly and

non-smelly developers. Classes modified by smelly developers are identified as smelly and non-smelly otherwise. The maintainability of these classes are assessed using change-proneness, fault-proneness, code smells and ISO/IEC 25010 defined five quality attributes, such as modularity, reusability, analyzability, modifiability, and testability. To identify these quality attributes, object-oriented metrics, such as complexity, coupling, etc., are used. Then, the distribution of these metrics are compared using statistical analysis.

Smelly classes are found about 15 times more change-prone and 19 times more fault-prone than non-smelly classes. These classes are 1.7 times more likely to contain code smells compared to non-smelly classes. In terms of object-oriented metrics, smelly classes are 56% more complex, 37% more coupled, and 28% less documented on average. This is inferred that community smells have negative impact on software maintainability. A class affected by community smell is less maintainable than a non-smelly class in terms of maintainability metrics. The details of this study are presented in Chapter 5.

1.4 Organization of the Thesis

This section provides an overview of the subsequent chapters. The chapters are organized in the following way:

- **Chapter 2 Background Study:** The knowledge base is provided in this chapter for understanding community smells and software maintainability. The chapter starts with describing software development community and related terminologies, such as communication and collaboration network. The different types of community smells are presented with examples. Next, software maintainability metrics are discussed. At the end, the different stages of an empirical research are described.

- **Chapter 3 Literature Review of Community Smells:** This chapter presents the existing literature related to community smells and software maintainability metrics. Studies related to community smells are categorized into three categories namely definition, detection and impact of community smells. Next, maintainability metrics related researches are discussed, followed by the summary of the literature review.

- **Chapter 4 Developers' Involvement in Missing Link Community Smell:** This chapter presents how developers' involvement in missing link smell impact software maintainability in the form of their number of contribution and bug introduction. At first, the methodology of the study is described. Next, the dataset description and result analysis are provided. Followed by this discussion, the threats to validity of the study are presented. Finally, the chapter ends with the summary of the study.

- **Chapter 5 Relationship Between Community Smell and Software Maintainability:** In this chapter, an empirical study is presented to identify the impact of community smells on different software maintainability metrics, such as change-proneness, fault-proneness etc. Firstly, the methodology includes the working steps of the study. Next, the description of the dataset is provided, followed by the results and discussions. After that, the threats to validity of the study are described. The summary of the study is provided at the end of the chapter.

- **Chapter 6 Conclusion:** In this chapter, the whole thesis is summarized as well as the future work is presented.

Chapter 2

Background Study

This chapter provides basic terminologies to better understand the concepts of community smells and software maintainability. As community smells occur in the software development community, the different types of network in the community are described first. Then, the definitions of community smells are provided with the identification approaches from these developer networks. Next, the maintainability metrics are described. Lastly, the process of an empirical study is presented to better understand the research design of this thesis.

2.1 Software Development Community

The relationship and interaction between developers in a software development community can be modeled through the network. The different types of network in a development community are discussed below:

Developer Social Network (DSN): Developer Social Network (DSN) is the network of a software development community where a node represents a developer and an edge represents the relationship between two developers [14]. The relationship can be formed considering their communication, coordination, etc. An example of DSN is illustrated in Figure 2.1. The upper part of the graph represents communication and the lower part represents the collaboration among

developers. The developers are connected with a solid line if they communicate with each other. The developers are connected to the file icon through a dashed line if they contribute to that source code file.

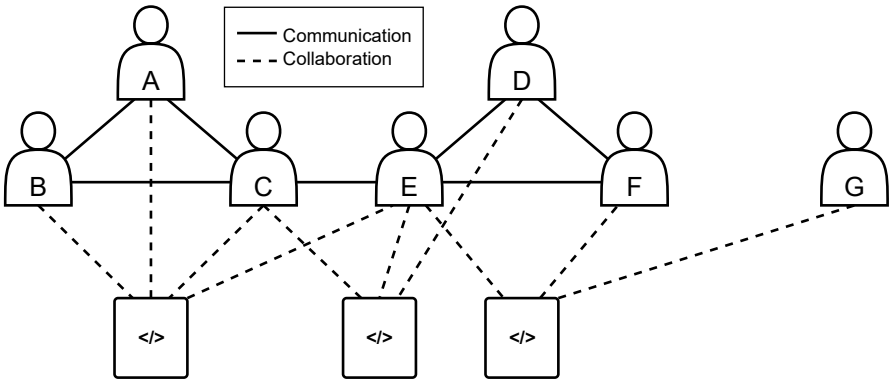

Figure 2.1: Developer Social Network (DSN)

Collaboration Network: A specific type of DSN which indicates the collaboration in a development community. Here, a node represents a developer who contributes to the project in the version control system. Two developers are connected through an edge if they contribute to the same part of the source code within a given time frame [9]. Figure 2.2 represents an example of a collaboration network. The technical structure of the software project is illustrated by this network.

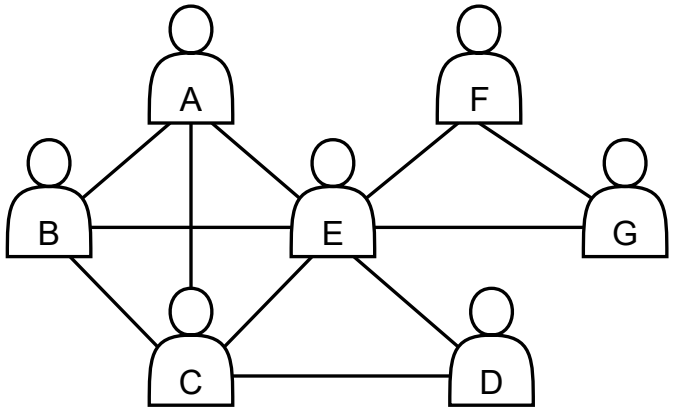

Figure 2.2: Collaboration network

Communication Network: A specific type of DSN which indicates the communication within the defined communication channel of a development commu-

nity. Here, nodes represent developers who communicate in the defined communication channel such as mailing list. Two developers are connected through an edge if they replied in the same e-mail within a given time frame [9]. A communication network is illustrated in Figure 2.3. It depicts the organizational structure of the development community in terms of its communication activities.

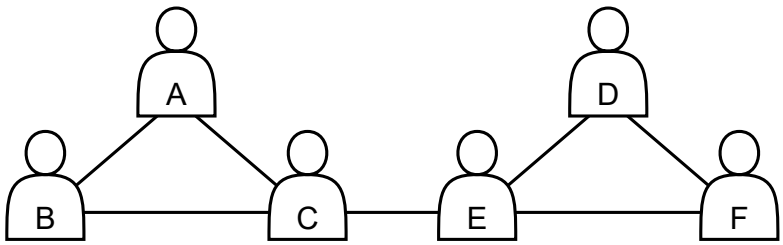

Figure 2.3: Communication network

2.2 Community Smells

The social and organizational anti-patterns in the software development community that lead to the emergence of social debt are known as community smells. The description and identification patterns of four community smells, considered in this study, are given below:

1. **Organisational Silo**: This smell occurs when there are siloed areas in the software development community that do not communicate with other members through the defined communication channel such as mailing list [3]. To detect this smell, developers who have collaboration without any communication need to be identified.

   This smell can be identified by looking for developers who are present in the collaboration network but missing in the communication network. For example, the developer named G is present in the collaboration network (Figure 2.2) but absent in the communication network (Figure 2.3). Therefore, it is an instance of organizational silo where developer G is involved.

2. **Lone Wolf or Missing Link**: This smell refers to the presence of such developers who carry out their work without communicating their peers. The situation arises when developers contribute to the same source code but do not communicate with each other [3]. This smell can be identified by detecting collaboration between two developers that do not have the communication counterpart in defined communication channel, for example, development mailing list [9].

   By comparing the collaboration network with the communication network, the instances of missing link can be detected. In Figure 2.2, there is a link between developer *C* and *D* in the collaboration network. On the other hand, there is no corresponding link between these two developers in the communication network (Figure 2.3). Developer *C* and *D* are collaborating on the same part of source code but they are not connected through any communication link. Thus, this is considered as an instance of a missing link between developer *C* and *D*.

3. **Radio Silence or Bottleneck**: This smell represents the situation when one member of the community is involved in every formal interaction across two or more sub-communities. Thus, there is little or no flexibility to introduce other parallel communication channels between remaining members of the sub-communities [4].

   Figure 2.3 shows that there exist two sub-communities which are sub-community *A-B-C* and *D-E-F*. *A-B-C* depends on the developer identified by letter *C* for communication with *D-E-F*. On the other hand, *D-E-F* communicates with *A-B-C* only through developer *E*. Thus, these incidents are identified as the occurrences of radio silence or bottleneck effects and both developers *C* and *E* are involved in these smells.

4. **Black Cloud**: This smell denotes the information overload due to lack of structured communication such as lack of periodic and official opportunities, for example, daily stand-ups to exchange knowledge among community members [4].

   There are two sub-communities present in the developer social network as shown in Figure 2.3, sub-community *A-B-C* and *D-E-F*. In these two sub-communities, developers denoted by letter C and E presents a unique connection who are responsible for knowledge exchanging between these sub-communities. Thus, this is identified as a potential black cloud and will represent an effective instance of black cloud if the same incident occurs time to time.

2.3 Software Maintainability

Software maintainability is the capability of software systems to be modified for bug-fixing, improvements or adaption to changed environment, etc. [7]. In the existing literature, maintainability is measured using change-proneness, fault-proneness, code smell and *ISO/IEC 25010* defined five quality attributes such as *modularity*, *reusability*, *analyzability*, *modifiability*, and *testability* [15, 16, 17, 18, 19]. The definitions of these metrics are described below:

- **Change-proneness**: Change-proneness refers to the extent of change carried out on a software artifact across releases and thus a useful metrics for maintainability. A software artifact that changes frequently is more difficult to maintain [20].

- **Fault-proneness**: Fault-proneness means the extent to which a software artifact is prone to faults. Fault-prone software artifacts have less maintainability as these have to go through frequent bug-fixing [21].

- **Code Smell**: Code smell denotes the poor implementation choices applied by developers. The maintainability of source code decreases, if it contains code smells [22, 23]. This is because code smell needs refactoring during the software evolution [24].

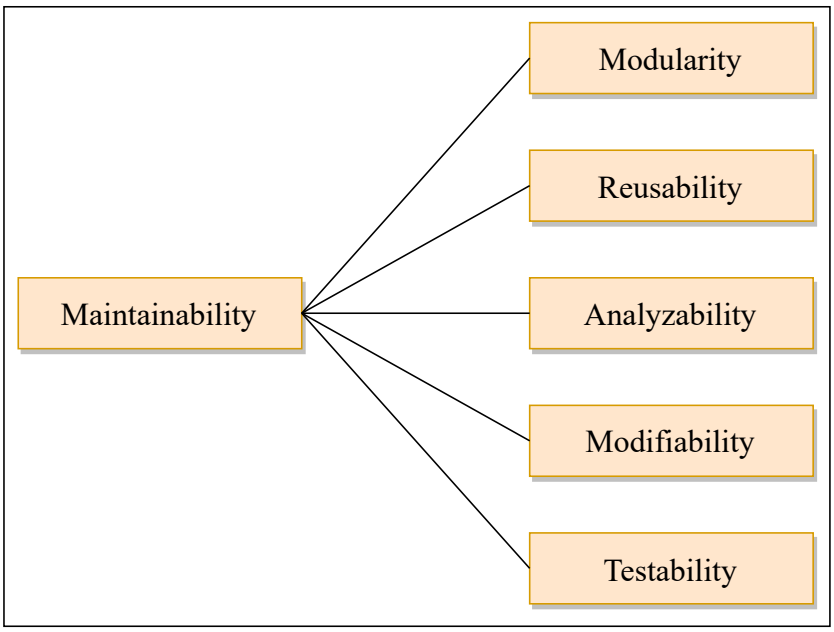

Figure 2.4: ISO/IEC 25010 defined quality attributes of maintainability

The quality attributes of maintainability defined by *ISO/IEC 25010* are illustrated in Figure 2.4. The definitions of these attributes are given below:

1. **Modularity**: The modularity refers to the degree to which a software is composed of distinct components. Thus, the system has minimal impact on others while changing one component [25].

2. **Reusability**: Reusability means the extent to which an artifact can be used in more than one software system [25]. That means whether the artifact can be utilized in the same or the other system.

3. **Analyzability**: Analyzability implies the degree to which it is possible to effectively assess the impact on intended change, or to identify the cause of failure, or to identify parts to be modified [26].

4. **Modifiability**: Modifiability refers to the degree of effectiveness and efficiency of a software system to be changed without introducing bugs [27] or degrading the current quality [25].

5. **Testability**: Testability assesses the degree to which the test criteria defined for the system is effective and efficient [25].

All these five quality attributes have the positive relationships with software maintainability. For example, the more modular a software system is, the easier it is to maintain.

2.4 Empirical Research

Empirical researches have taken an important role in the field of software engineering. It is essential for the software engineering domain, as it makes possible to incorporate human perspectives into the research [28]. Empirical Software Engineering (ESE) researches analyze software artifacts using qualitative and quantitative approaches with a view to describing, explaining, evaluating, maintaining, and monitoring these artifacts [29].

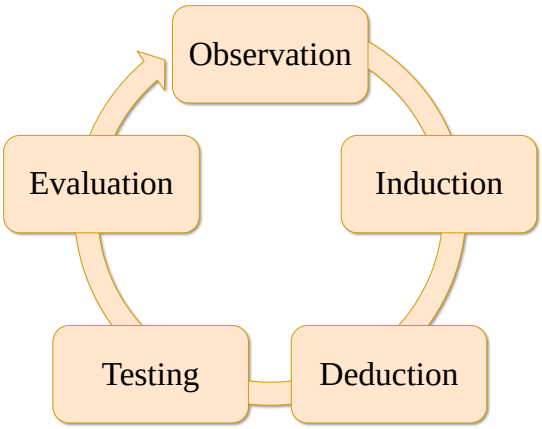

Figure 2.5: Empirical research cycle

According to A.D. de Groot, an empirical research has to go through a cycle which has five stages [30]. These five stages are shown in Figure 2.5. The description of each stage is provided below:

1. **Observation**: The specific phenomenon is observed and concerning causes are inquired in this stage. During this initial stage, an idea is triggered to form hypothesis. Observations generally come from previous research findings.

2. **Induction**: Induction is then used to form a hypothesis. In this stage, a general probable conclusion is made by inductive reasoning based on the observation. This rule or hypothesis is not necessarily true. It can be false which will be tested in the subsequent stages of the cycle.

3. **Deduction**: This stage involves the formulation of experiment that will test the hypothesis. Empirical studies can be broadly divided into two categories which are qualitative and quantitative. In the software engineering domain, quantitative method is the most widely used scientific method [31]. Quantitative research generates numerical data for analysis and applies mathematical or statistical methods to accept or reject hypothesis.

4. **Testing**: In this stage, the hypothesis is tested using statistical methods on the collected data. Based on the nature of the data, appropriate statistical tests are selected.

5. **Evaluation**: This is the final stage in an empirical research. The experiment and tests that are performed in the previous stages are evaluated in this stage. The observed results are interpreted and discussed with justifications. Finally, the conclusion is derived based on the findings of the research with supporting arguments.

2.5 Summary

In this chapter, the knowledge base is created for understanding community smells and software maintainability. First, the software development community is described with associated terms such as developer network, communication and collaboration network. Next, the definitions of different types of community smells are provided and how these smells occur is illustrated with a sample development community. After that, the terminologies related to software maintainability metrics such as change-proneness, fault-proneness, etc. are presented. Based on the concepts presented in this chapter, the existing literature related to community smells and software maintainability will be reviewed in the next chapter.

Chapter 3

Literature Review on Community Smells

In recent times, community smell related studies have gained attention in software engineering research. The objective is incorporating the organizational and social aspects of the software development community. From the literature review, studies related to community smells are divided into three categories such as definition of community smells, detection and prediction of community smells and impact of community smells on software artifacts. Some studies [1, 2] focused on defining different types of community smells while others focused on detecting these smells in open-source projects [3, 9]. Besides, a few studies [32, 33, 34] focused on community smells prediction. Moreover, the relationship and the impact of community smells on different software artifacts, such as code smell and bug, have also been studied by the research community [4, 10].

The literature related to community smells based on the above categories and literature related to software maintainability metrics are discussed in the following sections.

3.1 Definition of Community Smell

The concept of community smell, also called organizational and social anti-pattern, is first introduced by Tamburri et al. [2]. The authors conducted a survey-based qualitative study in a large software company to identify the anti-pattern that causes social debt that is unforeseen project costs. They observed the development scenario over the period of six months. In this time, the authors conducted several interviews to collect data from the employees. They analysed the collected data using Grounded Theory [35] to identify core concepts that revolve around social debt. Authors defined a social debt framework depicting these concepts and their role in social debt. To represent the causality, they applied an empirical causal model named 6C model [2]. Based on the analysis, authors recognized nine such circumstances and defined these as community smells. These are *Organizational Silo Effect*, *Black Cloud Effect*, *Prima-donnas Effect*, *Leftover-techie Effect*, *Sharing Villainy*, *Organisational Skirmish*, *Architecture Hood Effect*, *Solution Defiance* and *Radio Silence*. The short descriptions of these smells are given below:

1. *Organizational Silo effect:* is the existence of disconnected regions of the software developer community that do not communicate.

2. *Black-cloud effect:* represents the lack of people who can minimize the knowledge and experience gap between community members as well as the absence of effective knowledge sharing for example regular stand-up meetings.

3. *Prima-donnas effect:* is the existence of developers who show egotistical behavior and are not supportive to other developers.

4. *Leftover-techie effect:* occurs when mistrust is created among the technicians in the development community as a result of increased isolation.

5. *Sharing villainy:* refers to the situation when outdated, unverified or misinformation are spread by developers.

6. *Organisational Skirmish:* occurs when the organizational culture is not aligned between the development and operation unit.

7. *Architecture hood effect:* implies the situation when nobody take responsibility to lead the implementation according to the decision of architectural board.

8. *Solution defiance:* is the presence of divided groups of developers who have the opposite viewpoints regarding socio-technical or technical decisions.

9. *Radio-silence:* occurs when one member is present in every formal engagement among two or more sub-communities. Thus, the flexibility to create other channel for communication among the remaining members becomes difficult.

The authors also identified six recurrent decisions that helped in reducing community smells. These decisions are suggested as mitigation techniques to avoid the negative effects of community smells. The suggested mitigation techniques are as follows:

- Full-circle to mitigate leftover-techie
- Learning Community to mitigate radio-silence
- Culture conveyors to mitigate prima-donnas and sharing villainy
- Stand-up voting to architecture-hood
- Community-based contingency planning to mitigate prima-donnas and solution defiance
- Social wiki to mitigate prima-donnas, solution defiance, black-cloud effect, sharing villainy and organisational silo effect

3.2 Detection of Community Smell

Tamburri et al. [3] defined the identification pattern of four community smells which are *Organizational Silo*, *Lone Wolf or Missing Link*, *Bottleneck or Radio Silence*, and *Black Cloud*.

1. *Organizational Silo:* Organisational Silo Effect is identified by looking for community members who collaborate in the source code with other developers but do not communicate within the analysed communication channel, for example, mailing list.

2. *Lone Wolf or Missing Link:* Missing Link Community Smell is identified by detecting a pair of developers that do not have communication but they collaborate in the same source code.

3. *Bottleneck or Radio Silence:* Black-cloud Effect Community Smell is detected by on the identifying sub-communities that communicate with the a unique communication link in consecutive time periods.

4. *Black Cloud:* Radio Silence Community Smell is detected by looking for community members who act as a unique knowledge and information exchangers for different sub-communities.

The authors conducted an empirical study on 60 open-source projects and found that community smells are highly diffused in these projects. The study also investigated how developers perceive about community smells through interview. The developers involved in these projects recognized community smells as relevant problems for software evolution. They also identified the relationship between community smells and socio-technical factors such as socio-technical congruence, turnover, truck factor, etc. A number of socio-technical indicators, for example, socio-technical congruence, are found to be correlated with community smells. These indicators can be used to monitor community health and to avoid possible

emergence of social debt. In their study, the authors developed an open-source tool named *Codeface4Smells*[1] to detect community smells from the project repository and mailing list. The availability of such a tool enabled the research community to further investigate the impact of community smells.

Besides detection, a few studies [32, 33, 34, 36] tried to predict the community smells. Palomba et al. [32] built a model to predict the emergence of community smells using socio-technical metrics. Almarimi et al. [33] proposed a model to predict community smells using Ensemble Classifier Chain (ECC) and Genetic Programming (GP) techniques. Huang et al. [37] built a community smells prediction model on individual developers using their sentiment.

3.3 Impact of Community Smell

Palomba et al. [4] first investigated the relationship between community smells and code smells. The authors conducted an empirical study on 117 releases of 9 open-source projects using mixed-method (qualitative and quantitative) approach [38]. They surveyed 162 concerned developers to investigate their perception on the relationship between community smells and code smells. They found that community smells can influence the maintenance decision such as refactoring code smells. In several cases, developers preferred to keep code smells rather than dealing with community smells. They also investigated the impact of community smells on code smell intensity. They proposed a code smell intensity prediction model using community smells which performed better than a model that did not consider community-related information. The study provided the empirical evidence that community-related circumstances can affect the way developers act in the source code, and community smells should be taken into account while studying the software maintainability.

[1] http://siemens.github.io/codeface

Eken et al. [10] investigated the effect of community smells on bug prediction. They conducted an empirical analysis on 10 open-source projects to examine whether community smells contribute in predicting bug-prone classes. The authors built seven different bug prediction models such as a baseline model including the state-of-the-art metrics (code churn, number of developers etc.). Three models including community smells, code smells and code smell intensity individually into the baseline model and three other models including the combination of these smell related metrics. The result shows that the baseline model is improved up to 3% in terms of AUC by community smells while code smell intensity improves it by up to 40%. The authors compared their community-aware bug prediction model with the state-of-the-art models that consider code smell, code smell intensity and other process metrics such as code churn, number of developers etc. They found community smells and code smells are the good indicators in predicting bug-prone classes by revealing communication and collaboration flaws in software development teams. The finding of the study implies that most of the information about technical flaws are captured by code smell intensity while predicting bugs. Besides, incorporating community smells can contribute in bug prediction by providing information about communication and collaboration flaws. This indicates that social aspects need to be considered with technical factors while studying the software system.

Catolino et al. [5] analysed the role of gender diversity and women's participation in community smells. They compared the distribution of community smells between gender-diverse team and non-gender-diverse team. To examine how the presence of women influences the number of community smells, a statistical model is built by relating community smells with the measure of gender-diversity. They used the Blau-Index [39] to measure the diversity. The finding of the study shows that gender-diverse teams have significantly fewer community smells compared to non-gender-diverse teams. The result reveals that gender diversity and women par-

ticipation are significant factors for community smells and the presence of women in teams can reduce the number of community smells.

Later, a survey based empirical study is conducted [40] to explore strategies adopted by developers to deal with community smells. They inquired 76 developers about four community smells and their action in removing these smells. Based on the collected data, authors suggested some refactoring strategies to deal with community smells such as mentoring, creating communication plan and restructuring the development community.

In another study, Catolino et al. [6] investigated how the variability of community smells, that is, the increase or decrease of community smells, is related to socio-technical metrics. They built a statistical model on the dataset of 60 open-source projects containing four types of community smells such as Organizational Silo, Black Cloud, Lone Wolf, and Radio Silence and 40 socio-technical metrics such as turnover, communicability, truck factor, etc. The results of the study report that communicability metrics are the most important factors to reduce the emergence of community smells. On the other hand, increasing collaboration network is not always effective to decrease community smells.

3.4 Software Maintainability Metrics

Software maintainability is the capability of software systems to be modified for bug-fixing, improvements or adaption to changed environment etc. [7]. Palomba et al. [15] investigated the impact of code smells on software maintainability. The authors conducted an empirical study on 395 releases of 30 open-source projects. They analysed the diffuseness of code smell in these projects. The authors identified classes that are affected by code smells. In this case, they considered 13 well known code smells. The maintainability of classes is measured in terms of change-proneness and fault-proneness of the class. This study explored whether

classes affected by code smells exhibit significant difference in change and fault-proneness. Most of the analysed code smells are found highly diffused in the evaluated projects. Classes that are affected by code smells show statistically significant higher change and fault-proneness than classes those are not affected.

Zhang et al. [19] assessed software maintainability based on 39 code metrics. There are five quality attributes of software maintainability defined in *ISO/IEC 25010*, such as *modularity*, *reusability*, *analyzability*, *modifiability*, and *testability*. In this study, 39 class level metrics related to these attributes were considered to measure different aspects of maintainability. These metrics were further grouped into six groups based on the work of Kontogiannis et al. [41] which are *complexity*, *coupling*, *cohesion*, *abstraction*, *encapsulation*, and *documentation*. These categories are related with five quality attributes of maintainability, for example, low complexity of a class indicates high analyzability and modifiability, low coupling improves analyzability and reusability, high cohesion increases modularity and modifiability, high abstraction enhances reusability, high encapsulation implies high modularity, and documentation might contribute to analyzability, modifiability, and reusability.

3.5 Summary

This chapter discusses the existing studies related to community smells and software maintainability metrics. Community smells have been explored from different perspectives by the research community. These can be categorized into definition of community smells, detection and prediction of community smells, and impact analysis of community smells. Some researchers [1, 2] worked on defining different types of community smells while others focused on detecting these smells [3, 9]. Besides, separate studies [32, 33, 34] explored how community smells can be predicted from socio-technical metrics. The prediction of community smell on

individual developers based on their sentiment has also been studied [37]. These studies have improved the understanding of community smells and encouraged further research in this field to investigate the impact of community smells. A few works established the relationships between community smells and some technical factors such as code smell intensity and bug [3, 4, 10]. These studies provided the empirical evidence that community-related circumstances can affect the way developers act in the source code, and community smells should be taken into account while studying the software systems. On the other hand, to asses software maintainability, existing literature used different metrics such as change-proneness and fault-proneness [15]. Some other studies used object-oriented metrics to identify the quality attributes of maintainability [19, 41]. Although the impact analysis on some specific technical factors has been studied so far, the relationship between community smells and software maintainability is yet to be investigated. In the next chapter, an empirical study is presented on how developers' involvement in community smell impact software maintainability.

Chapter 4

Developers' Involvement in Missing Link Community Smell

Community smells can be defined as organizational and social anti-patterns in a development community. This can affect software maintenance by the means of lacking mutual awareness, mistrust and knowledge gap among developers. Existing studies have observed the relationship of community smell with different socio-technical factors such as socio-technical congruence, turnover, etc. This chapter focuses on how many developers are involved in community smells such as missing link and how this involvement impact software maintainability in the form of their contribution and bug introduction in the project. To do so, missing links and developers who are involved in these smells are detected. From this, the percentage of developers involved in community smells are identified. A correlation analysis is performed between the number of contributions and the number of involvements in missing link smells of the developer. To understand the quality of contribution, the relationship between smell and Fix-Inducing Changes (FIC), changes that introduce bugs, is investigated. It is observed that the percentage of smelly developers involved with missing link smell is 8.7% on average. The result also suggests a moderate positive correlation between the contribution of a

developer and the involvement in missing link smell. The number of FIC commits is found positively correlated with the number of smells. Furthermore, it is found that bugs introduced in smelly commits are mostly *Major* type in terms of severity which denotes that community smell has an impact on maintenance.

4.1 Introduction

Community smells are the organizational and social anti-patterns in a development community [2]. These may lead to the emergence of social debt which indicates unforeseen project costs connected to a sub-optimal software development community. Although community smells may not be an immediate obstacle for software development, these have the potential to affect software maintenance negatively in the long run [4]. Missing link is one of the most commonly reported community smells which occurs if developers do not communicate with each other while working collaboratively [9].

Missing link community smell implies the lack of communication among developers that can create knowledge gap in the community [3]. A software product can be thought of as the combined effort of all developers. So, the lack of communication and cooperation can negatively affect mutual awareness and trust among developers [9]. It is important to know how many developers are involved in missing link as they may affect the whole project. Analyzing the characteristics of these developers will help the project managers to take steps such as task reassigning, team reformation, increasing awareness, etc., to keep communication issues lower among the developers. Besides, previous studies found that community smells including missing link smell are related to code smells [11] and have an impact on code smell intensity [4, 42]. Since code smells are found to be successful indicators of maintainability in the form of bugs in software systems [15, 17], the relationship between community smells and bugs needs to be investigated.

In previous studies, the definition and detection of missing link smell in open-source projects have been studied. A few studies have explored the impact of missing link smell on different software artifacts such as code smell. Magnoni proposed the identification pattern of missing link community smell [9]. Tamburri et al. examined the relationship between community smells and different socio-technical factors, for example, socio-technical congruence, turnover, etc. [3]. They considered missing link, organizational silo, black cloud and radio silence community smell. Palomba et al. investigated the impact of missing link smell and four other community smells on code smell intensity [4]. Catolino et al. analyzed the role of four community smells including missing link smell on gender diversity and women's participation in open-source communities [5]. However, developers' involvement in missing link smell and how their contributions are affected by missing link smell have not been analyzed yet.

In this context, the current study aims to identify how many developers are involved in missing link smell and whether their involvement impact the maintenance in the form of contributions and bug introduction. To do so, the relationship between missing link smell and Fix-Inducing Changes (changes that introduce error into the system) are investigated. For analysis, seven diverse and open-source projects such as ActiveMQ and Cassandra are selected based on several criteria (for example, availability of developer mailing list). First, missing link smells are identified in each project finding cases where a collaboration link does not have its communication counterpart. Then, the developers involved with each smell are identified by extracting the instance of smell. Commits that have been submitted by developers involved in missing link smell are marked as smelly commits. The percentage of developers involved with missing link smell is calculated to check whether a subset of developers is involved with this type of smell. Then the correlation is investigated between the contribution of developers and their involvement in missing link smells. Commits that represent FIC are identified by analyzing

the project repository. Finally, a correlation analysis is performed between the number of smelly commits and FIC commits using Spearman's rank correlation coefficient [43]. To understand the severity of bugs that are introduced by developers who are involved in missing link smell, FIC commits that are submitted by smelly developers have been linked to the bug repository. After linking FIC to the bug repository, the information of severity is extracted from the bug report.

The results of the study show that a small part of the total developers is involved with missing link community smell. On average, 8.7% of the total developers of a project are involved with missing link smell. This study also finds a significant moderate positive correlation between the developer contribution and their involvement in missing link smell. The result of the study shows that there is a significant positive correlation between the number of smelly commits and FIC commits. The study also finds that bugs occurring in smelly commits are related to major loss of functionality. In the following sections, the details of the study are presented.

4.2 Methodology

This study aims at understanding how many developers of a project are involved in missing link smell. This study also wants to assess the relationship between a developer's contribution and involvement in missing link smell. To understand the quality of contribution, the relationship between missing link smell and bug introduction is also investigated in software projects. First, the missing link smell is detected for all the selected projects. Then, the percentage of smelly developers is retrieved for each project. Later, correlation analysis is performed between a developer's contribution and involvement in missing link smell. Next, the Fix-Inducing Changes (FIC) are identified from the project repository by finding erroneous code changes that induce a fix later. Finally, the relation between smelly commits and

FIC commits are analysed. The details of each step are provided in the following subsection.

4.2.1 Missing Link Smell Detection

The first step is to identify missing link smell from source code repository and mailing list. To detect missing link smells, a temporal window needs to be fixed as software community changes over time. Six-month analysis window is used in this case so that substantial changes can be found to analyse. For every analysis window of a project, a communication network is built for that window by examining the mailing list of the project. All messages in the mailing list of a project are analysed and developers who replied in the same email within that window are connected in the network. Developer mailing list is considered for the source of communication as it is the main communication platform for an open-source project [44]. For example, the contribution guideline of a popular open-source project of *Apache*[1] states,

> *"Discussions at Apache happen on the mailing list".*

Next, a collaboration network is generated analysing the project's *GitHub* repository. All commits are analyzed and developers who contribute to the same part of source code within that window are connected through an edge. Thus, the communication network is constructed by extracting communication data from development mailing list and the collaboration network is generated by extracting collaboration data from the project repository. After having both of these networks, for each edge in the collaboration network, the corresponding communication part is searched in the communication network. Any edge that is present in the collaboration network but absent in the communication network is identified as missing link smell.

[1]https://mahout.apache.org/developers/how-to-contribute

An open-source tool, *Codeface4Smells* [45], is used to detect missing link community smell. Missing link smells are identified in the aforementioned way from project repository and development mailing list. The source code repository and mailing list archive are provided as input and a list of missing link instances for each window of the project are found as output. The developers involved with these smells will be identified in the next step.

4.2.2 Smelly Developers Identification

An instance of missing link smell consists of two collaborating developers who do not communicate with each other. Thus, for every missing link smell, there are two involved developers. A developer who is involved in any missing link smell is considered as a smelly developer. On the other hand, a non-smelly developer is one who is not involved in any missing link smell. The smelly developers of a project x can be denoted by a set SD_x. The number of smelly developers of the project will be the number of elements in SD_x.

To calculate the percentage of smelly developers in a project, the total number of developers of that project is required. The total number of developers is computed as the sum of the number of developers who are involved in either collaboration or communication network. That means the developers who contribute to the source code in the repository and who communicate on mailing list both are considered [9]. The percentage of smelly developers of a project is calculated using Equation 4.1.

$$percSD_x = \frac{numSD_x}{totalDev_x} \times 100\%, \tag{4.1}$$

where $numSD_x$ is the number of smelly developers in project x and $totalDev_x$ is the number of total developers in the same project.

4.2.3 Identifying Developers' Contribution

In open-source projects, commits are the most representative form of coding contribution [36]. So, the contribution of a developer in a project is measured by the number of commits of that developer in that repository. The number of commits of every individual developer is retrieved by analysing the source code repository using *git* command.

All the commits of a smelly developer are identified as smelly commits. For each smelly developers, the number of smelly commits are calculated analysing the change history. Along with these, how many times a developer involved in missing link smell are also computed. The number of contribution and the number of smelly commits are used in the next step to correlate with missing link smell.

4.2.4 Fix-Inducing Changes (FIC) Detection

To understand the bug introduction by developers involved in missing link, Fix-Inducing Changes (FICs) are used. FICs are the erroneous changes to the code that induce fixes in the future. It is used to identify the contribution quality of developers who are involved in missing link smells. To find FIC, the following steps are followed:

The first step of detecting FIC is finding changes that fix a bug, called the Fixing Changes (FC). To find the FCs, all commit messages are extracted from the project repository. Then, commit messages are searched for keywords - "Fix", "Bug", "Patch" including their past and gerund form. These commits indicate bug fixing activities and are labeled as FC commits. Next, changes made in each FC commit are extracted comparing with its immediate parent commit. *Diffj* tool [46] is used to obtain the location of changes, that is modified or deleted line numbers. The white space or other formatting changes are ignored so that the possibility of finding false FICs can be mitigated [47]. Finally, the origin of these

change locations is tracked using *git-blame*[2] command. This command is used to identify which commit is responsible for the latest changes made to a specific line of a file. This leads to the commit that introduces the bug that is FIC. The process of detecting FIC adopted in this study is similar to [48, 49]. Both FIC and the corresponding FC commits are stored for analysis in the subsequent step.

4.2.5 Correlation Analysis

After collecting required data according to the steps described above, the analysis is performed from three perspectives. First, the relationship is investigated between a developer's contribution and their involvement in missing link smell. Next, the correlation analysis is performed between the number of smelly commits and FICs. Finally, the severity of bugs that are introduced by smelly developers are analysed.

4.2.5.1 Developer's Contribution and Involvement in Missing Link

To identify the relationship between a developer's contribution and involvement in missing link smell, the correlation is performed between following two measures:

1. how many commits a developer has in the project repository
2. how many times a developer is involved in missing link smell

Both the number of commits and the number of involvement in smells of a developer can vary due to project and community size. So, these are converted into percentage to achieve the relative measurement. The commit percentage of a developer is calculated using Equation 4.2.

$$percentCommit = \frac{numCommit_i}{\sum_{i=1}^{n} numCommit_i} \times 100\% \tag{4.2}$$

[2]https://git-scm.com/docs/git-blame

where $numCommit_i$ is the number of commits of developer i and n is the total number of smelly developers.

Equation 4.3 is used to calculate the percentage of involvement in missing link smell for a developer.

$$percentMissingLink = \frac{numMissingLink_i}{\sum_{i=1}^{n} numMissingLink_i} \times 100\% \tag{4.3}$$

where $numMissingLink_i$ is the number of involvement in missing link smells of developer i and n is the total number of smelly developers.

Finally, the correlation analysis is performed between *percentCommit* and *percentMissingLink* for each project individually. Kendall's tau-b [50] is used to assess the degree of association between these two variables. Both *percentCommit* and *percentMissingLink* have tied values in the dataset. As Kendall's tau-b can handle tied ranks, this is used for the correlation analysis. The correlation coefficient is considered significant if the obtained p-value is less than 0.01, which is widely used in software engineering empirical researches [31]. The correlation coefficient is interpreted according to Table 4.1. The coefficient, τ_b, indicates the strength of the correlation. τ_b has a range of value from -1.0 to 1.0. As τ_b closes to 0, it indicates less correlation between two variables. As τ_b approaches to -1.0 or +1.0, the strength of correlation between two variables is increased. The positive value of τ_b indicates a positive correlation and the negative value of τ_b indicates a negative correlation between two variables.

Table 4.1: Kendall's tau-b correlation coefficient interpretation

Correlation Coefficient (Negative)	**Correlation Coefficient (Positive)**	**Interpretation**
$-0.4 < \tau_b \leq 0.0$	$0.0 \leq \tau_b < 0.4$	Weak
$-0.7 < \tau_b \leq -0.4$	$0.4 \leq \tau_b < 0.7$	Moderate
$-0.9 < \tau_b \leq -0.7$	$0.7 \leq \tau_b < 0.9$	Strong
$-1.0 \leq \tau_b \leq -0.9$	$0.9 \leq \tau_b \leq 1.0$	Very Strong

4.2.5.2 Number of Smelly Commits and FICs

To understand the direction of the relationship between the number of smelly commits and the number of FICs, correlation analysis is conducted. As a monotonic trend is observed between these two variables, Spearman's rank correlation [43] method is chosen. A monotonic trend implies both variables tend to increase together and decrease together, or the opposite, but not exactly at a constant rate like a linear relationship. In Spearman's rank correlation coefficient, the relationship between two variables can be assessed using a monotonic function. The interpretation of the correlation coefficient is adapted from [51] and displayed in Table 4.2. The correlation coefficient, ρ, indicates the strength of the correlation. The value -1 or +1 means a perfect relationship and 0 means no relationship between two variables. As the value approaches -1 or +1, it indicates more strong correlation. A value closer to 0 indicates a weaker relationship. The positive value indicates a positive correlation whereas the negative value indicates a negative correlation. The correlation coefficient is considered significant in this study if the p-value is less than 0.01.

Table 4.2: Interpretation of the Spearman's rank correlation coefficient

ρ **(Negative)**	ρ **(Positive)**	**Interpretation**
$\rho = 0$	$\rho = 0$	Zero
-0.4$<<$ 0.0	0.0$<\leq$ 0.4	Weak
-0.7$<\leq -0.4$	0.4$\leq<$ 0.7	Moderate
-1$<\leq -0.7$	0.7$\leq<$ 1	Strong
ρ = -1	$\rho = 1$	Perfect

4.2.5.3 Bug Severity Analysis

To understand the severity of bugs that are introduced while developers are involved in missing link smell, smelly FIC commits are analysed. For every smelly FIC commit, the corresponding FC commits are identified from the mapping stored in FIC detection step. Only those FC commits are considered that contain

bug ID in their commit message as without this it can not be linked to bug repository. The corresponding bug report is retrieved from the bug repository using that bug ID. Thus, smelly FIC commits are linked to the bug repository and the severity of bugs introduced in these commits can be known.

4.3 Experiment and Result Analysis

According to the procedures described in the above sections, the experimentation is performed. The following subsections provide the description of the dataset and discuss the obtained results.

4.3.1 Dataset

This study aims at investigating the relationship between missing link smell and developers' contribution. To perform the analysis, the study needs some specific software artifacts such as collaboration information, communication information and bug severity information. Thus, the choice of the subject systems for this study is guided by the following factors:

1. Publicly available source code hosted in version control system
2. Publicly available archive of Developer mailing list
3. Bug repository maintaining the information of bug severity

Therefore, seven open-source projects from *Apache* ecosystem are selected for analysis considering the above criteria. The name of the selected projects is provided in Table 4.3 with their source code repository, mailing list and analysis period. These projects are hosted in the online version control system *GitHub*. The development mailing list archive is available on *Gmane* [52]. All the selected projects use *Jira* [53] as the issue tracker. Projects of different ages and sizes are

Table 4.3: List of analysed projects

#	**Project**	**Source Code**	**Mailing List**	**Analysis Period**
1	ActiveMQ	github.com/apache/activemq	gmane.comp.java.activemq.devel	Apr-2006 - Dec-2020
2	Cassandra	github.com/apache/cassandra	gmane.comp.db.cassandra.devel	Oct-2009 - Sep-2020
3	Cayenne	github.com/apache/cayenne	gmane.comp.java.cayenne.devel	Nov-2007 - Aug-2020
4	CXF	github.com/apache/cxf	gmane.comp.apache.cxf.devel	Nov-2010 - Sep-2020
5	Jackrabbit	github.com/apache/jackrabbit	gmane.comp.apache.jackrabbit.devel	Dec-2005 - Sep-2020
6	Mahout	github.com/apache/mahout	gmane.comp.apache.mahout.devel	Oct-2008 - Aug-2020
7	Pig	github.com/apache/pig	gmane.comp.java.hadoop.pig.devel	Oct-2010 - Aug-2020

chosen for analysis. The age of the selected projects ranges from 10 to 15 years and the number of commits ranges from 2,451 to 17,098.

4.3.2 Results and Discussions

This section presents the results obtained from the experimentation described above. The analysis and discussion of the results are elaborated in the following subsections.

4.3.2.1 Number of Developers Involved in Missing Link Smell

Table 4.4 demonstrates the percentage of smelly developers for each project. For example, Apache Cassandra project has 1380 total developers and 205 smelly developers, which is 14.9% of total developers. It is observed that on average 10.5% of total developers of a software community are involved in missing link smells. *Apache Cayenne* community has the highest percentage of smelly developers (21.1%). This is also the smallest community among 7 communities. Tamburri et. al. found that the number of community smell grows quadratically with the number of community members until the threshold of 200 community members [3]. The occurrences of community smell tend to stabilize after this threshold. As the number of total developers in *Apache Cayenne* community is less than 200, the number of missing link smell has not been stabilized yet. So,

Table 4.4: Percentage of smelly developers

#	**Project**	**Total Developers**	**Smelly Developers**	**Smelly Developers(%)**	**Average**
1	Cassandra	1380	205	14.9%	**8.7%**
2	CXF	972	94	9.7%	
3	Jena	244	34	13.9%	
4	Mahout	615	28	4.6%	
5	Pig	668	22	6.0%	
6	Jackrabbit	927	28	3.0%	
7	Cayenne	175	37	21.1%	
	Average	**668**	**64**	**10.5%**	

this project has relatively more missing link smell and consequently more smelly developers. Excluding *Apache Cayenne* project, the rest six projects have 8.7% smelly developers on average.

These results suggest that only a small portion of developers in an open-source software community are involved with missing link smells. They do not communicate appropriately with their co-committing or collaborative developers. Thus, they contribute to the total number of community smells in a software community.

4.3.2.2 Relationship of Missing Link Smell with Developer's Contribution

First, the correlation analysis is performed individually for each development community. The Kendall's tau-b coefficients and p-values are provided in Table 4.5. For example, the correlation coefficient for Apache Cassandra project is 0.508 and it represents a moderate positive correlation. The value of correlation coefficient is significant with a p-value less than 0.01. All seven projects of this study show a moderate positive correlation between number of commits and number of smells which is statistically significant with $p<0.01$.

Another correlation analysis is performed after combining the data from all the projects. The value of the correlation coefficient is slightly increased to 0.612 but still falls under the range of moderate positive correlation. This result is also

Table 4.5: Result of correlation analysis between number of involvements in missing link and number of commits

#	**Project**	**Tau-b**	**p-value**
1	Cassandra	0.508	<0.01
2	Cayenne	0.543	<0.01
3	CXF	0.528	<0.01
4	Jackrabbit	0.589	<0.01
5	Jena	0.452	<0.01
6	Mahout	0.409	<0.01
7	Pig	0.513	<0.01
	Overall	**0.612**	**<0.01**

statistically significant with a p-value less than 0.01.

These results suggest that the number of involvements in missing link smell increases with the number contributions. A developer who contributes more in a project tends to have more missing link smells. This can happen because developers with more contribution have to communicate more with other developers. The overload of communication is a problem in satisfying the required communication needs for these developers. However further analysis is required to find out the causes of involving in more smells.

4.3.2.3 Relationship of Smelly Commits and FICs

To get an idea regarding the proportion of developers involved in missing link smell, the ratio of smelly committers to total committers is calculated. Table 4.6 shows the ratio of smelly committers per six-month analysis window for each evaluated project. The first column shows the name of the project, the second column shows the number of windows analysed in each project. The average number of committers and smelly committers are presented in the third and fourth columns. Finally, the ratio of smelly committers to total committers is shown in the last column. The result suggests that on average 53% committers are involved in missing link smell per window.

Table 4.6: Percentage of smelly committers per window

#	**Project**	**#Analysed Windows**	**Avg. #committers**	**Avg. #Smelly Committers**	**Ratio**
1	ActiveMQ	30	13.27	7.17	0.54
2	Cassandra	26	6.85	4.00	0.58
3	Cayenne	20	23.05	12.55	0.54
4	CXF	30	9.83	4.77	0.48
5	Jackrabbit	16	12.75	4.63	0.36
6	Mahout	24	8.21	4.25	0.52
7	Pig	20	6.10	4.70	0.77
	Overall	*166*	*11.17*	*5.92*	*0.53*

In this study, the relationship between missing link smell and bug introduction is examined. To understand the relation, Spearman's rank correlation is performed between the number of smelly commits and the number of FIC commits for all projects individually. The number of smelly commits and number of FIC commits are calculated for each window. The result of correlation analysis is shown in Table 4.7 with Spearman's correlation coefficient (ρ) and corresponding p-value.

Table 4.7: Result of correlation analysis between the number of smelly commits and FIC commits

#	**Project**	**rho (ρ)**	**p-value**
1	ActiveMQ	0.858	< 0.01
2	Cassandra	0.648	< 0.01
3	Cayenne	0.797	< 0.01
4	CXF	0.944	< 0.01
5	Jackrabbit	0.768	< 0.01
6	Mahout	0.769	< 0.01
7	Pig	0.941	< 0.01

The correlation coefficient is interpreted according to Table 4.2 and considered to be significant if the p-value is less than 0.01. The result suggests that there is a significant positive correlation between number of smelly commits and FIC commits. Among seven evaluated projects, *CXF*, *Pig*, *ActiveMQ*, *Cayenne*, *Jackrabbit* and *Mahout* show a strong positive correlation. A moderate positive correlation is found in *Cassandra*.

These results suggest that missing link smell and bugs are correlated in terms of number of smelly commits and number of FIC commits. It indicates that commits submitted by smelly developers, that is, smelly commits, are very likely to introduce bugs in the system. This information can help the reviewing process in open-source projects. Smelly commits should be reviewed thoroughly to avoid possible bug introduction.

4.3.2.4 Bug Severity Analysis Result

Smelly FIC commits are analysed to understand the severity of bugs introduced by developers who are involved in missing link smell. The following five bug severity categories are found in *Jira* for these projects.

1. **Blocker** - These bugs block development and/or testing work. The production can not run.
2. **Critical** - These bugs cause crashes, loss of data, or severe memory leaks.
3. **Major** - These bugs result in major loss of function.
4. **Minor** - These bugs cause minor loss of function or other problems where an easy workaround is present.
5. **Trivial** - These bugs are about cosmetic problems, for example, misspelled words or misaligned text.

All the evaluated projects except *Cassandra* use the above categorization for bug severity.

Table 4.8 reports the severity of bugs introduced in smelly FIC commits. For example, smelly FIC commits introduce 4.4% *Blocker* bugs, 7.7% *Critical* bugs, 74.4% *Major* bugs, 11.4% *Minor* bugs and 2.2% *Trivial* bugs in *CXF* project. Figure 4.1 shows that most of the bugs produced by smelly commits are major bugs.

Table 4.8: Bug severity of smelly FIC commits

#	**Project**	**Blocker** (%)	**Critical** (%)	**Major** (%)	**Minor** (%)	**Trivial** (%)
1	ActiveMQ	4.4	7.7	74.4	11.4	2.2
2	Cayenne	0.0	0.6	88.5	10.8	0.0
3	CXF	0.3	4.3	82.1	13.0	0.3
4	Jackrabbit	3.2	5.3	68.4	22.1	1.1
5	Mahout	0.0	1.5	67.0	31.0	0.4
6	Pig	0.7	2.1	90.7	5.0	1.4
	Average	*1.4*	*3.6*	*78.5*	*15.6*	*0.9*

On average, 78.5% smelly FIC introduce *Major* bugs, 15.6% introduce *Minor* bugs, 3.6% introduce *Critical*, 1.4% introduce *Blocker* bugs and 0.9% introduce *Trivial* bugs in the system.

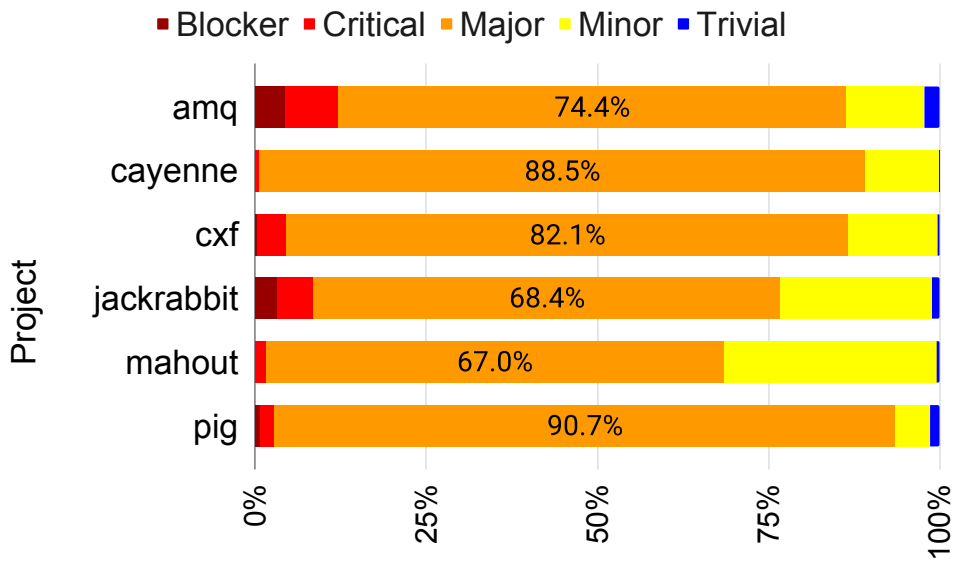

Figure 4.1: The severity of bugs in smelly FIC commits

These results suggest that developers introduce mostly *Major* level bugs in their FIC commits while involved in missing link smell. *Major* bugs are found to have longer fixing time in the literature [54]. Hence, extra maintenance effort and cost may be needed to fix these bugs introduced by the developers who are involved in missing link smell.

4.4 Threats to Validity

This section presents several potential threats that may affect the validity of this study.

Threats to External Validity: Threats to external validity deal with the generalization of the results of the study. Seven open-source projects from *Apache* are analysed in this study. The choice of these projects is guided by several factors such as the availability of source code repository, mailing list archive and bug

repository. However, projects of different sizes and ages are selected for analysis to mitigate this threat. The age of the evaluated projects varies from 10 to 15 years and the size of projects ranges between 2,451 to 17,098 in terms of number of commits.

Threats to Internal Validity: Threats to internal validity deal with the factors that may threaten the validity of the result but are not accounted for. An open-source tool, *Codeface4Smells* [55], is used to detect missing link smell in this study. The identified smells are directly included in the analysis of this study without further verification. However, this tool is commonly used to detect community smell in related studies [4, 5]. Moreover, this tool uses mailing list as the source of communication data to generate communication network. The result can be different if other communication channels, for example, Skype, Slack, etc. are considered. However, according to contribution guidelines of evaluated projects, mailing list is the primary communication channel in these communities. Mailing list is used as the communication source in other related studies [3, 56].

4.5 Summary

This study investigates the relationship between developers' contribution and their involvement in missing link smell. At first, missing link smells are detected for all the projects. Next, the smelly developers are identified by extracting missing link instances. The percentage of smelly developers are calculated for every project. The contribution of a developer to a project is measured by the number of commits. Finally, correlation analysis is done between the contribution and their involvement in smell. This study also explores the relationship between missing link smells and FIC. Furthermore, it examines the severity of bugs that are introduced in the system by the developers who are involved in missing link smell.

For this purpose, seven diverse and open-source projects from *Apache* are analysed. The results suggest that there is a moderate positive correlation between the number of commits of a developer and the number of involvement in missing link smells. The developers who contribute more tend to involve in more missing link smell. Furthermore, there is a significant positive correlation between the number of smelly commits and FIC commits. The findings reveal that developers mostly introduce major bugs in the system while involved in missing link smell. Based on this result, the next chapter investigates the impact of community smells on maintainability metrics in details.

Chapter 5

Relationship Between Community Smell and Software Maintainability

Community smells can be defined as organizational and social anti-patterns in a development community. These smells can influence developers' maintenance decisions and activities such as bug fixing, source code refactoring, etc. For this reason, understanding of how and to which extent community smells impact software maintainability is important and yet to be analyzed. To identify the impact, an empirical study is conducted to investigate whether the maintainability differs between classes affected by community smells and those which are not. To assess maintainability, change-proneness, fault-proneness, code smells and metrics related to five quality attributes of maintainability such as modularity, modifiability, etc. are considered. Then, the distributions of these metrics are compared between smelly and non-smelly classes using statistical tools. The result shows that smelly classes are more change and fault-prone than non-smelly classes as well as more likely to contain code smells. In terms of quality attributes, the maintainability appears to be lower in smelly classes compared to non-smelly classes.

5.1 Introduction

Community smells are communication and collaboration issues which are common among developers in a software development community. These, the organizational and social anti-patterns in the community, may lead to the emergence of unforeseen project costs, known as social debt [1]. Software maintainability is defined as the modification capability of a software [7]. It may decrease, as the developers' interaction with source code is not only dependent on technical factors but also on inter-personal issues [4]. To keep the source code of a software more maintainable, understanding how and to which extent community smells impact software maintainability is important.

To understand the impact of community smells on software maintainability, the relationships between community smells and maintainability metrics need to be established. Community smells occur based on how developers communicate and collaborate among themselves while developing software. It is to be investigated whether the maintainability of the source code differs due to developers' involvement in community smells during the development time. How maintainability metrics differ in the presence and absence of community smells can yield an understanding of how software maintainability is affected by community smells.

Existing researches have been studying community smells from different perspectives such as definition, detection and impact analysis. It has started by defining community smells from an industrial case study [2]. Later, researchers have studied how to detect those in open-source projects analysing project repositories and development mailing lists [3, 9, 11]. These definitions and detection methods provide the basic understandings of community smells, necessary for further impact analysis of community smells. In another study, developers' perceptions about community smells are investigated where it is considered as harmful for development [3]. Based on that, by predicting code smell intensity [4] and bug

[10] from community smells, a few studies established the relationship between community smells and these technical factors. Although the impact of community smells on some individual technical factors have been studied, the impact of community smells on software maintainability as a whole is yet to be analyzed.

In this study, an empirical research is conducted to identify the impact of community smells on software maintainability. This study considers four community smells namely *Organizational Silo*, *Missing Link*, *Radio Silence*, and *Black Cloud*, which are mostly used in the literature to represent community smells [4, 6, 57]. The details about these smells are discussed in Chapter 2. To detect these smells, open-source project repositories are cloned from *GitHub* and developer mailing lists are collected from the archive named *Gmane* [52]. Communication and collaboration graphs are built by parsing the change history of the repository and the mailing list respectively. From these graphs, community smells are identified along with involved developers (called smelly developers) by looking for communication and collaboration patterns [3].

On the other hand, to measure software maintainability, this study uses change-proneness, fault-proneness, code smell and *ISO/IEC 25010* defined five quality attributes namely *modularity*, *reusability*, *analyzability*, *modifiability*, and *testability* [7, 15, 17, 18]. The change-proneness is calculated as the number of changes performed in the class. For this purpose, the change history is parsed from the project repository. The fault-proneness of a class is measured as the number of fixing-changes. From the project repository and bug repository, fixing-changes are extracted matching the issue ID written in the commit message with the corresponding issue report. To detect code smells in a class, a prominent and efficient rule-based approach is used named *DECOR* [58]. To identify *ISO/IEC 25010* defined quality attributes, 14 class level metrics are also calculated by static code analysis on the project repositories [19].

317 releases of 14 open-source projects are analysed to investigate the impact of community smells on software maintainability. The result of the study shows that software maintainability is affected by community smells in terms of considered metrics. It is found that the change-proneness and fault-proneness are high in smelly classes compared to non-smelly classes. The observed differences in the mean values of change-proneness and fault-proneness are found statistically significant from Mann–Whitney U test [59]. Furthermore, classes affected by community smells are more likely to have code smells than classes not affected by community smells. As suggested by the odds ratio, smelly classes are 1.7 times more likely to contain code smells than non-smelly classes. The result of the object-oriented metrics shows that smelly classes are less maintainable compared to non-smelly classes. The finding indicates that community smells have an adverse impact on software maintainability. The details of the study are presented in the following sections.

5.2 Methodology

In this study the impact of community smells on software maintainability is investigated. The overview of the proposed methodology is shown in Figure. 5.1. First, community smells are identified from the project repository and mailing list. After detecting community smells, the developers involved in any of those community smells are defined as smelly developers. Next, classes affected by the community smells, that is, modified by smelly developers, are called smelly classes, otherwise non-smelly classes [4, 10]. As shown in Figure. 5.1, the next step is measuring the maintainability of the software system. To measure maintainability, change-proneness, fault-proneness, code smell, and *ISO/IEC 25010* defined five quality attributes such as modularity, reusability, etc. are used as these are commonly utilized in previous studies [7, 15, 16, 17, 18, 19]. Finally, statistical analysis is per-

formed to identify whether maintainability differs between smelly and non-smelly classes. The details of the research steps are elaborated below.

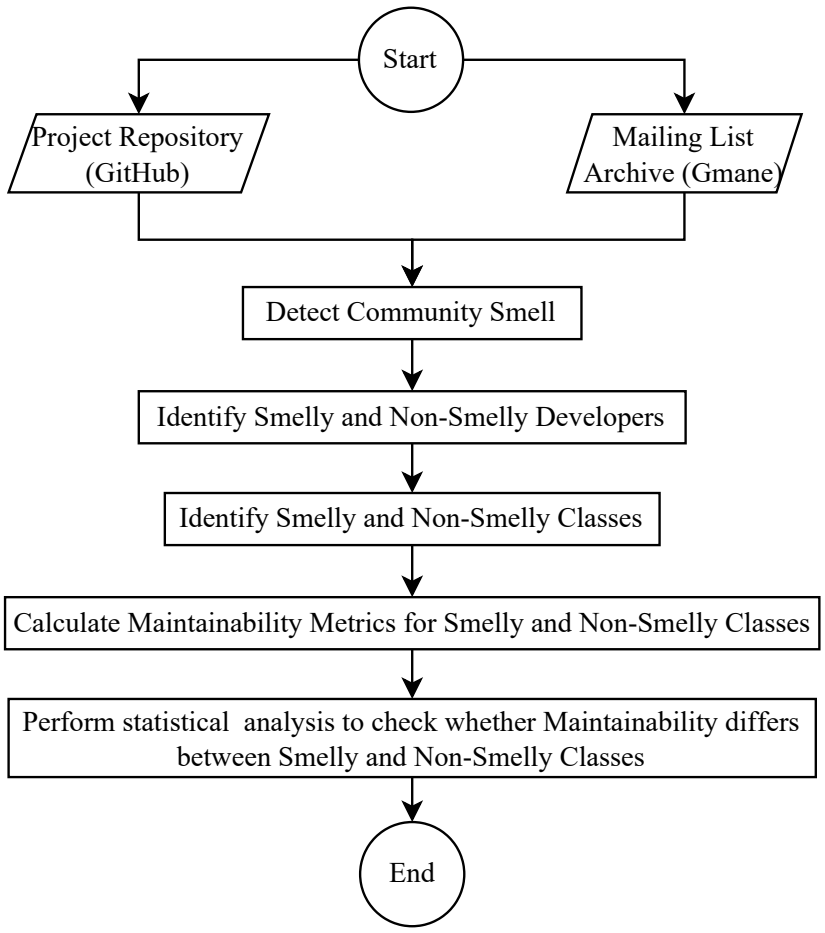

Figure 5.1: Overview of the methodology

5.2.1 Detecting Community Smells

Detecting community smells is the first step of this study. Community smells are identified from the source code repository and development mailing list. First, the collaboration network of the development community is generated from the change history of the project by connecting developers who work on the same source code. On the other hand, the communication network is generated mining the mailing list by connecting developers who reply in the same email. From collaboration and communication network, community smells are detected according

to the identification patterns given by Tamburri et al. [3]. For instance, Lone Wolf is detected by identifying an edge that is present in the collaboration network but missing from the communication network. To detect community smells, the state-of-the-art tool *Codeface4Smells* [9] is used. This tool is publicly available and widely used by most of the studies related to community smells [3, 4, 5, 6]. It can detect four community smells namely *Organizational Silo*, *Lone Wolf*, *Bottleneck*, and *Black Cloud*. The validity of the tool has been empirically evaluated through a qualitative investigation with developers [60]. According to the findings, the community smells found by this tool are all true positives and no additional smell instances were mentioned by developers which reduces the risk of having false negatives. This validation makes the tool reliable and hence used in this study.

The software development community changes from time to time, so, a temporal window must be set to detect community smells. In this study, the release is chosen because substantial changes are needed to observe the impact of community smells on maintainability metrics that can be found between two releases. After detecting community smells, developers in a release are divided into two categories - smelly developers and non-smelly developers. A developer involved in any kind of community smell is considered a smelly developer for that release otherwise non-smelly developer.

The involvement of classes in any kind of community smells is identified to divide those into two categories, which are smelly and non-smelly class. A class is said to be affected by community smells, if the class has been modified by a smelly developer [4, 10]. For example, if a class C_i is modified by a smelly developer in between two consecutive releases r_{j-1} and r_j, class C_i is considered smelly otherwise non-smelly. This approach has been followed by existing studies as well [4, 10].

5.2.2 Detecting Change-proneness

Change-proneness is one of the important metrics to understand the maintainability of a class. The change-proneness of a class can be computed as the ratio of number of changes in a class to the number of total changes between two releases [15]. The change in a class can be addition or deletion. So, the number of changes of a class C_i is calculated using equation (5.1).

$$\#changes(C_i) = added(C_i) + deleted(C_i) \tag{5.1}$$

In equation (5.1), $added(C_i)$ and $deleted(C_i)$ are the number of added and deleted lines respectively. The number of added lines ($added(C_i)$) and deleted lines ($deleted(C_i)$) of each class between r_{j-1} and r_j are computed using commit history. Finally, the change-proneness of a class C_i in a release r_j is computed using equation (5.2),

$$change\_proneness(C_i, r_j) = \frac{\#changes(C_i)_{r_{j-1} \to r_j}}{\#changes(r_{j-1} \to r_j)} \tag{5.2}$$

where $\#changes(C_i)_{r_{j-1} \to r_j}$ is the number of changes in C_i during the time t between release r_{j-1} and r_j, and $\#changes(r_{j-1} \to r_j)$ is the total number of changes in the whole project during t.

The length of t can bias the change-proneness of a class. For example, the number of changes in a class is usually higher for longer interval between releases [15]. Since the time period t between release r_{j-1} and r_j is not equal for all releases, the change-proneness is normalized in equation (5.2) dividing the number of changes by the total number of changes in a release.

5.2.3 Detecting Fault-proneness

Fault-proneness is another important metric which is needed to understand the maintainability of a class. The fault-proneness of a class is calculated as the ratio of

number of bug-fixing changes in a class and the total number of bug-fixing changes in between two releases. Where, the bug-fixing changes (addition or deletion for fixing bug) are identified through bug-fixing commits. To extract bug-fixing commits that contain *Issue ID* or *Bug ID*, commit messages are searched using regular expression [15]. For example, let consider the following commit message from *ActiveMQ*[1] project:

> "[AMQ-8309] Fix spring import range, change optional to be more flexible . . . "

The above commit can be identified with the regular expression "*AMQ-\\d+*". As there are multiple types of issues such as bug, enhancement etc., issues related to bugs need to be distinguished. For each *Issue ID* found in commit messages, the corresponding issue report is extracted from the issue tracker such as *Jira* [53] and *Bugzilla* [61]. Then, bug type issues are separated by checking their types. However, there may be duplicated or false-positive bugs which can bias the result. To exclude those bugs, only bugs that have the status *Closed* or *Resolved* and the resolution *Fixed* are considered [15].

Next, the number of bug-fixing changes of a class C_i is calculated using equation (5.3).

$$\#bug\_fixing\_changes(C_i) = fix\_added(C_i) + fix\_deleted(C_i) \tag{5.3}$$

Here, $fix\_added(C_i)$ and $fix\_deleted(C_i)$ are the number of added and deleted lines respectively for fixing changes. Finally, the fault-proneness of a class C_i in a release r_j is computed using equation (5.4).

$$fault\_proneness(C_i, r_j) = \frac{\#bug\_fixing\_changes(C_i)_{r_{j-1} \to r_j}}{\#bug\_fixing\_changes(r_{j-1} \to r_j)} \tag{5.4}$$

[1]https://github.com/apache/activemq

In equation (5.4), $\#bug\_fixing\_changes(C_i)_{r_{j-1} \to r_j}$ is the number of bug-fixing changes in C_i during the time t between release r_{j-1} and r_j, and $\#bug\_fixing\_changes(r_{j-1} \to r_j)$ is the total number of bug fixing-changes in the whole project during t.

Table 5.1: List of considered code smells in this study

Code Smell	**Reference**
AntiSingleton	[62]
BaseClassKnowsDerivedClass	[58]
BaseClassShouldBeAbstract	[58]
Blob	[62]
ClassDataShouldBePrivate	[62]
ComplexClass	[62]
FunctionalDecomposition	[62]
LargeClass	[24]
LazyClass	[24]
LongMethod	[24]
LongParameterList	[24]
ManyFieldAttributesButNotComplex	[58]
MessageChains	[24]
RefusedParentBequest	[24]
SpaghettiCode	[62]
SpeculativeGenerality	[24]
SwissArmyKnife	[62]
TraditionBreaker	[58]

5.2.4 Detecting Code Smells

Code smell denotes the poor implementation choices applied by developers which is needed to understand the maintainability of a class. To detect code smells of a class, the rule-based approach is followed named as *DECOR* [58]. In this approach, code smells are identified based on a set of rules, called rule cards[2]. Among the available code smell detection tools, *DECOR* [58] is selected as it has been utilized in earlier studies of code smells efficiently with a good performance [15, 17, 63]. Table 5.1 provides the list of 18 code smells considered in this study.

[2]http://www.ptidej.net/research/designsmells/grammar/Blob.txt

After detecting these smells, classes are categorized into two groups which are *CodeSmell* and *NoCodeSmell*. A class C_i is labelled as *CodeSmell* in release r_j, if it contains any kind of code smell in that release, otherwise labelled as *NoCodesmell*.

Table 5.2: List of maintainability metrics

Category	**Metrics**
Complexity	Class Lines of Code (LOC) [64]
	Number of Instance Methods (NIM) [64]
	Number of Instance Variables (NIV)
	Weighted Methods per Class (WMC) [65]
Coupling	Coupling Between Objects (CBO) [65]
	Response For a Class (RFC) [65]
Cohesion	Lack of Cohesion in Methods (LCOM) [65]
Abstraction	Number of Immediate Base Classes (IFANIN) [64]
	Number of Immediate Subclasses (NOC) [65]
	Depth of Inheritance Tree (DIT) [65]
Encapsulation	Ratio of Public Methods (RPM) [64]
	Ratio of Static Methods (RSM) [64]
Documentation	Comment of Lines per Class (CLOC) [64]
	Ratio Comments to Codes per Class (RCC) [64]

5.2.5 Object Oriented Metrics

To examine *ISO/IEC 25010* defined five quality attributes of maintainability, namely *modularity*, *reusability*, *analyzability*, *modifiability*, and *testability*, six groups of object-oriented metrics are employed [41]. These are *complexity*, *coupling*, *cohesion*, *abstraction*, *encapsulation*, and *documentation*. These groups are related with the above five quality attributes in the following way [19]:

- high complexity of a class indicates low analyzability and modifiability

- high coupling negatively affects analyzability and reusability

- low cohesion reduces modularity and modifiability

- low abstraction decreases reusability

- low encapsulation contributes to less modularity

- proper documentation implies high analyzability, modifiability and reusability

14 class level metrics are selected ensuring at least one metric from each group [19]. The list of object-oriented metrics of maintainability and their categories are shown in Table 5.2. A static code analysis tool, *Understand* (version:5.1, Build:1029), is used to compute these metrics. The identified metrics will be used in the next step for statistical analysis.

5.2.6 Statistical Analysis

According to the research steps described above, all required artifacts such as community smell, smelly and non-smelly classes, change-proneness, fault-proneness and other maintainability metrics are collected. Then, statistical analysis is performed to analyze the relationship between community smells and maintainability metrics.

To investigate the impact of community smell on metric m, the following null hypothesis is formulated.

$H_0 1$*: There is no difference in the distribution of metric values between community smell affected class and unaffected class.*

Mann-Whitney U test is used to examine the difference in the distribution of values of metric m between smelly and non-smelly classes [59]. This test is a non-parametric statistical test and does not require a normal distribution. It is used to assess the null hypothesis stating that a randomly selected observation from one group is equally likely to be less than or greater than a randomly selected observation from another group. In this study, the test is applied with 1% significance level, that is, p-value < 0.01 as a widely used level of significance in software engineering empirical researches [31] As the study investigates 16 metrics, Bonferroni correction is applied which adjusts p-values by multiplying with the number of tests (16 tests) to eliminate family-wise error rate [66]. This hypothesis

testing reveals whether metric m differs in the presence and absence of community smells.

As code smells are computed as a categorical variable namely *CodeSmell* and *NoCodeSmell*, a separate hypothesis is formulated to investigate whether classes affected by community smells exhibit a different likelihood of having code smell as compared to non-smelly classes.

$H_0$2: The proportion of classes having at least one code smell does not differ between smelly and non-smelly classes.

To attempt rejecting $H_0$2, Fisher's Exact test [67] is used which checks whether a proportion varies between two groups. To understand the likelihood of the event to occur, the *Odds Ratio (OR)* [68] is also computed. The *Odds Ratio (OR)* is defined as the ratio of odds p of an event occurring in one group, to the odds q of the same event occurring in the second group. For example, the odds that smelly classes having code smell, to the odds non-smelly classes having code smell. An *OR* equal to one indicates that the event is equally likely in both groups. An *OR* greater than one indicates that the event is more likely in the first group, that is, smelly group. An *OR* less than one indicates that the event is more likely in the second group, that is., non-smelly group. The result of the hypothesis reveals whether classes affected by community smells are more likely to contain code smells. The following section presents the experiment and result discussions of the study.

5.3 Experiment and Result Analysis

To examine the impact of community smell on software maintainability, the experiment is carried out on 317 releases of 14 open-source projects according to the methodology described in Section 5.2. The description of the dataset and the results of the experiment on the dataset are described in the following subsections.

Table 5.3: List of analyzed software projects

Project	**Description**	**Source Code**	**#Analysed Releases**	**Analysis Period**	**#Analysed Commits**
ActiveMQ	Java Message Broker	github.com/apache/activemq	24	Feb-2007 - Jan-2021	9086
Ant	Build System	github.com/apache/ant	33	Jul-2002 - May-2020	11797
Cassandra	Database Management System	github.com/apache/cassandra	18	Oct-2011 - Jul-2020	10102
Cayenne	ORM Framework	github.com/apache/cayenne	17	Jul-2007 - Jul-2020	6569
CXF	Web Services Framework	github.com/apache/cxf	16	Oct-2010 - Aug-2020	14988
Drill	Distributed Query Engine	github.com/apache/drill	24	Sep-2013 - Jun-2021	3842
Eclipse-CDT	Integrated Development Environment	github.com/eclipse-cdt/cdt	38	Jun-2009 - Mar-2021	21818
Jackrabbit	Java Content Repository	github.com/apache/jackrabbit	28	Oct-2006 - Jul-2020	7060
Jena	Semantic Web Framework	github.com/apache/jena	30	Oct-2012 - Mar-2021	7911
Mahout	Distributed Linear Algebra Framework	github.com/apache/mahout	12	May-2010 - Sep-2020	4026
OpenNLP	Natural Language Processor	github.com/apache/opennlp	8	Dec-2014 - Jul-2020	1871
Pig	Large Dataset Analyzer	github.com/apache/pig	15	Dec-2010 - Jun-2017	3476
POI	API to access Microsoft Office formats	github.com/apache/poi	19	Jun-2007 - Jan-2021	8820
Tomcat	Java HTTP WebServer	github.com/apache/tomcat	35	Jun-2010 - Sep-2020	19370
Total			*317*		*130736*

5.3.1 Data Description

To find the subject systems for analysis, a list of 94 projects is retrieved from datasets used in previous empirical studies related to community smells [3, 4]. From this list, this study focuses on projects written in Java programming language. This is because Java is a popular object-oriented language and necessary resources are available to detect code smells. After filtering out non-Java systems, the number of projects reduces from 94 to 28. Then, 3 projects are excluded as the issue tracker is not found in Jira or Bugzilla which is necessary for identifying fault-proneness. From remaining 25 projects, 11 more projects terminated with error while analysing collaboration and communication by *Codeface4Smells*. The reasons of the error are manually checked. For example, a project named *Jmeter* has no communication in the provided mailing list in between considered releases. After excluding these projects, the final dataset consists 14 projects.

The details of the selected projects are provided in Table 5.3 with the project name, the number of considered releases, analysis period and the number of analysed commits. For example, 28 releases of *ActiveMQ*, a java message broker system, are considered for analysis which has covered about 14 years of lifetime and 9086 commits.

The descriptive analysis is performed on these projects to get an idea about the dataset. The source code of selected projects is available in *Github* and the development mailing list archives are available in *Gmane* [52]. The issues of the selected projects are maintained in either *Jira* or *Bugzilla*. The selected projects have 317 releases and change history of 130,736 commits in total. In terms of commits, the size of the projects ranges from 3,476 to 21,818 commits. The selected projects belong to different application domains such as DBMS, IDE, Web Framework, Data Analyzer, etc. The resulting dataset consists of 955,237 classes from 317 releases of 14 open-source projects. The dataset contains 149,365 smelly classes and 805,872 non-smelly classes.

Table 5.4: Result of the overall impact of community smells on software maintainability (n.s. means non-significant p-value)

Category	**Metrics**	**Community Smell**		**Adj. p-value**
		NonSmelly	*Smelly*	
Change-proneness		8.47E-05	1.34E-03	< 0.01
Fault-proneness		7.04E-05	1.39E-03	< 0.01
Complexity	LOC	99.65	174.50	< 0.01
	NIM	7.73	11.53	< 0.01
	NIV	1.96	2.93	< 0.01
	WMC	8.60	13.06	< 0.01
Coupling	CBO	10.06	16.37	< 0.01
	RFC	39.33	43.72	< 0.01
Cohesion	LCOM	32.29	38.78	< 0.01
Abstraction	IFANIN	1.32	1.34	n.s.
	NOC	0.64	0.71	< 0.01
	DIT	2.20	2.09	< 0.01
Encapsulation	RPM	0.77	0.75	< 0.01
	RSM	0.12	0.14	< 0.01
Documentation	CLOC	38.75	57.03	< 0.01
	RCC	0.80	0.58	< 0.01

5.3.2 Results and Discussions

The results of the study are presented in this section. First, the overall impact of community smells on software maintainability is discussed where there is no discrimination among the specific kind of community smells. That means a class is considered smelly if it is affected by any of the four considered community smells. Table 5.4 shows the summary of the results with the mean value of each metric for smelly and non-smelly classes.

Then, the individual impact of each community smell on software maintainability is presented. Among four community smells, *Black Cloud* is found only in 9% releases (27 out of 317) whereas *Organisational Silo*, *Lone Wolf*, and *Radio Silence* are found in 73%, 76%, and 89% of the considered releases respectively. This finding is consistent with the previous study [3], where the presence of *Black Cloud* instances was less. The reason for the absence of *Black Cloud* smell in open-source projects is that open-source communities do not lack structured com-

Table 5.5: Result of the impact of Organizational Silo on software maintainability (n.s. means non-significant p-value)

Category	**Metrics**	**Organizational Silo**		**Adj. p-val**
		NonSmelly	*Smelly*	
	Change-proneness	2.19E-04	1.53E-03	< 0.01
	Fault-proneness	2.09E-04	1.65E-03	< 0.01
Complexity	LOC	106.694	206.693	< 0.01
	NIM	8.080	13.307	< 0.01
	NIV	2.056	3.314	< 0.01
	WMC	9.017	14.970	< 0.01
Coupling	CBO	10.677	18.614	< 0.01
	RFC	39.635	47.885	< 0.01
Cohesion	LCOM	32.969	40.239	< 0.01
Abstraction	IFANIN	1.322	1.364	<0.01
	NOC	0.641	0.772	< 0.01
	DIT	2.182	2.151	n.s.
Encapsulation	RPM	0.766	0.747	< 0.01
	RSM	0.121	0.133	< 0.01
Documentation	CLOC	40.402	66.380	< 0.01
	RCC	0.775	0.559	< 0.01

munication so they can avoid information overload [3]. Therefore, *Black Cloud* is excluded from the individual impact analysis of community smells on maintainability. Table 5.5, 5.6 and 5.7 demonstrate the impact of individual community smell on maintainability metrics.

Next, the impact analysis is conducted controlling the size of the class considering all four community smells. This is done to ensure the results achieved are not simply due to the reflection of class size. So, both smelly and non-smelly classes are grouped together with similar sizes. Classes are categorized into three groups namely *large*, *medium* and *small*, based on Line of Codes (LOC), similar as [15]. Classes having a size lower than the first quartile of the distribution of LOC are considered as *small*, classes having a size between the first and the third quartile are considered *medium*, and classes having a size larger than the third quartile are considered *large* classes. Table 5.8, 5.9 and 5.10 show the result of the size based analysis. The results are discussed below elaborately with each maintainability metric.

Table 5.6: Result of the impact of Lone Wolf on software maintainability (n.s. means non-significant p-value)

Category	**Metrics**	**Lone Wolf**		**Adj. p-val**
		NonSmelly	*Smelly*	
	Change-proneness	1.31E-04	1.21E-03	<0.01
	Fault-proneness	1.19E-04	1.25E-03	<0.01
Complexity	LOC	101.225	174.427	<0.01
	NIM	7.821	11.454	<0.01
	NIV	1.989	2.893	<0.01
	WMC	8.699	13.007	<0.01
Coupling	CBO	10.177	16.468	<0.01
	RFC	39.526	43.092	<0.01
Cohesion	LCOM	32.404	38.934	<0.01
Abstraction	IFANIN	1.321	1.341	<0.01
	NOC	0.639	0.697	<0.01
	DIT	2.194	2.096	<0.01
Encapsulation	RPM	0.767	0.752	<0.01
	RSM	0.118	0.146	<0.01
Documentation	CLOC	39.363	55.619	<0.01
	RCC	0.798	0.558	<0.01

5.3.2.1 Change-proneness

Table 5.4 shows that the mean change-proneness of smelly classes (1.34E-03) is about 15 times higher than the mean change-proneness of non-smelly classes (8.47E-05). The Mann-Whitney U test confirms that the observed difference is statistically significant, that is, the adjusted p-value is less than 0.01. Moreover, Table 5.5, 5.6 and 5.7 show that smelly classes are more change-prone than non-smelly classes when affected by *Organisational Silo*, *Lone Wolf*, and *Radio Silence* individually. The result is also found consistent for *small*, *medium* and *large* sized classes as shown in Table 5.8, 5.9 and 5.10 respectively.

The results suggest that classes affected by community smells are more change-prone than unaffected classes. The knowledge gap created by community smells can cause the frequent change in a class. As an example, the class *ComponentHelper* of the *Ant* project is involved in 26 changes on average during the time period when it is not affected by community smells (18 releases). On the other

Table 5.7: Result of the impact of Radio Silence on software maintainability (n.s. means non-significant p-value)

Category	**Metrics**	**Radio Silence**		**Adj. p-val**
		NonSmelly	*Smelly*	
	Change-proneness	1.84E-04	1.36E-03	< 0.01
	Fault-proneness	1.71E-04	1.46E-03	< 0.01
Complexity	LOC	105.311	179.030	< 0.01
	NIM	8.017	11.760	< 0.01
	NIV	2.037	2.979	< 0.01
	WMC	8.929	13.396	< 0.01
Coupling	CBO	10.554	16.569	< 0.01
	RFC	39.723	43.331	< 0.01
Cohesion	LCOM	32.781	39.209	< 0.01
Abstraction	IFANIN	1.323	1.337	n.s.
	NOC	0.641	0.713	< 0.01
	DIT	2.191	2.056	<0.01
Encapsulation	RPM	0.766	0.754	< 0.01
	RSM	0.119	0.149	< 0.01
Documentation	CLOC	40.021	59.441	< 0.01
	RCC	0.779	0.600	< 0.01

hand, the average number of changes increased to 178 during the time period when it is affected by community smells (11 releases). Thus, community smells show a negative impact on software maintainability, as change-prone classes are more difficult to maintain [20].

5.3.2.2 Fault-proneness

From Table 5.4, it is observed that smelly classes have higher fault-proneness than non-smelly classes. The mean fault-proneness of smelly classes (1.39E-03) is about 19 times higher than non-smelly classes (7.04E-05). The observed difference is statistically significant. Table 5.5, 5.6 and 5.7 illustrate that when affected by individual community smell, smelly classes are also more fault-prone compared to non-smelly classes. As presented in Table 5.8, 5.9 and 5.10, the similar results are found for *small*, *medium* and *large* sized classes.

The result implies that smelly classes are more prone to faults than non-smelly classes. Being more fault-prone, smelly classes affect software maintainability

Table 5.8: Result of the impact of community smells on software maintainability in small classes (n.s. means non-significant p-value)

Category	**Metrics**	**Small Classes**		**Adj. p-val**
		NonSmelly	*Smelly*	
	Change-proneness	2.30E-05	4.11E-04	< 0.01
	Fault-proneness	1.62E-05	3.77E-04	< 0.01
Complexity	LOC	11.496	12.026	< 0.01
	NIM	1.857	1.814	< 0.01
	NIV	0.353	0.400	< 0.01
	WMC	2.034	2.017	n.s.
Coupling	CBO	3.197	3.595	< 0.01
	RFC	34.279	33.252	< 0.01
Cohesion	LCOM	8.685	7.680	< 0.01
Abstraction	IFANIN	1.186	1.189	n.s.
	NOC	0.537	0.437	n.s
	DIT	2.428	2.320	<0.01
Encapsulation	RPM	0.744	0.742	< 0.01
	RSM	0.086	0.107	< 0.01
Documentation	CLOC	14.464	14.132	n.s.
	RCC	1.649	1.628	< 0.01

negatively. This is because a fault-prone class needs frequent bug-fixing which makes it difficult to maintain [21].

5.3.2.3 Code Smells

Table 5.11 reports the number of classes that are (1) smelly and have at least one code smell, (2) smelly and have no code smell, (3) non-smelly and have at least one code smell, and (4) non-smelly and have no code smell. The table also reports the *OR* value and the result of Fisher's Exact test. The value of *OR* implies that smelly classes are 1.7 times more likely to contain code smells than non-smelly classes when all community smells are considered together. The *OR* is found 1.8, 1.69 and 1.64 respectively when *Organisational Silo*, *Lone Wolf*, and *Black Cloud* are considered individually. As shown in Table 5.11, the result of Fisher's Exact test shows that the proportion of classes with code smells significantly different (p-value < 0.01) between smelly and non-smelly classes.

Table 5.12 shows that the OR values are 1.28 and 1.32 for *medium* and *large*

Table 5.9: Result of the impact of community smells on software maintainability in medium classes (n.s. means non-significant p-value)

Category	**Metrics**	**Medium Classes**		**Adj. p-val**
		NonSmelly	*Smelly*	
	Change-proneness	5.12E-05	7.97E-04	<0.01
	Fault-proneness	4.24E-05	7.70E-04	<0.01
Complexity	LOC	53.706	58.187	<0.01
	NIM	5.845	5.807	<0.01
	NIV	1.577	1.632	n.s.
	WMC	6.459	6.526	n.s.
Coupling	CBO	8.716	10.513	<0.01
	RFC	37.222	36.717	<0.01
Cohesion	LCOM	33.106	32.452	<0.01
Abstraction	IFANIN	1.314	1.281	<0.01
	NOC	0.594	0.577	<0.01
	DIT	2.170	2.110	<0.01
Encapsulation	RPM	0.805	0.791	<0.01
	RSM	0.119	0.139	<0.01
Documentation	CLOC	26.625	25.127	<0.01
	RCC	0.561	0.483	<0.01

classes respectively. This indicates that *medium* and *large* classes are more likely to have code smells when those are affected by community smells. As suggested by odds ratio (OR=0.88), *small* classes are not likely to contain more code smells due to community smells.

The results suggest that developers should be careful of classes affected by community smells as those are more likely to have code smells. Therefore, smelly classes may need extra maintenance effort as they need refactoring [69]. Thus, community smells impact the software maintainability negatively.

5.3.2.4 Object Oriented Metrics

To better understand the impact of community smells on object oriented metrics of maintainability, the results are discussed below from the perspective of six categories: *complexity*, *coupling*, *cohesion*, *abstraction*, *encapsulation*, and *documentation*.

Table 5.10: Result of the impact of community smells on software maintainability in large classes (n.s. means non-significant p-value)

Category	**Metrics**	**Large Classes**		**Adj. p-val**
		NonSmelly	*Smelly*	
	Change-proneness	2.34E-04	2.34E-03	< 0.01
	Fault-proneness	1.98E-04	2.51E-03	< 0.01
Complexity	LOC	308.661	377.353	< 0.01
	NIM	18.981	22.162	< 0.01
	NIV	4.750	5.467	< 0.01
	WMC	21.244	25.180	< 0.01
Coupling	CBO	21.265	28.302	< 0.01
	RFC	50.132	56.177	< 0.01
Cohesion	LCOM	58.483	58.165	< 0.01
Abstraction	IFANIN	1.501	1.459	<0.01
	NOC	0.849	0.962	< 0.01
	DIT	1.987	1.971	n.s.
Encapsulation	RPM	0.710	0.710	n.s.
	RSM	0.151	0.166	< 0.01
Documentation	CLOC	95.149	112.038	< 0.01
	RCC	0.330	0.306	< 0.01

Complexity: The complexity of a class is measured by four metrics in this study which are LOC, NIM, NIV and WMC. The mean values of these metrics are reported in Table 5.4 for both smelly and non-smelly classes. All the metrics for complexity show that the mean values are higher in smelly classes than non-smelly classes. For instance, the mean value of WMC metric is 13.06 in smelly classes and 8.60 in non-smelly classes respectively. The mean values of all complexity metrics imply that smelly classes are more complex than non-smelly classes. The results of Mann-Whitney U test tell those differences are statistically significant. From Table 5.4, the values of complexity metrics are 56% higher in smelly classes with respect to non-smelly classes.

Table 5.5, 5.6 and 5.7 show that the results are consistent for each three individual community smells. As presented in Table 5.10, only for *large* classes the mean values of all complexity metrics are higher in smelly classes. These results suggest that community smells have an impact on the complexity of classes when they become large. For *small* (Table 5.8) and *medium* (Table 5.9) classes, it is

Table 5.11: Result of community smell and code smell analysis

		CodeSmell	NoCodeSmell	OR	p-val
Overall	Smelly	79110	70255	1.70	<0.01
	NonSmelly	320589	485283		
Org. Silo	Smelly	24809	19665	1.80	<0.01
	NonSmelly	374890	535873		
Lone Wolf	Smelly	70055	62069	1.69	<0.01
	NonSmelly	329644	493469		
Radio Silence	Smelly	41572	36680	1.64	<0.01
	NonSmelly	358127	518858		

Table 5.12: Result of community smell and code smell in different sized classes

Size		CodeSmell	NoCodeSmell	OR	p-val
Small	Smelly	3688	17868	0.88	<0.01
	NonSmelly	40607	174064		
Medium	Smelly	27351	42908	1.28	<0.01
	NonSmelly	136677	273777		
Large	Smelly	48071	9479	1.32	<0.01
	NonSmelly	143305	37442		

easier to understand the source code and maintain the complexity regardless of the effect of community smells. As the class size grows large, it becomes difficult to reduce the complexity without adequate knowledge about the class which is hindered by community smells. The high complexity of such classes indicates less modifiability and analyzability and thus affect software maintainability negatively.

Coupling: To measure the coupling of classes, two metrics are used such as CBO and RFC. Table 5.4 shows that the mean values of CBO are 16.37 and 10.06 in smelly and non-smelly classes respectively. On the other hand, the mean values of RFC metric are 43.72 and 39.33 respectively for smelly and non-smelly classes. Both metrics indicate high coupling in smelly classes with respect to non-smelly classes. The observed differences are statistically significant for both metrics. On average, the values of coupling metrics are 37% higher in smelly classes than non-smelly classes.

Table 5.5, 5.6 and 5.7 show the similar results for all three individual community smells. When classes are grouped according to their size, the mean values of

CBO are found significantly higher in smelly classes for *small* (Table 5.8), *medium* (Table 5.9) and *large* (Table 5.10) classes. But the mean value of RFC is found significantly higher in smelly classes only for large classes. It indicates that RFC is not affected by community smells in small and medium sized classes. As being more coupled than non-smelly classes, smelly classes affect maintainability negatively by decreasing modularity and analyzability.

Cohesion: Cohesion is measured by LCOM metric in both smelly and non-smelly classes. The mean value of LCOM in smelly classes is 38.78, whereas the mean value of LCOM is 32.29 in non-smelly classes as reported in Table 5.4. It means that smelly classes are less cohesive with respect to non-smelly classes. The result of Mann-Whitney U test confirms that the observed difference is statistically significant (p-value < 0.01).

As shown in Table 5.5, 5.6 and 5.7, the results are similar for *Organisational Silo*, *Lone Wolf*, and *Radio Silence* community smell. The smelly classes are found more cohesive than non-smelly classes when analysed by grouping similar sized classes together as demonstrated in Table 5.8, 5.9 and 5.10. It indicates that the previous effect on LCOM is due to the size of the class. The reason behind this can be smelly developers prefer using methods from classes where they work rather than using other classes developed by other developers.

Abstraction: The abstraction property of classes is measured by three metrics such as IFANIN, DIT and NOC. As reported in Table 5.4, the mean values of DIT in smelly and non-smelly classes are 2.09 and 2.20 respectively. The difference is statistically significant (p-value < 0.01). When individual community smell is analysed, the similar results are found as shown in Table 5.5, 5.6 and 5.7. The results are also consistent over *medium* and *large* classes. The mean values of IFANIN are also less in smelly classes for *medium* (Table 5.9) and *large* (Table 5.10) classes. As inherited classes usually implement inherited methods, the size of classes increases. It can be the reason for not having effect of community smells

on DIT and IFANIN in small classes. Another metric NOC, which measures the number of immediate sub-classes, found significantly higher in smelly classes as reported in Table 5.4. From the above results, it is not evident that abstraction metrics are affected by community smells.

Encapsulation: The encapsulation property is measured by two metrics namely RPM and RSM. The mean values of RSM are 0.14 and 0.12 in smelly and non-smelly classes respectively. The difference is significant from the result of Mann-Whitney U test. The similar results are found in Table 5.5, 5.6 and 5.7 when community smells are considered individually. Furthermore, as shown in Table 5.8, 5.9 and 5.10, the results are consistent when classes are grouped into *large*, *medium* and *small* classes.

In case of another metric RPM, the mean values are high in both classes. That means encapsulation is violated in these classes irrespective of community smells. This can happen due to lack of communication among developers. They are not sure about which methods will be used by other classes and which will not. So, they keep those methods public thinking that others may need this functionality. On the other hand, smelly classes have more static methods than non-smelly classes which affect encapsulation property negatively.

Documentation: To measure the documentation of classes, two metrics are used which are Comment Lines of Code (CLOC) and Ratio Code to Comment (RCC). As shown in Table 5.4, the mean value of CLOC is 57.03 in smelly classes and 38.75 in non-smelly classes. As total lines of comment can vary with class size, comment ratio can better describe the documentation property. From Table 5.4, it can be seen that the mean values of RCC are 0.58 and 0.80 in smelly and non-smelly classes respectively. It is about 28% less documentation in smelly classes compared to non-smelly classes. The observed difference in RCC is also found statistically significant (p-value < 0.01).

Table 5.5, 5.6 and 5.7 show the results are similar for each three individual

community smells. When controlled for the class size, the mean values of RCC are always less in smelly classes compared to non-smelly for *small* (Table 5.8), *medium* (Table 5.9) and *large* (Table 5.10) sized classes. The result indicates that classes affected by community smells are less documented than non-smelly classes. It reduces the analyzability, modifiability and reusability of classes and thus affects software maintainability.

From the above discussions, it is observed that classes affected by community smells are more prone to change and fault. Moreover, smelly classes have a higher likelihood of having code smells. Furthermore, complexity, coupling, abstraction, encapsulation, and documentation of classes are found to be affected by community smells. All of these properties reflect that community smells have an impact on software maintainability. The findings indicate that smelly classes are less maintainable than non-smelly classes. The findings of this study will be useful for keeping a software more maintainable. The practitioners can focus on the developers who are involved in community smells and software artifacts that are affected by smells. With this information, they can better manage their resources for maintenance activities. It will be helpful to make decisions for mitigating community-related problems in the development environment.

5.4 Threats to Validity

This section discusses potential aspects that may threaten the validity of the study:

Threats to External Validity: In this study, 317 releases of 14 open-source projects are analysed to understand the impact of community smells on maintainability. The choice of these projects is guided by several factors such as Java programming language, the availability of source code repository, mailing list archive and issue tracker in public. The analyzed projects have different codebase sizes (ranges from 3,476 to 21,818 commits), different ages (ranges from 9 years to 18

years), and belonging to different application domains (DBMS, IDE, Web Framework, etc.). Despite the above diversity, the generalization can be threatened as this study only focuses to Java projects. However, further studies are desirable replicating the study on other programming languages. Moreover, this study focuses on open-source projects and the results can be different while analysing industrial projects.

Threats to Internal Validity: To detect community smell, an open-source tool, *Codeface4Smells* [45], is used. The identified smells are directly included in the analysis of this study without further verification. However, this tool is commonly used to detect community smell in related studies [4, 5, 6, 32]. This tool uses developer mailing list archives as the communication source and does not consider other communication channels, for example, Skype, Slack, etc. However, mailing list is the primary communication channel in the analysed communities according to contribution guidelines of evaluated projects. Mailing list is used commonly as the communication source in related studies as well [56, 70]. This study uses maintainability metrics such as change-proneness, fault-proneness, code smells, complexity, coupling, cohesion metrics etc., which are commonly used to assess software maintainability in previous studies [4, 17, 18, 19]. However, changing the metrics may impact the observed results.

5.5 Summary

This study conducts an empirical investigation to examine how community smells relate to software maintainability. First, community smells are detected along with involved developers by analysing communication and collaboration patterns [3]. The classes of a project are categorized based on the fact that whether those are modified by these developers. Thus, a class is defined as a smelly class if it is modified by a smelly developer otherwise non-smelly. Next, to assess the

maintainability of the software system, metrics such as change-proneness, fault-proneness, code smell, are considered [15, 16, 17, 18, 19]. Along with these metrics, *ISO/IEC 25010* defined five quality attributes such as modularity, analyzability, etc. are identified with 14 object oriented-metrics. To investigate the impact of community smell on each metric, smelly and non-smelly classes are compared to find the difference. The likelihood of having code smells is also compared between smelly and non-smelly classes.

The results suggest that classes affected by community smell exhibit more change-proneness with respect to non-smelly classes. The study also finds that classes affected by community smells are more fault-prone than classes not affected by community smells. Smelly classes are more likely to have code smells than other classes. In terms of object-oriented metrics, classes affected by community smells are found less maintainable than non-smelly classes. These results indicate that community smells have a negative impact on software maintainability. Thus, classes affected by community smells are less maintainable than non-smelly classes. The next chapter contains the concluding remarks of the whole thesis and possible future directions.

Chapter 6

Conclusion

Community smells represent poor social and organizational phenomena in the development community that can lead to the emergence of social debt. Community related aspects can influence the software development maintenance decisions such as code refactoring, bug fixing etc. [4]. The current thesis investigates how and to which extent community smells impact software maintainability. First, the involvement of developers in community smell such as missing link, is analysed to understand whether their involvement have impact on maintainability in the form of developers' contribution and bug introduction. Next, the impact of community smells on software maintainability as a whole is investigated by comparing classes affected by community smells and those that are not. The comparison is made in terms of different maintainability metrics such as change-proneness, fault-proneness, code smell, complexity, coupling, etc. In this chapter, the summary of the whole thesis is presented focusing the contributions and achievements. Finally, the possible future directions are discussed to conclude the thesis.

6.1 Involvement of Developers in Missing Link Community Smell

At first, this research investigates the relationship between developers' contribution and their involvement in missing link smell. Missing link smell occurs when developers collaborate in the source code without communicating. These smells are detected by analysing source code repositories and development mailing lists. By extracting missing link instances, developers involved in those smells are identified. Developers are divided into two categories based on their involvement, such as smelly and non-smelly. Next, the percentage of smelly developers are calculated for every project. The number of involvements in missing link are calculated for each smelly developer. The contribution of those developers to a project is measured by the number of commits. The number of total commits is counted analysing change history. Finally, correlation analysis is done between the contribution and their involvement in smell. This study also explores the relationship between missing link smells and bug introduction. To identify bug introduction, the number of Fix-Inducing Changes (FIC) are calculated by analysing source code and change histories. Furthermore, it examines the severity of bugs that are introduced in the system by the developers who are involved in missing link smell.

For this purpose, seven diverse and open-source projects from *Apache* are analysed. The results suggest that there is a moderate positive correlation between the number of commits of a developer and the number of involvements in missing link smell. The developers who contribute more tend to involve in more missing link smells. Furthermore, it is evident that there is a significant positive correlation between the number of smelly commits and FIC commits. The results reveal that developers mostly introduce major bugs in the system while involved in missing link smell. The above results indicate that the contributions of developers are affected due to their involvement in community smell. Bugs introduced in the sys-

tem by smelly developers can cause the major loss of functionality which denotes the negative impact on software maintainability.

6.2 How Community Smells and Software Maintainability Metrics Are Related

Based on the finding that community smells such as missing link can affect software maintainability in the form of developers' contribution and bug introduction, the overall impact on software maintainability is investigated in terms of different maintainability metrics. First, community smells are detected analysing communication and collaboration patterns from project repository and mailing list [3]. The developers involved in these smells are defined as smelly developers otherwise non-smelly developers. The classes of a project are categorized based on the fact whether those are modified by smelly developers. Thus, a class is defined as a smelly class if it is modified by a smelly developer otherwise non-smelly class. Next, to assess the maintainability of the software system, metrics which are commonly used in previous studies [15, 16, 17, 18, 19], such as change-proneness, fault-proneness, code smell, are considered. Along with these metrics, *ISO/IEC 25010* defined five quality attributes such as modularity, analyzability, etc. are identified with 14 object oriented-metrics. To compute these above metrics, the source code, commit history and issue history are analyzed.

After gathering all required artifacts, the experiment is carried out on a dataset which contains 317 releases from 14 open-source projects. To investigate the impact of community smell on each metric, smelly and non-smelly classes are compared using Mann-Whitney U test. The likelihood of having code smells is also compared between smelly and non-smelly classes using odds ratio, and the statistical significance is measured using Fisher's Exact test.

The results suggest that classes affected by community smell exhibit more

change-proneness with respect to non-smelly classes. The study also finds that classes affected by community smells are more fault-prone than classes not affected by community smells. Smelly classes are more likely to have code smells than other classes. In terms of object-oriented metrics, classes affected by community smells are found less maintainable than non-smelly classes. These results indicate that community smells have a negative impact on software maintainability. Thus, classes affected by community smells are less maintainable than other classes. Therefore, community-related aspects should also be considered in software maintenance activities.

6.3 Future Work

In this thesis, the relationship between community smells and software maintainability metrics has been established by conducting an empirical study using open-source projects. This research can be further extended in the following directions:

1. **Analysing industrial projects**: This research focused on open-sourced projects for the availability of the artifacts needed for analysis. The nature of communication and collaboration in closed-source or industrial projects are different from open-source projects. Being informed by the findings of the research, future studies can incorporate industrial projects to explore the impact of community smell on maintainability in those projects.

2. **Analysing non-object-oriented projects**: In this research, only object-oriented projects which use Java as the primary programming language are focused to identify the impact of community smells on maintainability. Further studies can replicate the study using other programming languages and non-object-oriented projects.

3. **Utilizing communication information other than mailing list**: In this research, communication information is collected from developer mailing lists. In recent times, many software projects use *Github* as project management system where all communications are happened [71]. Being informed by the research findings, communication data from *Github* can be used in detecting community smells to identify the impact on those projects.

Bibliography

[1] D. A. Tamburri, P. Kruchten, P. Lago, and H. van Vliet, "What is social debt in software engineering?," in *2013 6th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*, pp. 93–96, IEEE, 2013.

[2] D. A. Tamburri, P. Kruchten, P. Lago, and H. Van Vliet, "Social debt in software engineering: insights from industry," *Journal of Internet Services and Applications*, vol. 6, no. 1, pp. 1–17, 2015.

[3] D. A. Tamburri, F. Palomba, and R. Kazman, "Exploring community smells in open-source: An automated approach," *IEEE Transactions on Software Engineering*, vol. 47, no. 3, pp. 630–652, 2021.

[4] F. Palomba, D. Andrew Tamburri, F. Arcelli 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, 2021.

[5] G. Catolino, F. Palomba, D. A. Tamburri, A. Serebrenik, and F. Ferrucci, "Gender diversity and women in software teams: How do they affect community smells?," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Society (ICSE-SEIS)*, pp. 11–20, IEEE, 2019.

[6] G. Catolino, F. Palomba, D. A. Tamburri, and A. Serebrenik, "Understanding community smells variability: A statistical approach," in *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Society (ICSE-SEIS)*, pp. 77–86, 2021.

[7] I. Iso, "Iec 9126-1: Software engineering-product quality-part 1: Quality model," *Geneva, Switzerland: International Organization for Standardization*, vol. 21, 2001.

[8] S. W. Yip and T. Lam, "A software maintenance survey," in *Proceedings of 1st Asia-Pacific Software Engineering Conference*, pp. 70–79, IEEE, 1994.

[9] S. Magnoni, "An approach to measure community smells in software development communities," Master's thesis, Politecnico di Milano, Italy, 2016.

[10] B. Eken, F. Palma, B. Ayşe, and T. Ayşe, "An empirical study on the effect of community smells on bug prediction," *Software Quality Journal*, vol. 29, no. 1, pp. 159–194, 2021.

[11] F. Giarola, "Detecting code and community smells in open-source: an automated approach," Master's thesis, Politecnico di Milano, Italy, 2018.

[12] D. Spinellis, "Version control systems," *IEEE Software*, vol. 22, pp. 108–109, Sep. 2005.

[13] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?," in *Proceedings of the 2005 International Workshop on Mining Software Repositories*, MSR '05, (New York, NY, USA), p. 1–5, Association for Computing Machinery, 2005.

[14] Q. Hong, S. Kim, S. Cheung, and C. Bird, "Understanding a developer social network and its evolution," in *2011 27th IEEE International Conference on Software Maintenance (ICSM)*, pp. 323–332, Sep. 2011.

[15] F. Palomba, G. Bavota, M. Di Penta, F. Fasano, R. Oliveto, and A. De 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.

[16] W. Fenske, S. Schulze, and G. Saake, "How preprocessor annotations (do not) affect maintainability: A case study on change-proneness," *SIGPLAN Not.*, vol. 52, p. 77–90, Oct. 2017.

[17] F. Khomh, M. Di 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.

[18] A. Yamashita and L. Moonen, "Do code smells reflect important maintainability aspects?," in *2012 28th IEEE international conference on software maintenance (ICSM)*, pp. 306–315, IEEE, 2012.

[19] 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*, pp. 350–359, IEEE, 2013.

[20] B. Isong, O. Ifeoma, and M. Mbodila, "Supplementing object-oriented software change impact analysis with fault-proneness prediction," in *2016 IEEE/ACIS 15th International Conference on Computer and Information Science (ICIS)*, pp. 1–8, 2016.

[21] P. Bellini, I. Bruno, P. Nesi, and D. Rogai, "Comparing fault-proneness estimation models," in *10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'05)*, pp. 205–214, IEEE, 2005.

[22] P. Kruchten, R. L. Nord, and I. Ozkaya, "Technical debt: From metaphor to theory and practice," *Ieee software*, vol. 29, no. 6, pp. 18–21, 2012.

[23] F. A. Fontana, V. Ferme, and S. Spinelli, "Investigating the impact of code smells debt on quality code evaluation," in *2012 Third International Workshop on Managing Technical Debt (MTD)*, pp. 15–22, June 2012.

[24] M. Fowler, *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 2018.

[25] ISO, "Iso/iec 25010." `https://iso25000.com/index.php/en/iso-25000-standards/iso-25010?start=6`. Accessed: 2022-05-06.

[26] I. Heitlager, T. Kuipers, and J. Visser, "A practical model for measuring maintainability," in *6th International Conference on the Quality of Information and Communications Technology (QUATIC 2007)*, pp. 30–39, Sep. 2007.

[27] M. Monperrus, "Automatic software repair: A bibliography," *ACM Comput. Surv.*, vol. 51, jan 2018.

[28] C. Wohlin, M. Höst, and K. Henningsson, "Empirical research methods in software engineering," in *Empirical methods and studies in software engineering*, pp. 7–23, Springer, 2003.

[29] L. Zhang, J.-H. Tian, J. Jiang, Y.-J. Liu, M.-Y. Pu, and T. Yue, "Empirical research in software engineering — a literature survey," *Journal of Computer Science and Technology*, vol. 33, pp. 876–899, Sep 2018.

[30] A. Degroot, *Methodology: Foundations of Inference and Research in the Behavioral Sciences*. Psychological Studies, Walter de Gruyter GmbH & Company KG, 1969.

[31] R. Malhotra, *Empirical research in software engineering: concepts, analysis, and applications*. Chapman and Hall/CRC, 2019.

[32] F. Palomba and D. A. Tamburri, "Predicting the emergence of community smells using socio-technical metrics: A machine-learning approach," *Journal of Systems and Software*, vol. 171, p. 110847, 2021.

[33] N. Almarimi, A. Ouni, M. Chouchen, I. Saidani, and M. W. Mkaouer, "On the detection of community smells using genetic programming-based ensemble classifier chain," in *Proceedings of the 15th International Conference on Global Software Engineering*, ICGSE '20, (USA), p. 43–54, Association for Computing Machinery, 2020.

[34] N. Almarimi, A. Ouni, and M. W. Mkaouer, "Learning to detect community smells in open source software projects," *Knowledge-Based Systems*, vol. 204, p. 106201, 2020.

[35] J. M. Corbin and A. Strauss, "Grounded theory research: Procedures, canons, and evaluative criteria," *Qualitative Sociology*, vol. 13, pp. 3–21, Mar 1990.

[36] Z.-J. Huang, Z.-Q. Shao, G.-S. Fan, H.-Q. Yu, X.-G. Yang, and K. Yang, "Community smell occurrence prediction on multi-granularity by developer-oriented features and process metrics," *Journal of Computer Science and Technology*, vol. 37, pp. 182–206, Feb 2022.

[37] Z. Huang, Z. Shao, G. Fan, J. Gao, Z. Zhou, K. Yang, and X. Yang, "Predicting community smells' occurrence on individual developers by sentiments," *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*, pp. 230–241, 2021.

[38] R. B. Johnson and A. J. Onwuegbuzie, "Mixed methods research: A research paradigm whose time has come," *Educational Researcher*, vol. 33, no. 7, pp. 14–26, 2004.

[39] P. M. Blau, *Inequality and heterogeneity: A primitive theory of social structure*, vol. 7. Free Press New York, 1977.

[40] G. Catolino, F. Palomba, D. A. Tamburri, A. Serebrenik, and F. Ferrucci, "Refactoring community smells in the wild: The practitioner's field manual," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Society*, ICSE-SEIS '20, p. 25–34, 2020.

[41] Y. Zou and K. Kontogiannis, "Migration to object oriented platforms: A state transformation approach," in *International Conference on Software Maintenance, 2002. Proceedings.*, pp. 530–539, IEEE, 2002.

[42] F. Palomba, D. A. Tamburri, A. Serebrenik, A. Zaidman, F. A. Fontana, and R. Oliveto, "Poster: How do community smells influence code smells?," in *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*, pp. 240–241, IEEE, 2018.

[43] C. Spearman, "The proof and measurement of association between two things," *The American journal of psychology*, vol. 100, no. 3/4, pp. 441–471, 1987.

[44] C. Gutwin, R. Penner, and K. Schneider, "Group awareness in distributed software development," in *Proceedings of the 2004 ACM Conference on Computer Supported Cooperative Work*, CSCW '04, (New York, NY, USA), p. 72–81, Association for Computing Machinery, 2004.

[45] "Codeface4smells." `https://github.com/maelstromdat/CodeFace4Smells`. Accessed: 2022-05-06.

[46] "Diffj." `https://github.com/jpace/diffj`. Accessed: 2022-05-06.

[47] S. Kim, T. Zimmermann, K. Pan, E. James Jr, *et al.*, "Automatic identification of bug-introducing changes," in *21st IEEE/ACM international conference on automated software engineering (ASE'06)*, pp. 81–90, IEEE, 2006.

[48] S. Kim, E. J. Whitehead, and Y. Zhang, "Classifying software changes: Clean or buggy?," *IEEE Transactions on Software Engineering*, vol. 34, no. 2, pp. 181–196, 2008.

[49] S. F. Huq, A. Z. Sadiq, and K. Sakib, "Understanding the effect of developer sentiment on fix-inducing changes: an exploratory study on github pull requests," in *2019 26th Asia-Pacific Software Engineering Conference (APSEC)*, pp. 514–521, IEEE, 2019.

[50] M. G. Kendall, *Rank correlation methods*. 1948.

[51] C. P. Dancey and J. Reidy, *Statistics without maths for psychology*. Pearson education, 2007.

[52] L. M. Ingebrigtsen, "Gmane: A public mailing list archive." `http://gmane.io`. Accessed: 2022-05-06.

[53] "Jira: Issue & project tracking software." `https://www.atlassian.com/software/jira`. Accessed: 2022-05-06.

[54] L. D. Panjer, "Predicting eclipse bug lifetimes," in *Fourth International Workshop on Mining Software Repositories (MSR'07: ICSE Workshops 2007)*, pp. 29–29, IEEE, 2007.

[55] "Codeface." `http://siemens.github.io/codeface`. Accessed: 2022-05-06.

[56] M. Joblin, W. Mauerer, S. Apel, J. Siegmund, and D. Riehle, "From developer networks to verified communities: A fine-grained approach," in *IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1, pp. 563–573, IEEE, 2015.

[57] G. Catolino, F. Palomba, D. A. Tamburri, A. Serebrenik, and F. Ferrucci, "Gender diversity and community smells: Insights from the trenches," *IEEE Software*, vol. 37, no. 1, pp. 10–16, 2020.

[58] N. Moha, Y.-G. Gueheneuc, 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, 2010.

[59] N. Nachar *et al.*, "The mann-whitney u: A test for assessing whether two independent samples come from the same distribution," *Tutorials in quantitative Methods for Psychology*, vol. 4, no. 1, pp. 13–20, 2008.

[60] D. A. Tamburri, F. Palomba, A. Serebrenik, and A. Zaidman, "Discovering community patterns in open-source: a systematic approach and its evaluation," *Empirical Software Engineering*, vol. 24, no. 3, pp. 1369–1417, 2019.

[61] "Bugzilla." `https://www.bugzilla.org`. Accessed: 2022-05-06.

[62] 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, Inc., 1998.

[63] F. A. Fontana, E. Mariani, A. Mornioli, R. Sormani, and A. Tonello, "An experience report on using code smells detection tools," in *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, pp. 450–457, 2011.

[64] "Scitools." `http://www.scitools.com/documents/metricsList.php`. Accessed: 2022-05-06.

[65] S. Chidamber and C. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on Software Engineering*, vol. 20, pp. 476–493, June 1994.

[66] J. M. Bland and D. G. Altman, "Multiple significance tests: the bonferroni method," *Bmj*, vol. 310, no. 6973, p. 170, 1995.

[67] R. A. Fisher, "Statistical methods for research workers," in *Breakthroughs in statistics*, pp. 66–70, Springer, 1992.

[68] M. Szumilas, "Explaining odds ratios," *Journal of the Canadian academy of child and adolescent psychiatry*, vol. 19, no. 3, p. 227, 2010.

[69] G. Bavota, B. De Carluccio, A. De Lucia, M. Di Penta, R. Oliveto, and O. Strollo, "When does a refactoring induce bugs? an empirical study," in *2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation*, pp. 104–113, Sep. 2012.

[70] C. Bird, A. Gourley, P. Devanbu, M. Gertz, and A. Swaminathan, "Mining email social networks," in *Proceedings of the 2006 International Workshop on Mining Software Repositories*, MSR '06, (New York, NY, USA), p. 137–143, Association for Computing Machinery, 2006.

[71] A. Guzzi, A. Bacchelli, M. Lanza, M. Pinzger, and A. van Deursen, "Communication in open source software development mailing lists," in *2013 10th Working Conference on Mining Software Repositories (MSR)*, pp. 277–286, May 2013.