TAXONOMY OF BUG TRACKING PROCESS SMELLS: PERCEPTIONS OF PRACTITIONERS AND AN EMPIRICAL ANALYSIS

A THESIS SUBMITTED TO

THE GRADUATE SCHOOL OF ENGINEERING AND SCIENCE

OF BILKENT UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR

THE DEGREE OF

MASTER OF SCIENCE

IN

COMPUTER ENGINEERING

By

Khushbakht Ali Qamar

January 2022

Taxonomy of Bug Tracking Process Smells: Perceptions of Practitioners and an Empirical Analysis
By Khushbakht Ali Qamar
January 2022

We certify that we have read this thesis and that in our opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Eray Tüzün(Advisor)

Halil Altay Güvenir

İsmail Sengör Altıngövde

Approved for the Graduate School of Engineering and Science:

Ezhan Karaşan
Director of the Graduate School

ABSTRACT

TAXONOMY OF BUG TRACKING PROCESS SMELLS: PERCEPTIONS OF PRACTITIONERS AND AN EMPIRICAL ANALYSIS

Khushbakht Ali Qamar
M.S. in Computer Engineering
Advisor: Eray Tüzün
January 2022

Bug tracking is the process of monitoring and reporting malfunctions or issues found in software. While there is no consensus on a formally specified bug tracking process, some certain rules and best practices for an optimal bug tracking process are accepted by many companies and open-source software (OSS) projects. Despite slight variations between different platforms, the primary aim of all these rules and practices is to perform a more efficient bug tracking process. Practitioners' non-compliance with the best practices not only impedes the benefits of the bug tracking process but also negatively affects the other phases of the life cycle of software development.

The goal of this study is to gain a better knowledge of the bad practices that occur during the bug tracking process, that is *bug tracking process smells*. In this study, based on the results of a multivocal literature review, we analyzed 60 sources in academic and gray literature and propose a taxonomy of 12 bad practices in the bug tracking process, that is *bug tracking process smells*. To quantitatively analyze these process smells, we inspected bug reports collected from six projects (four of them are Jira-based and the other two are Bugzilla-based). To get an idea about the perception of practitioners about the taxonomy of bug tracking process smells, we conducted a targeted survey with 30 software practitioners from different countries. Moreover, we statistically analyzed the impact of bug tracking process smells on the resolution time and reopening count of bugs.

We observed from our empirical results that a considerable amount of bug tracking process smells exist in all projects and some of the process smell categories have a statistically significant impact on quality and speed. Survey results

showed that the majority of software practitioners agree with our taxonomy of bug tracking process smells. The empirical analysis reveals that bug tracking process smells have a significant impact on OSS projects. In practice, the proposed taxonomy may serve as a foundation for best practices and tool assistance for detecting and avoiding bug tracking process smells.

Keywords: the bug tracking system, process mining, conformance checking, anti-patterns, bug tracking smells, process smell.

ÖZET

HATA TAKIP SÜREÇLERINDEKI KÖTÜ UYGULAMALARIN SINIFLANDIRILIMASI: YAZILIM GELIŞTIRICILERIN ALGISI VE DENEYSEL ANALIZ

Khushbakht Ali Qamar
Bilgisayar Mühendisliği, Yüksek Lisans
Tez Danışmanı: Eray Tüzün
Ocak 2022

Hata takibi, bir yazılımda bulunan problemleri izleme ve raporlama sürecidir. Literatürde kabul görmüş bir hata takip süreci olmasa da, en işlevsel hata takip süreci için bazı kurallar ve kabul gören uygulamalar, birçok şirket ve açık kaynaklı yazılım projesi tarafından kullanılmaktadır. Farklı platformlar arasında küçük farklılıklar olsa da tüm bu kural ve uygulamaların temel amacı daha verimli bir hata takip süreci gerçekleştirmektir. Yazılım mühendislerinin, bu kabul gören uygulamaları izlememesi, yalnızca hata takip sürecinin faydalarını azaltmakla kalmaz, aynı zamanda yazılım geliştirme yaşam döngüsünün diğer aşamalarını da olumsuz etkiler.

Bu çalışmanın amacı, hata takip sürecinde meydana gelen kötü uygulamaları yani hata takip süreci kokularını incelemektir. Bunun için, çok sesli bir literatür taramasının sonuçlarına dayanarak, akademik ve gri literatürdeki 60 kaynağı analiz ettik ve hata takip sürecindeki 12 kötü uygulamayı içeren bir taksonomi önerdik. Kötü uygulamaları nicel olarak analiz etmek için 4'ü Jira tabanlı, 2'si Bugzilla tabanlı olmak üzere 6 projeden toplanan hata raporlarını inceledik. Ardından, yazılım mühendislerinin, bu taksonomi hakkındaki düşüncelerini öğrenmek için farklı ülkelerden 30 yazılım profesyoneli ile bir hedefli anket gerçekleştirdik. Ayrıca, kötü uygulamaların, bu uygulamalardan etkilenen hatalardaki kapanma süresi ve yeniden açılma sayısını istatistiksel olarak nasıl etkilediğini analiz ettik.

Ampirik sonuçlara dayanarak, tüm projelerde önemli sayıda kötü uygulamanının bulunduğunu ve bazı kötü uygulamaların kalite ve hız üzerinde istatistiksel olarak önemli etkileri olduğunu gözlemledik. Anket sonuçları, yazılım profesyonellerinin çoğunun kötü uygulama taksonomimizle aynı fikirde olduğunu

gösterdi. Ampirik analiz, hata takibindeki kötü uygulamaların açık kaynak kodlu projeler üzerinde önemli bir etkiye sahip olduğunu ortaya koymaktadır. Bu çalışmada önerilen taksonomi, hata takip sürecinde kötü uygulamaları tespit etmek ve önlemek için geliştirilecek olan araçlara bir temel oluşturabilir.

Anahtar sözcükler: hata takip sistemi, süreç madenciliği, uygunluk kontrolü, anti-kalıplar, hata takip kokuları, süreç kokusu.

Acknowledgement

I would like to express my gratitude to my advisor Asst. Prof. Dr. Eray Tuzun. His continuous motivation, supervision and encouragement throughout my degree program and specifically through this thesis were impetus for my hard work. His guidance helped me to develop both professionally and personally. I would extend thanks to my jury members Prof. Dr. Halil Altay Güvenir and Assoc. Prof. Dr. İsmail Sengör Altıngövde for giving their invaluable feedback on this thesis.

This thesis would not have been possible without the help and support of my parents Muhammad Ali and Naheed Ali. My siblings played a crucial role in this regard who not only supported me in my endeavors but also made my life away from home easier. My friends in Bilkent deserve a special thanks who kept me cherished through my hard times and uplifted my morale.

Emre Sulun merits special recognition for his contribution in this thesis as well as all other members of BILSEN (Bilkent University Software Engineering and Data Analytics Research Group). Their input throughout my thesis journey was extremely precious. Lastly, I would like to thank TUBITAK. This research was supported by The Scientific and Technological Research Council of Turkey (TUBITAK) 1505 program. Project Number: 5200078.

Contents

1 Introduction **1**

1.1 Research Problem 2

1.2 Contributions of the Thesis 3

2 Background & Related Work **5**

2.1 Bug Tracking 5

2.2 Related Work 8

3 Research Methodology **12**

3.1 Multivocal Literature Review (MLR)- Creating Taxonomy 12

3.1.1 Search Strategy 13

3.1.2 Exclusion/Inclusion Criteria & Quality Assessment 14

3.1.3 Data Extraction & Final Pool of Sources 15

3.2 Empirical Analysis 15

3.3 Software Practitioner's Survey 16

3.4 Hypothesis Testing 17

4 Taxonomy of Bug Tracking Process Smells 18

4.1 Unassigned Bugs 18

4.2 No Link to Bug-Fixing Commit 20

4.3 Ignored Bugs 21

4.4 Bugs Assigned to a Team 22

4.5 Missing Priority 23

4.6 Not Referenced Duplicates 24

4.7 Missing Environment Information 25

4.8 Missing Severity 26

4.9 Reassignment of Bug Assignee 27

4.10 No Comment Bugs 28

4.11 Non-Assignee Resolver of Bug 30

4.12 Closed-Reopen Ping Pong 31

5 Empirical Analysis 34

5.1 Dataset 34

5.2 Data Cleaning and Preprocessing 35

5.3 Mining Bug Tracking Process Smells 37

5.4 Comparison of Bug Tracking Tools 41

5.5 Time-Based Analysis 42

6 Synthesizing Practitioner's Survey 43

6.1 Unassigned Bugs 45

6.2 No Link to Bug-Fixing Commit 46

6.3 Ignored Bugs 47

6.4 Bugs Assigned to Team 49

6.5 Missing Priority 50

6.6 Not Referenced Duplicates 51

6.7 Missing Environment Information 52

6.8 Missing Severity 52

6.9 Reassignment of Bug Assignee 53

6.10 No Comment Bugs 55

6.11 Non-Assignee Closer of Bug 56

6.12 Closed Reopen Ping-Pong 58

7 Hypothesis Testing 60

7.1 TTR Vs Process Smells 62

7.2 Reopen Count Vs Process Smells 64

8 Discussion **66**

8.1 Context Dependency of Smells 66

8.2 Practical Advice to Practitioners 67

9 Threats to Validity **69**

10 Conclusion and Future Work **72**

A Literature Sources **83**

B Code and Reproducibility **86**

List of Figures

2.1 Activity Diagram for the Bug Tracking Process. 10

2.2 State Diagram for the Bug Tracking Process. 11

3.1 Research Methodology Followed in This Study. 13

3.2 Flow Diagram of MLR 14

5.1 Empirical Evaluation Steps 36

5.2 Heatmap of Smell Ratio 38

5.3 Smell Ratio Change of All Projects by Year 39

5.4 Smell Ratio Change of Wireshark Project 40

5.5 Smell Ratio Change of Confluence Server & Data Center Project . 40

5.6 Percentage of Bug Affected by Different No. of Smells 41

6.1 Distribution of answer to the question: *How often do you encounter a process smell in your company?* 44

6.2 Distribution of answer to the question: *Are there any actions taken to prevent smell in your Company?* 46

List of Tables

4.1 Taxonomy of Bug Tracking Process Smells 33

5.1 The details of projects 35

5.2 Number of Bug Reports that have the corresponding smells for each project (B: Bugzilla, J: Jira) 36

6.1 Survey Results 59

7.1 p-values for statistical tests of TTR and smelling bug instances for six projects. Expected Results (Green Color), Unexpected Results (Yellow Color) 61

7.2 p-values for statistical tests of reopen count and smelling bug instances for six. Expected Results (Green Color), Unexpected Results (Yellow Color) 64

A.1 MLR Results with the Related Smells 84

A.2 MLR Results with the Related Smells 85

Chapter 1

Introduction

Bug tracking (BT) is a methodology for reporting and keeping track of the bugs in software products. It lets developers refine the software design further by making continuous updates or adjustments to the software product. In a Bug Tracking System (BTS), each item is followed and tracked through to completion with informative reports which display the progress. Users might potentially neglect the bugs found in the system and ignore them so it is important to keep track of all the bugs. BT's stakeholders are consumers, software developers, project team members, testers, and board members. Developers can continue to work on the next bugs when they have completed a task, whilst the Quality Assurance (QA) team will start testing. In large-scale software projects, the number of bugs can be vast thus piling up a large amount of work which can prolong the fixing task and affect the quality of task execution. While there is no agreement on a formally specified BT process, some certain rules and best practices for an optimal BT process are reported in both white [1–3] and gray literature[1]. Practitioners' non-compliance with the best practices impedes the benefits of the BT process and should be avoided.

[1] `https://bugs.eclipse.org/bugs/page.cgi?id=bug-writing.html`

1.1 Research Problem

To collect the set of deviations from the best practices, we explore the bad practices that developers follow throughout the BT process, in this study. To denote these bad practices, we use the term *bug tracking process smells*. Some of the issues in the BT process have been referred to in previous work from various perspectives, such as reassignment of fields [4] and bug reports without comments [1]. Gupta [5,6] proposes a framework called Nirikshan to observe inconsistencies between the runtime process model (real-life model) and the design-time model (ideal model) within the bug life cycle of an open source project. However, none of them have gathered the bad practices followed during the BT process with a systematic approach. We believe that this set of bad practices would be valuable for software practitioners to identify possible bottlenecks or problem areas in the bug tracking process. To explore these bad practices further, we identify the following research questions within our study:

RQ1- What are the observed bad practices followed by developers during the bug tracking process?

To address this, we scanned academic and gray literature. We reviewed the studies that address bad practices (anti-patterns) and problems encountered during the BT process. Afterward, we proposed a taxonomy of BT process smells to illustrate the cases where the developers do not conform to the ideal BT process.

To show quantitative evidence for the BT process smells gathered in our taxonomy, the following research question is defined:

RQ2- How frequently does each BT process smell occur in practice?

To answer this research question, each BT process smell is empirically evaluated by mining BT histories of six projects; two Bugzilla Projects (Wireshark, GCC), and four Jira Projects (MongoDB Core Server, Evergreen, Confluence Server & Data Center, Jira Server & Data Center).

Previously, we reported our initial empirical analysis of BT process smells in [7]. In the thesis, we extended our prior work with the addition of two more RQs which explain the perception of software practitioners about our proposed taxonomy of BT process smells and the statistical impact of these process smells on important bug tracking quality measures such as time to resolve a bug and reopening count of a bug. To explore the perception of software practitioners about BT process smells, the following research question is defined:

RQ3- What is the perception of software practitioners about the bug tracking process smells?

To answer this research question, we conducted a survey with 30 software practitioners who have experience with the bug tracking process. From survey results, we get an expert opinion about BT process smells and get an idea if there are any actions taken in different companies to avoid these process smells.

Afterward, we also wanted to analyze if these process smells affect the BT process negatively. For this reason, we statistically analyzed the effect of factors affecting the BT process like time to resolution (TTR) of a bug and reopening count of a bug. This leads us to define our final research question;

RQ4- Is there any effect of process smells on the TTR and reopening count of bugs?

To answer this, we statistically analyzed the effect of type of process smells on the TTR and reopening of bugs. We used statistical tests to analyze if the effect is statistically significant or not.

1.2 Contributions of the Thesis

The main contributions of this thesis are:

- Proposed a novel taxonomy of BT process smells (Table 4.1), based on a

multivocal literature review.

- Performed an empirical analysis with six OSS projects to demonstrate that all the process smells occur in software bug reports with varying ratios.
- Observed that over time, the occurrence of some specific BT process smells in software projects is decreased. The reason for this improvement might be associated with the advancements in BT tools and improved best practices for the BT process.
- Each process smell is assessed in our study by obtaining the opinions of domain experts i.e. software practitioners.
- Analyzed the statistical impact of process smells on TTR and reopen count of bugs.

The rest of this thesis is organized as follows. The following chapter discusses the background of BT process and related work in our research field. Chapter 3 explains our research methodology. Chapter 4 demonstrates our BT process smells taxonomy and the technique for detecting each type of smell. Chapter 5 gives the empirical evaluation of BT process smells on six projects. Chapter 6 describes the perception of software practitioners about the proposed taxonomy of BT process smells. Chapter 7 explains the hypothesis testing we have done in order to analyze the impact of BT process smells. Chapter 8 discusses the context-dependency of BT process smells and practical advice to the software practitioners to avoid the smells during BT process. Chapter 9 addresses the validity threats of this study and lastly, Chapter 10 presents our conclusion and future work.

Chapter 2

Background & Related Work

2.1 Bug Tracking

Although there are many processes involved in developing software, the following is the standard process used by most organizations:

- Identifying user's needs
- Designing a solution
- Coding the product
- Testing the code (internal bug reporting)
- Launching the product
- Gathering user feedback (external bug reporting)

Bug Tracking is a process of streamlining and keeping track of the reported software issues and disorders. For those participating in issue fixing, a good bug tracking method is self-explanatory and consistent. In a system, there are two categories [1] of bugs that are reported.

[1] https://kissflow.com/workflow/issue-tracking/bug-tracking-process/

1. **Internal Bugs:** Before a product or a solution is released, developer and quality assurance officers test the code for bugs. Using this approach, the tester explores all possible use cases to see how the software would perform and find possible errors.

2. **External Bugs:** After launching a product, companies often implement a way for users to report errors they encounter while using the product.

In order to check the conformance of developers to the bug tracking process, a generic bug tracking process model needs to be defined as a baseline. Therefore, we conducted a grey literature review on the bug tracking process. Figure 2.1 and 2.2 illustrates the activity and state diagrams of the generic code review process respectively. The bug tracking cycle begins when the users or quality assurance testers spot an issue in the running program, or the developers spot an issue in the code. By combining all perspectives, the following generic process model of bug tracking is defined as follows

1. Whenever a bug is reported, it is compared to other existing bugs to check for duplicates, it is run through the Bug Tracking Software's (BTS) bug database to check if any such bug is already open.

2. If a duplicate is not found, a new bug is opened. To open the unique bug in an ideal manner, the following are done in no particular order. However, they are all essential and must be done before finishing the creation of an issue.

    - Title & Description
    - Issue code
    - Tags/Labels
    - Priority
    - Severity
    - Assignment & Access Limiters
    - Reviewer

- Project/Product Association
- Project Milestone
- CC list
- Deadline
- Deployment Prerequisites/Dependency Clearance

3. After a bug is opened and a developer is assigned to it, this developer starts working on the bug to try and solve/implement it.
4. When the assigned developer believes he/she is done with the bug, he/she submits it for review for the reviewer.
5. If the bug fails the review, it is reopened with feedback from the overseer, after which the developer fixes the problems with the bug using said feedback, and submits the bug for review once again.
6. If the bug passes the review, it is deployed/merged to its parent branch, after which it goes through a series of tests to measure the robustness of its integration to this parent branch.

A bug life cycle describes the workflow of a bug from its creation to its resolution [8]. Figure 2.2 shows the default workflow of Bugzilla [2] where the vertices represent the states and the edges represent the transitions between each state. When a bug is submitted, its state is either New or Unconfirmed. A new bug from a user who has permission to create a bug directly to the system or a product without an Unconfirmed state is registered as New while others are registered as Unconfirmed. When a bug is proven or obtains enough votes, its status moves from Unconfirmed to New. The status of a bug that is New or Unconfirmed switches to Assigned when a developer is assigned to it. When a developer successfully fixes a bug, the status changes to Resolved. The bug's status changes to New when the developer stops working on it. When a bug is not genuine or is repaired willingly before it is assigned to someone, unconfirmed and new bugs can also shift to the Resolved status directly. Finally, the status of the resolved

[2]https://www.bugzilla.org/docs/3.6/en/html/lifecycle.html

bug switches to Verified once it has been verified by the quality assurance (QA) team. The bug status switches to Reopen if the quality assurance team is not happy with the solution or if the bug recurs. A bug that has been reopened can be assigned and resolved again. If a bug in the Resolved, Verified state is reopened and never validated, it will revert to the Unconfirmed state. When a bug has been verified or resolved, it is marked as closed.

2.2 Related Work

Software maintenance is not only intended to repair bugs but also to enhance performance and improve software functionality, leading to improved software quality. Software artifacts, notably bug reports have become an important medium for helping developers in resolving bugs. There have been various studies on detecting the patterns in the BT data.

The vast number of process logs from real-life software projects recently have enabled mining these processes to find cases in reality that contradict the ideal process definitions. Gupta and Sureka [5,6] put forward a framework called Nirikshan that will help to observe any anomalies between the runtime process model (real-life model) and the design-time model (ideal model) within the bug life cycle of an OSS project. In the study of Gupta et al. [9], the Chromium project's bug life cycle is elaborated by issue tracking, mining peer code review, and version control systems. Additionally, certain deviations from the optimal method and bottlenecks are detected and diagnosed during the life cycle. Halverson et al. [10] worked on the visualization of state changes and unveil the problematic bug patterns like multiple reopen/resolve cycles. They proposed a visualization that emphasizes the social aspects of bug reports and change requests. D'Ambros et al. [11] used data from a release history database for the visualization of a bug's life-cycle. Their tool focuses on the activity, time, and priority/severity features of the bug. The advantage of such a representation is the ease with which you can switch between an overview and a thorough examination of a single issue. An

interactive approach was introduced by Knab et al. [12] for visualizing the patterns of process life-cycle and effort estimations using event logs and bug reports in BTS to detect flaws, outliers, and other interesting properties.

Other methodologies focus on the quality of bug reports, Ko et al. [13] provided finer tool support for bug reporting. They analyzed the linguistic attributes of the bug report descriptions for the improvement of bug reports' quality. Hooimeijer et al. [14] presume that high-quality bug reports are addressed faster as compared to low-quality bug reports. Based on this presumption they anticipate if a bug is resolved in a specified time by using different bug report characteristics, e.g., submitter reputation, readability, severity, etc. 'Who should fix this bug' by Anvik et al. employ data mining techniques to propose prospective developers to whom a bug should be assigned [15] . The same authors discuss the issues that arise when using open bug repositories, such as irrelevant and duplication [16]. Recently, Dogan and Tuzun [17] used the same approach as ours i.e. mixed-method approach and provided the taxonomy of code review smells. Their study indicates that code review smells have a significant impact on open source projects. These code review smells could compromise some of the software development community's most important characteristics, such as productivity, team assessment, and code quality.

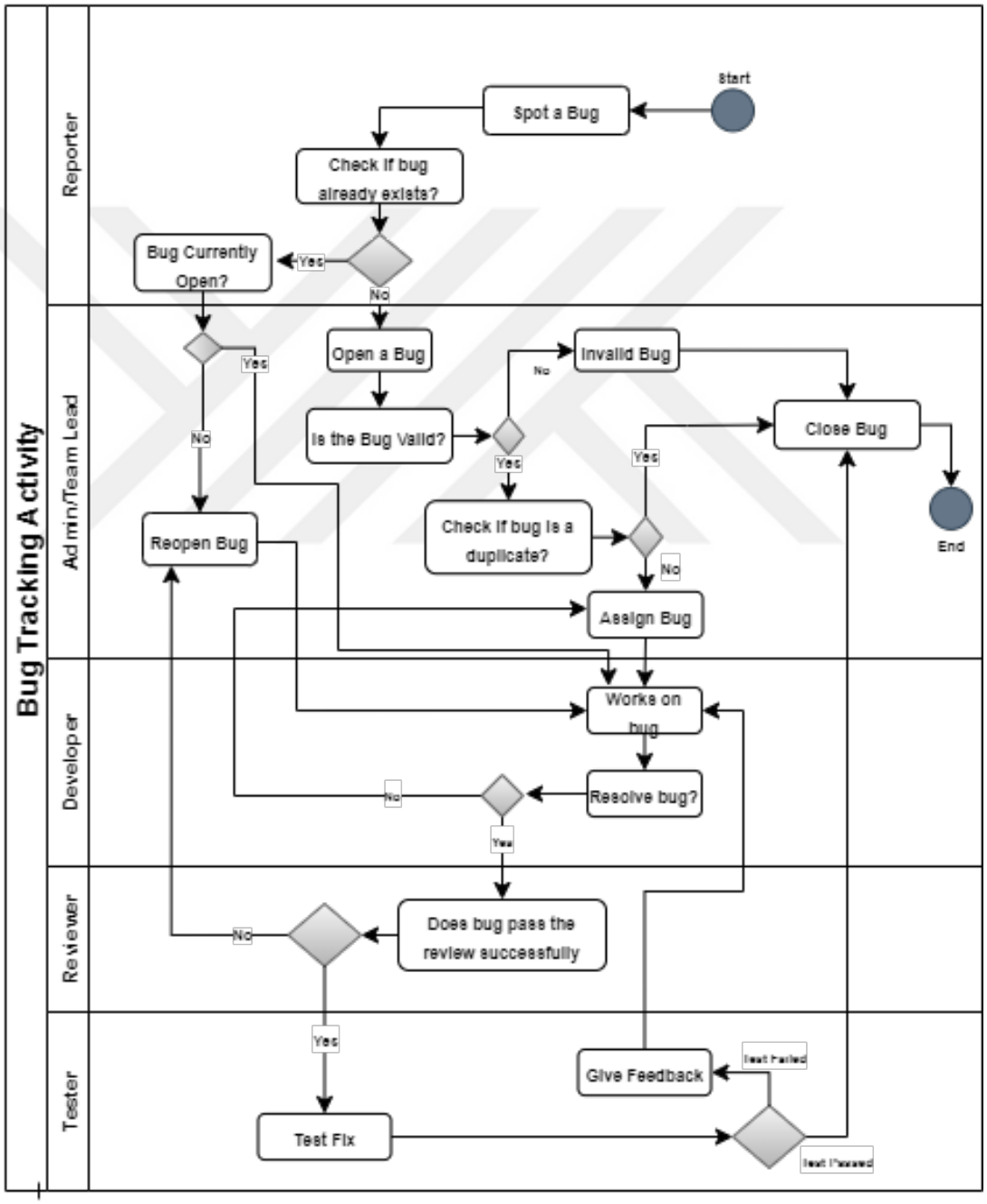

Figure 2.1: Activity Diagram for the Bug Tracking Process.

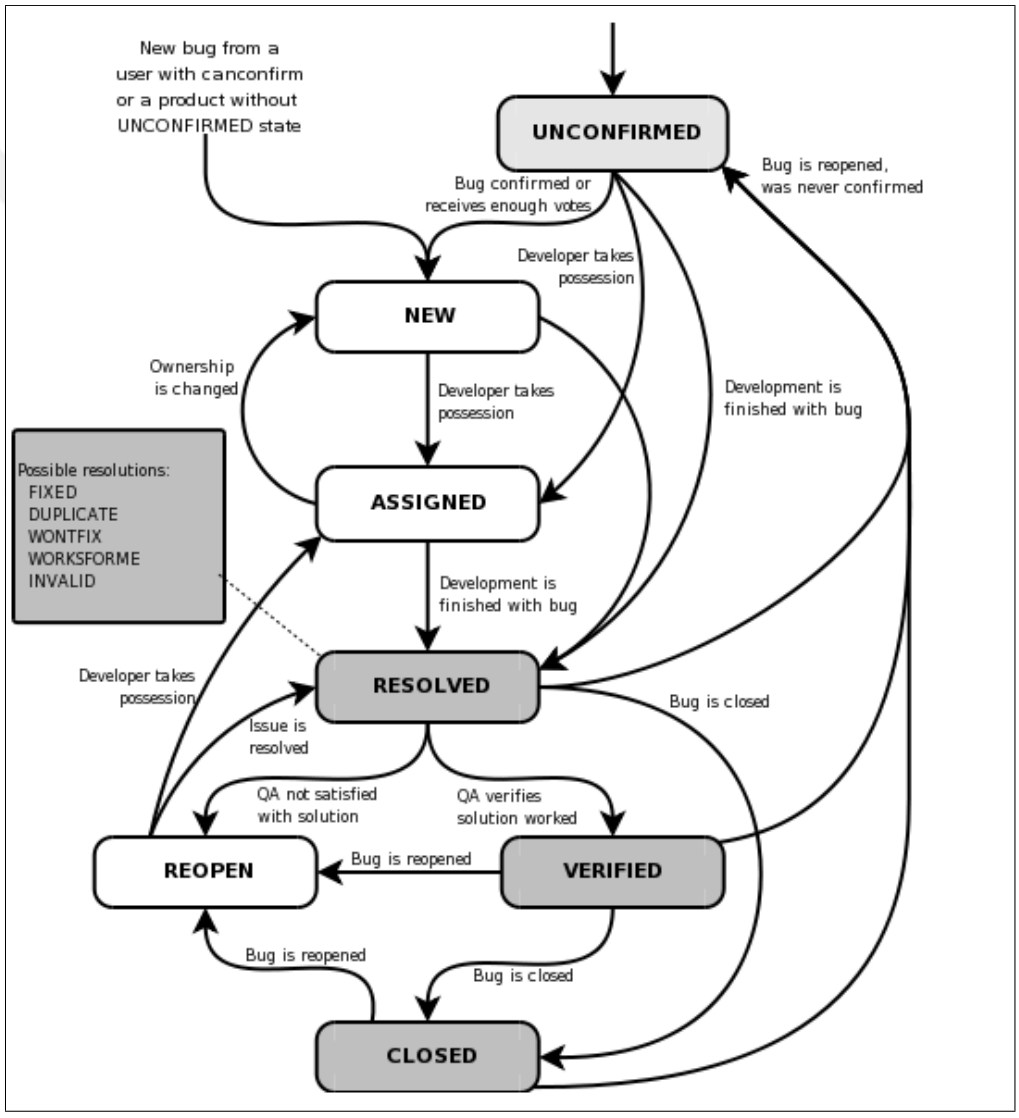

Figure 2.2: State Diagram for the Bug Tracking Process.

Chapter 3

Research Methodology

In our study, we follow a mixed-methods-based approach. We used this research methodology to support our qualitative analysis with quantitative results [18]. Figure 3.1 depicts an overview of the research approach used in this study. The objective of this study is to recognize the bad practices (smells) in the BT process (RQ1) and to perform a quantitative analysis on BT data (RQ2) to show that these smell categories exist in bug tracking systems (further details of the study setup are expanded in Chapter 5). Moreover, to know the perception of software practitioners, we conducted a survey (RQ3), and to observe the effect of process smells on BT process, a statistical analysis is performed (RQ4).

3.1 Multivocal Literature Review (MLR)- Creating Taxonomy

We selected MLR as a review method following the guidelines of Garousi et al. [19]. Generally, in a systematic literature review (SLR), the search is only limited to the academic literature [20]. However, it is seen that many software developers do not publish their work formally in academic forums [21], that is to say, if we are not considering the gray literature (GL), then we might limit the

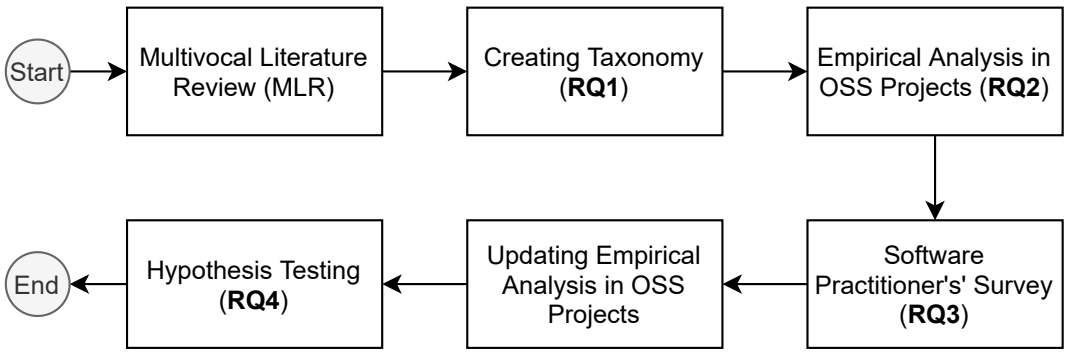

Figure 3.1: Research Methodology Followed in This Study.

voice of practitioners. Thus, we included GL (blog posts, white papers, articles, etc) as well. We believe that the usefulness and relevance of our study for both industrial and academic points of view (practitioners and researchers) would be increased [22] by including the GL. Therefore, we constructed the literature search in two major steps.

- First, we conducted an SLR and performed a search using a search string in Google Scholar, then used those as a starting point for backward and forward snowballing of academic literature.
- Second, Google Search Engine was used to search for relevant GL sources.

The flow diagram of our MLR is shown in Figure 3.2.

3.1.1 Search Strategy

Based on our research goal and the RQs, we defined the search strings. Then, we looked up the searches in Google Scholar for formally published academic papers, and for GL we used the Google search engine. The initial search results for academic papers gave us 1704 results related to our search string. The search string for academic literature is as follows.

(“***Issue***” *OR* “***Bug***” *OR* “***Defect***”) *AND* (“***Tracking***” *OR*

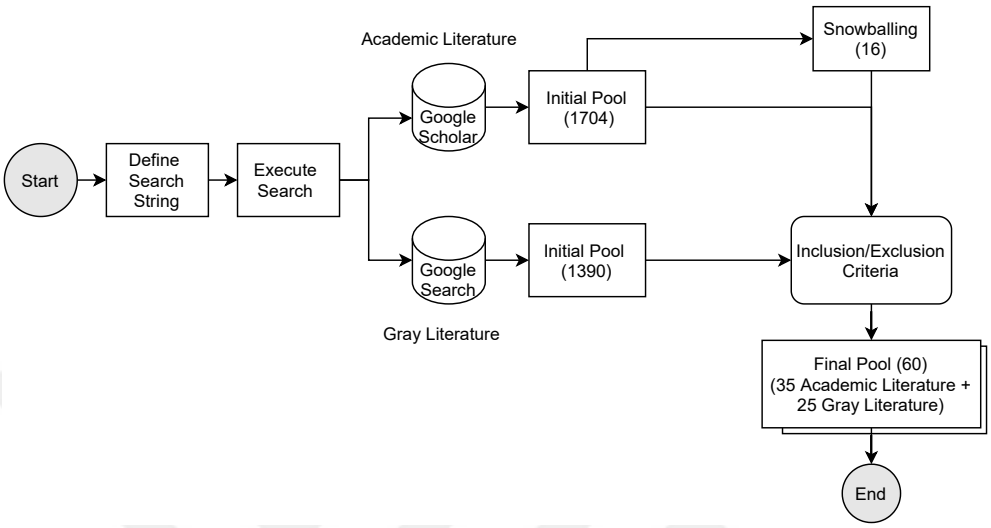

Figure 3.2: Flow Diagram of MLR

(“***Management***”) *AND* (“***Bad Practice***” *OR* “***Smell***” *OR* “***Anti-pattern***”)

For GL, we searched for the terms “bug tracking (219 results)”, “bug management (208 results)”, “defect tracking (230 results)”, “defect management (255 results)”, “issue tracking (234 results)”, and “issue management (244 results)” and their back-links on Google Search. We came up with a total of 1390 sources in the initial pool.

To include all the relevant sources, we conducted snowballing, as suggested by SLR guidelines [20], on the set of selected papers in our initial search pool. We carefully read each paper and looked for the potential bad practices followed during the BT process.

3.1.2 Exclusion/Inclusion Criteria & Quality Assessment

We defined the exclusion and inclusion criteria carefully to ensure that we included all the sources that are relevant to our study and excluded all the out-of-scope sources. For the quality evaluation of sources, we followed the checklist proposed by Garousi et al. [19].

As inclusion criteria, we searched for studies describing the non-ideal or bad practices followed during the BT process. We included studies that discuss the anti-patterns in BT and deviation from the ideal BT process. As an exclusion criterion, we eliminated the studies written in a language other than English. We also only considered the relevant sources with clear author information (anonymous sources are eliminated) in *Gray Literature*.

3.1.3 Data Extraction & Final Pool of Sources

For academic literature, we came up with a list of 35 primary studies after applying inclusion/exclusion criteria. Each resulting paper was mapped to the related BT process smell. For GL, we shortlisted 25 sources and each source is referring to at least one smell/bad practice. We have provided the finalized sources list online[1] for ensuring the transparency of our study.

3.2 Empirical Analysis

After MLR, a taxonomy of BT process smells is finalized. Then, each process smell is evaluated on six OSS projects using Jira and Bugzilla as their BT tool. While selecting the projects, in addition to the data availability criteria, we considered the tool and organization diversity; the projects belong to four different organizations(Atlassian, Mongo, GNU Project, and Wireshark Team) and use two different issue tracking tools (Jira and Bugzilla). Chapter 5 goes into the study's setup in further detail.

[1] `https://figshare.com/s/f8ce1820d9a371a73071`

3.3 Software Practitioner's Survey

RQ3 is related to the perception of software practitioners about our taxonomy of bug tracking process smells. For this purpose, we conducted a survey with software practitioners. Following the MLR, we created an extensive survey [2] for the software practitioners who are actively taking part in BT process, using the guidelines from Kitchenhem and Pfleeger [23]. The aims of conducting this survey are:

- To see if the definitions of BT process smells are acceptable to software practitioners.
- To get feedback on the detection methods of process smells (specifically, thresholds for classifying an instance as BT process smell).
- To get an experts' opinion on the potential side effects and root causes of each process smell.
- To get information about the actions taken by different software companies to avoid these process smells.

We first piloted our survey with 6 respondents from our research group. These 6 responses helped us in refining the survey and also, in validating the survey's ability to answer our RQ. To complete our survey, we contacted a total of 60 software practitioners (Developer, Team Lead, QA) from authors' professional and personal networks. The survey was completed by 30 software practitioners. All respondents work for software organizations around the globe (e.g. Turkey, Pakistan, Denmark, Sweden). We used convenience sampling to pick the participants rather than relying on a public survey. Our survey is quite an extensive survey, as it covers all 12 bug tracking process smells. It takes around 45-50 minutes on average to complete our survey.

The respondents have an average software development experience of 8 years and bug tracking experience of 6.2 years. Furthermore, each process smell is

[2] https://figshare.com/s/f8ce1820d9a371a73071

defined in detail in the survey, along with a real-life example. The respondents are then asked a series of questions on how familiar they are with each process smell. We also requested their feedback on a set of 'thresholds' we employed in our empirical study. Before moving on to the analysis, we double-checked the accuracy of the responses. Chapter 6 goes into the survey's setup in further detail.

3.4 Hypothesis Testing

To analyze the effect of process smells on time to resolution (TTR) and bug reopening, we propose to use hypothesis testing (statistical tests). We used *Mann-Whitney U Test* [24] which is a non-parametric statistical test. The reason is our data is not normally distributed and we are comparing two independent groups (i.e. Bugs with process smell and Bugs with no process smell). Further details about hypothesis testing are explained in Chapter 7.

Chapter 4

Taxonomy of Bug Tracking Process Smells

RQ1 is about compiling the bad practices which are being followed by practitioners during the BT process. MLR was conducted and we observed the potential bad practices which are not complying with the ideal flow of the BT process. This leads us to 12 BT process smells. A summary of the proposed BT process smells taxonomy is given in Table 4.1. In this chapter, each smell is introduced in the following format: (1) a detailed description of the smell (2) potential root causes (3) our smell detection methodology.

4.1 Unassigned Bugs

Each reported bug must be triaged to decide if this bug describes a significant and novel enhancement or problem, it must be assigned to an appropriate developer for further investigation [25]. Once a bug is assigned, it is resolved according to the priority, severity, and workload of the developers after the bug status is set as open. However, this is a very time-consuming process, research has been done to automatically assign a competent developer to the new bug reports [26, 27].

Bug assignment lets users focus on their assigned duty and ease the process of tracking a bug life cycle. However, it is observed that there are bugs that have no assignee at all even if the bug is resolved. This is a potential indicator that the BT process is not followed properly [28]. Thus, we consider it a process smell and call it an *unassigned bug*. Potential *impacts* of not assigning a developer to a bug could be loss of traceability [29] and delays in bug resolution [15].

Potential Root Causes:

- *Availability Reasons:* The triager could not find an available developer (depending upon the developer's workload) at that moment and opened a bug without assigning any developer, then, later on, forgot to update it.
- *Time Pressure:* Assigning developers to the bug is a very time-consuming and laborious task. So, finding a resource with relevant expertise is not that simple; it could cause unassigned bugs in the system.
- *Unknown Developer's Expertise:* The triager might not be knowing about an expert developer with the relevant expertise in bug resolution and in that dilemma, triager missed assigning the developer for bug resolution.
- *BTS Lack Embedded Developer Recommendation:* Developer recommendation systems have been introduced by researchers to automate developer assignments, but it could be possible that an organization is using the BTS that lacks a Developer Recommendation System. So, it might result in unassigned bugs.

Smell Detection Method:

First, we check whether the bug is fixed and closed. If so, we check whether the assignee field is empty or not. However, there are also some cases where the assignee field is not empty but an invalid email address is written. For instance, if *unassigned@gcc.gnu.org* address is used as an assignee email, we consider this case a smell.

4.2 No Link to Bug-Fixing Commit

In software engineering, software artifacts especially commit logs and bug reports are widely used. To obtain useful information about a software project's evolution and history, bug reports and commit logs should be integrated. Ideally, in a BTS, all bug-fixing commits should be linked to their respective bug reports [30]. If a proper bug is closed without any link to the bug-fixing commit, then in the future, it will be difficult to discover what happened with that bug. In our case, we see this as a process smell and we call this *No Link to Bug-Fixing Commit*. The potential *impact* of this process smell is losing tracks of the bugs and eventually the traceability of the bug decreases [31]. It also affects the related software development tasks such as prediction of bug locations, recommendation of bug fixes, and software cost [31].

Potential Root Causes:

- *Weak Understanding of Software Development:* It could be possible that the underlying software development interactions are not understandable by the team members.
- *Committing the link is not straightforward:* It could be possible that some ineffective linking technique is being employed in the BTS being used by an organization.

Smell Detection Method:

First, we check whether the bug is fixed. If so, we check the comments and designated fields to find a link to the version control system.

4.3 Ignored Bugs

In a BTS, once the bug is opened, it should not be left unattended or open for a long time. The knowledge of the bug may be forgotten over time [32]. Even if it is not closed, some progress should be made to resolve the bug. However, it is seen that there exist some bugs which are neither assigned to anyone nor resolved. They are left in an idle state for a long time (three months or longer). Another scenario could be an *Incomplete* resolution of bugs. It is observed that some bugs are marked as *Incomplete* resolution states. No information is given in the bug report about what has been done to make such bugs incomplete. For both the above-mentioned scenarios, we call this *Ignored Bugs* smell. The potential *impact* of this process smell is to create a delay in the bug resolution process [33].

Potential Root Causes:

- *Vague Bug Description:* It could be that the reporter did not provide the necessary and complete information about the bug due to which the developer could not be able to solve that bug completely.
- *Incorrect Prioritization:* Another possible root cause could be incorrect prioritization of bugs by the reporter due to which the developer might think of fixing it later but that bug is never resolved again and is ignored.
- *Reassignment of Bug:* Another reason could be that the fixer is reassigned to a bug and with the passage of time the bug loses its information track and then remained in the ignored state.

Smell Detection Method:

We compare the dates of sequential activities in the bug history. While the bug is not resolved, if there is a gap longer than three months between any two activities, we consider it a smell.

4.4 Bugs Assigned to a Team

During the bug resolution process, a bug must be assigned to a particular developer so that it could be resolved. During our analysis, we observed that some of the bugs are assigned to a team rather than a particular developer. Moreover, in some of those cases, it is also observed that these bugs had no fixes or any comments in a short time, which is a deviation from the ideal BT process. However, in distributed environments, a network of people may cause the dispersal of responsibility as *everybody's problem becomes nobody's responsibility* [34]. Whenever a bug is assigned to any team, and it is not specified which member of the team is going to solve that bug; it becomes everyone's problem but no one has its responsibility individually. We call these *Bugs Assigned to a Team* smell. Potential *impacts* of this process smell could be loss of traceability and accountability of bugs [35]. The problem of *team bug-assignment* was introduced by Jonsson et al., [35, 36] in which instead of a single developer, bugs are redirected to one of the many accessible teams.

Potential Root Causes:

- *System Knowledge:* It could be possible that in large organizations that the triager lacks detailed knowledge of the overall system. Due to which bug reports might route wrongly within an organization and cause a delay in bug resolution.
- *Mistakenly Assigned by a Triager:* It could be possible that the triager has assigned the bug report to the developer who has no expertise with that particular type of bug.
- *Availability of Developer:* Another possible reason could be that at the time of assignment developers are not available, so the triager assigned the bug to any team.

Smell Detection Method:

First, we check whether the bug is assigned. If so, we search for the selected keywords: "team", "group" and "backlog" in the assignee field. We found those keywords by manually inspecting the assignee names in each project. For example, in MongoDB Core Server project history, there are some bugs assigned to *Backlog - Sharding Team*, we consider it a smell.

4.5 Missing Priority

During the life cycle of a bug, the bug's priority plays a very important role in its resolution [37]. Whenever a bug is reported, it might be possible that many other functionalities of the system are dependent on that particular bug, so it needs to be resolved as soon as possible. Priority refers to how quickly a bug needs to be resolved and the order in which developers have to fix bugs [38, 39]. Correctly assigning bug priority is integral to successfully plan a software development life cycle. There are different levels of bug priority e.g. *Low priority*, *Medium priority*, and *High priority* depending upon its effect on the system. We are considering not prioritizing bugs as a process smell and calling it a *missing priority*. The potential *impact* of this process smell is on the development-oriented decisions (time and resource allocation) [40].

Potential Root Causes:

- *Triager Experience:* Priority assessment greatly depends on the triager experience and expertise as its high subjective thing. So this factor could cause this smell.
- *Time-Consuming Process:* For assigning a priority to any bug, triager needs to analyze the contents of the bug report fully which is a very time-consuming process. So it could be one of the possible root causes for our smell.

Smell Detection Method:

We check whether the priority field is valid or not. Some of the invalid priority strings are *None*, *Not Evaluated*, and *"–"*.

4.6 Not Referenced Duplicates

If the problem of duplicate bug recognition is solved, it enables the developers to fix bugs more efficiently rather than waste time resolving the same bug [41, 42]. However, it is observed that some bugs that are marked as duplicates in BTS are not referenced to the original bug within the *references* section of a bug report. Instead, the reporters only put the duplicate keyword into the *status* section, which reduces the traceability. Most of the bugs have their duplicate bug referenced correctly on the references section which increases the traceability of the bug. However, some of them do not have their duplicate bug IDs in the *references* section. As far as we have observed, most of these bugs still reference the duplicate bug in some way, such as referring to it in the *comment* section. But some of them are marked as duplicate and do not have any reference to the duplicate bug, and it is a deviation from ideal BT behavior. Therefore, we call it *not referenced duplicates* smell. The potential *impact* of this process smell is on the identification of master bug reports [43].

Potential Root Causes:

- *Search Feature of BTS:* It has been observed that the search feature in some of the BTS is of limited use and it does not allow the user to look for the master bug report.
- *New to Project/Unfamiliarity to Project:* It might be possible that the person marking the bug as a duplicate is new to the project and might have not enough knowledge of the ongoing project.

Smell Detection Method:

First, we check whether the bug is marked as a duplicate. If so, we check the linked issues field to find whether the duplicate bug is linked and has a reference to the other bug.

4.7 Missing Environment Information

During the BT process, it is necessary to know where the bug has been encountered [39] as bugs often only happen in certain environments [44]. It is critical to ensure that all the information related to the environment of that particular bug is listed (e.g. operating system (OS), the browser, the version of hardware and software, and a component of the system in which that bug is encountered [28]).

Every field has its importance, and if they are mentioned, they help the developer to resolve the bug quickly. A version of the product is an important field. If version information is missing in the bug report, the developer does not know in which version the user or tester is having this bug. So, missing version information in the bug report is considered to be a smell. To have complete information about a bug, the component information should also be mentioned in the bug report [3]; otherwise, it would be difficult to locate the bug within the product. Similarly, it is important to know on which OS the bug was encountered (e.g. Windows, Linux, macOS). Therefore, we are also considering the missing component and OS information a process smell. If all of this information is missing, then we call this smell *Missing Environment Information*. The potential *impact* of this process smell is on the bug replication. The bug with no environment information is difficult to reproduce [38].

Potential Root Causes:

- *Information is Unknown:* It could be possible that version/OS/component is unknown to the person reporting the bug so he/she did not mention it while reporting.
- *Forgot to Mention:* It could also be possible that the reporter forgot to

mention in which version/OS/component of the product the bug was found.

- *Shortage of Time:* It could also be possible that the reporter has a short time and in the hustle, he/she missed mentioning the information.

Smell Detection Method:

We check whether the *component*, *version*, *environment*, and *operating system* fields are empty. If all of these properties are empty, we consider it a smell.

4.8 Missing Severity

Upon identification of a bug, the bug report is submitted to a BTS. The bug report is triaged and the severity (e.g., low, medium, high) of the bug is assigned after a bug report has been submitted. The task of assigning a bug severity is a resource-intensive task [45]. Severity ratings help in determining the priority of a bug i.e. in which order the bugs should be fixed. A bug could be incorrectly prioritized if the severity of the bug is not mentioned, which in turn can affect the quality of the product that is being developed. It also helps into whom the bug should be assigned to fix it. [28, 39]. Therefore, it needs to be ensured the bug severity is correctly assigned. Several methods have been developed to automate the assignment of a bug severity [46]. In our case, if the severity information is missing, we consider it a process smell and call this smell *missing severity*. The potential *impact* of this process smell is; it affects the resource allocation and planning of other bug fixing activities [47].

Potential Root Causes:

- *Time Consuming:* Assigning bug severity is a time-consuming task, as triager has to analyze and understand the bug. Therefore, triagers can skip or miss this accidentally or unintentionally.

- *Triager Workload:* It could also be possible the cause of missing severity as

a wide variety of bug reports being submitted in OSS increases the triager's workload.

- *Criticality:* Realizing the severity of a bug is critical from a risk assessment and management point of view, so the reporter might get confused and not assign.

- *Lack of Domain Knowledge:* As severity, scores are assigned on human judgment so triager must have a piece of adequate domain knowledge, so that bug can be given a proper severity.

Smell Detection Method:

We check whether the severity field is valid or not. Some of the invalid severity strings are *N/A* and *"—"*. The detection mechanism may change across different BT tools. For example, Jira does not include a severity field by default but some organizations create their custom fields. For example, in the Atlassian organization, *Symptom Severity* and *Common Vulnerability Scoring System (CVSS) severity* fields are introduced. We treat them as an ordinary severity field while mining the bug tracking history.

4.9 Reassignment of Bug Assignee

Research shows that the time-to-correction for a bug is increased by the reassignments of developers to a bug [27]. Therefore, we consider the reassignment of developers to the same bug a process smell and call this smell *reassignment of the bug assignee*.

The potential *impact* of this process smell is an increase in the bug fixing time, which eventually delays the delivery of the product [4].

Potential Root Causes:

- *Bug Report Correction:* Some fields could be wrongly assigned when a bug report is submitted. Thus, these fields need to be reassigned.
- *Triager new to the system:* A bug report may be submitted where some of the fields are incorrectly allocated when a triager is new to the open-source project.
- *Admin Batch Operations:* In some organizations to better organize the project, administrators may also shuffle some fields in the bug reports.
- *Reassignment of fields:* The reassignment of some fields is caused by the reassigning of other fields.
- *Mistakenly Assigned to the Wrong Developer:* Bugs are sometimes reassigned to developers by mistake which is a common occurrence.

Smell Detection Method:

The mining strategy for this smell is to look in the bug history for the *assignee* property. If the *assignee* field is changed more than twice, then we count it as a smell. Also, we observed that there are some multiple assignee changes done in a very short interval. We consider these multiple assignee changes as a mistake and do not count them as a smell. The interval duration is set as five minutes. Therefore, if multiple assignments are done in five minutes, they are counted as one assignment.

4.10 No Comment Bugs

Generally, two of the important sources of information for developers during the software development life cycle are the bug reports and the comments that are associated with bug reports [1, 39]. When stakeholders or developers do not understand the bug then at that time comments play a vital part [48].

In BTS, comments can be posted by anyone in response to an initial bug report. Therefore, it means that for some notions of popularity, comment count can be used as a proxy. Textual contents of bug reports such as descriptions, summaries, and comments have been utilized by textual information analysis-based approaches to detect the bug duplicates [49]. In proposing the Bug Reopen predictor [50] features like description features, comment features, and meta-features are being used. For identifying the blocking bugs [51], the most important features are comment size, comment text, reporter's experience, and the count of developers. Comments on the bug report serve as a forum for discussing the feature design alternatives and implementation details. Generally, the developers who have interests in the project or who want to participate post the comments and indulge in a discussion on how to fix the bug. Considering the importance of comments in bug reports, we observed that some bug reports have no comments and consider it a process smell. We call this smell *no comment bugs*. The potential *impact* of this process smell is on the bug resolution time and other linked software development activities such as developer recommendation, duplicate bug detection, and bug reopening prediction [14, 52].

Potential Root Causes:

- *Developers/Contributors forgets to comment:* It could be possible that due to workload or any other reason the developer working on the project forgets to comment on bugs.

Smell Detection Method:

First, we check whether the bug is closed. If so, we check whether there is at least one comment in the bug.

4.11 Non-Assignee Resolver of Bug

In BTS, whenever a bug is encountered, it should be well documented and then resolved so that it can be traced later on if required. Therefore, it is important for traceability that bugs are assigned and resolved by the same person during their life cycle. To promote traceability, whoever is the assignee of a bug, that person should be the person to resolve the bug. Applying this will be helpful to have a "go-to person" if things fail. However, if this practice is not followed, it could be difficult to understand why some person has resolved a bug. If someone other than the assigned-to developer (but not the person who submitted it, i.e. reporter) resolves the bug report, then we consider it a process smell and call it a *non-assignee resolver of the bug*. The potential *impact* of this process smell is on the traceability of a bug [29].

Potential Root Causes:

- *Administrative Rules:* In some organizations, the privilege for closing and re-opening of a bug report is reserved for the administrative roles, so it can cause this smell.
- *Bug Reassignment:* It could be possible that the tossing of bug assignee results in this smell.
- *Assignee Forgot to Close Bug:* It could be possible that the assignee of the bug forgot to close the bug after resolving it. So, some other developers closed the bug upon seeing the resolved bug open.
- *Resolved By Expert:* It could also be possible that the bug could not be resolved by the assignee of the bug, later on, someone else who was an expert for those types of bugs resolved the bug and closed it.

Smell Detection Method:

First, we check whether the bug is assigned and resolved. If so, we compare the assignee and the person who resolved the bug.

4.12 Closed-Reopen Ping Pong

Reopened bugs are those that were previously closed by the developers but were later reopened for various reasons (such as not being reproduced by the developer or improperly tested fix). Reopened bugs reduce the overall software quality, increase maintenance expenses, as well as unnecessary rework by developers [53]. In a project, when a significant number of bugs are reopened frequently, then it can be an indication that the project is already in trouble, and maybe heading towards trouble soon [54]. We call this smell *closed-reopen bug ping pong*; as it is the ping pong among the bug states during its life cycle. Potential *impacts* of reopened bugs could be; they take a notably longer time to resolve [55]. Reopened bugs also increase development costs, affect the quality of product, prediction of release dates reduce the team morale leading to poor productivity [54].

Potential Root Causes:

- *No Review Bugs:* The bugs that are closed without reviewing have more chances of re-opening.
- *Not Tested Bugs:* The bugs that are not tested by QA (tester) could get reopened later on also the insufficient testing resources could also be a cause of this smell.
- *Ambiguous Task Specifications:* The bugs that have an ambiguous description and cannot be fully understandable by developers have a high chance of reopening.
- *Changed Scope:* It could also be possible that the developer fixed and closed the bug but later on it is realized that there should be an additional step(s), so the scope of the bug is changed somehow, and then it needs to be reopened.
- *Not Properly Fixed:* It could be possible that the bugs which are not properly fixed due to lack of attention of the assignee or any other factor might get reopened later on.

- *Incomplete Closed Bugs:* It could be possible that closed bugs with incomplete status might get reopened with the intention to fix that bug completely.

Smell Detection Method:

We check the history of the *status* field of the bug. Some projects explicitly use *REOPENED* value for the status field while others do not. In such cases, we check whether the status field is changed from *Closed* to another value, and we count it as a smell. Also, we observed that there are some multiple status changes in a very short interval. We consider these multiple changes as a mistake and do not count them as a smell. The interval duration is set as five minutes. Therefore, if a bug status is changed multiple times in five minutes, it is counted as one change.

Table 4.1: Taxonomy of Bug Tracking Process Smells

Smell Name	Definition	Potential Negative Impacts	Potential Root Causes
Unassigned Bugs	No assignee, but the bug is fixed and closed	- Loss of traceability - Delay in bug resolution	- Developer's availability - Time pressure on triager - Triager couldn't find an expert developer
No Link to Bug-Fixing Commit	If a bug is closed without any link to bug-fixing commit	- Loss of traceability - Affects related software Development tasks	- Developer forgets to mention - Committing link is not straightforward in BTS - Weak understanding of BTS
Ignored Bugs	Bugs that are left open for a long time or bugs that have incomplete resolution	- Delay in bug resolution	- Incorrect severity indication - Inadequate bug description - Incorrect prioritization - Overlooked bug
Bugs Assigned to a Team	The bugs which are assigned to a team but not a particular assignee	- Loss of traceability & accountability	- Mistakenly assigned by the triager - Unavailability of a developer
Missing Priority	The priority of the bug has not been set	- Affects software development oriented decisions (time & resource allocation)	- Expertise of triager - Wrong bug severity - Overlooked by triager
Not Referenced Duplicates	Duplicate bugs that do not have a reference to their original bug	- Identification of master bug report becomes difficult	- Person marking duplicate is new to the system - Unaware of the previous bugs - Poor search feature of BTS to find duplicates
Missing Environment Information	Environment information (Version, OS, and Component of the product) about the bug is missing	- Bug reproduction becomes harder	- Reporter forgets to mention - Reporter do not know the environment details - Time constraints of reporter
Missing Severity	The severity of the bug has not been set	- Affects resource allocation & planning of other bug fixing activities	- Triager overlooks
Reassignment of Bug Assignee	Fixer for the bug is assigned more than once	- Increase bug fixing time - Delay software product delivery	- Reassignment of some fields cause others to be reassigned - Triager don't know the suitable developer - Developer recommendation system is not integrated into BTS - Admin batch operations
No Comment Bugs	A resolved bug with no comments	- Increased bug resolution time - Affects software development activities (e.g. developer recommendation, duplicate bug detection and bug reopen prediction)	- Ignored bug - Developers/Contributors forget to write comments - Time constraints of developer
Non-Assignee Resolver of Bug	When a bug is resolved by any person other than assignee(s)	- Loss of traceability	- Assignee forgot to close - Bug was originally resolved by someone else - Bug can be closed by administrative roles - Assignee might not be able to resolve and toss it to other developers
Closed-Reopen Ping Pong	Bugs which are reopened many times	- Increases development cost - Affects software's quality - Less accurate prediction of release date - Poor team productivity	- Insufficient unit testing - Ambiguous bug specifications - Changed bug scope - Poorly/incorrectly fixed bugs - Tester not testing properly

Chapter 5

Empirical Analysis

RQ2 investigates whether the BT process smells we proposed in our taxonomy exist in real-life BT data and to what extent. To answer this research question, each BT process smell is empirically evaluated by mining the BT histories of all six projects. The details about the dataset, data cleaning procedure, and the details of the mining process are given in the following subsections.

5.1 Dataset

We analyzed six large-scale software projects that have a publicly available issue tracking system. The projects are GCC[1], Wireshark[2], Confluence Server and Data Center[3], Jira Server and Data Center[4], MongoDB Core Server[5], and Evergreen[6].

We utilized the Perceval tool [56] to fetch the issue tracking histories of those

[1] https://gcc.gnu.org/bugzilla/
[2] https://bugs.wireshark.org/bugzilla/
[3] https://jira.atlassian.com/projects/CONFSERVER
[4] https://jira.atlassian.com/projects/JRASERVER
[5] https://jira.mongodb.org/projects/SERVER
[6] https://jira.mongodb.org/projects/EVG

Table 5.1: The details of projects

Project	Starting Year	Ending Year	Number of Issues	Number of Bugs
GCC	1999	2020	76910	69784
Wireshark	2005	2020	16609	13420
Jira Server and Data Center	2002	2020	46097	21609
Confluence Server and Data Center	2003	2020	42934	25841
MongoDB Core Server	2009	2020	50147	22593
Evergreen	2013	2020	10545	2688

projects. When fetching the histories, no time limit is set. Therefore, the dataset includes all the issues of projects up to December 2020 except Wireshark. The issue tracking system of Wireshark was migrated from Bugzilla to GitLab in August 2020. Thus, the latest issue of the Wireshark project is updated in August 2020. The starting dates are given in Table 5.1. Also, we share the project history datasets in our replication package[7].

5.2 Data Cleaning and Preprocessing

After downloading the histories of the issue tracking systems, we extracted the bug reports. An issue may fall into one of the different categories such as feature requests, bug reports, and enhancements. We only processed bug reports and removed other types of issues from our dataset.

The default issue type in Bugzilla projects is a bug. In Jira, we only included the issues whose type is bug or defect. The total number of bugs and issues are shown in Table 5.1.

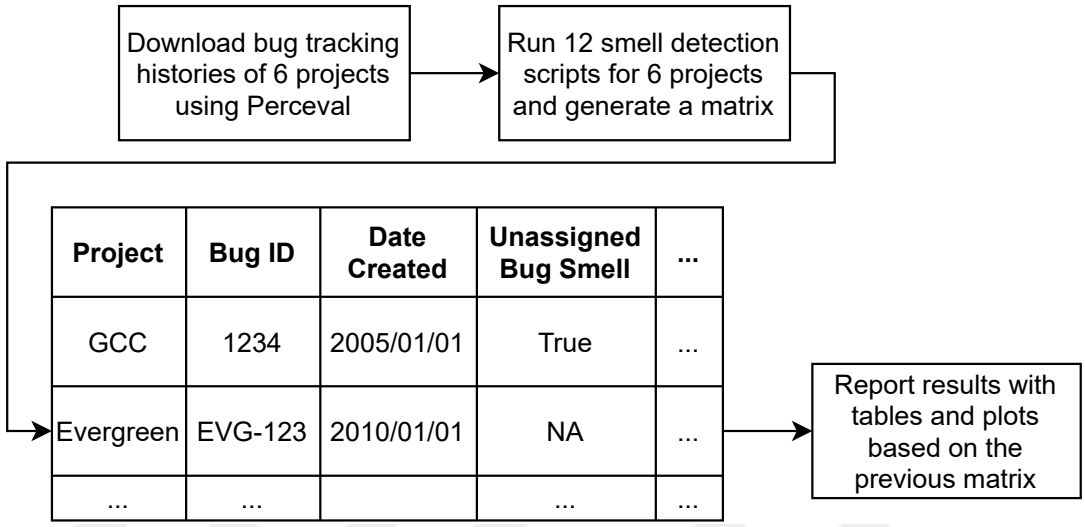

Figure 5.1: Empirical Evaluation Steps

Table 5.2: Number of Bug Reports that have the corresponding smells for each project (B: Bugzilla, J: Jira)

	Confluence (J)			**Evergreen** (J)			**GCC** (B)			**Jira** (J)			**MongoDB Server** (J)			**Wireshark** (B)		
	False	True	NA	False	True	NA	False	True	NA	False	True	NA	False	True	NA	False	True	NA
Assigned to a Team	10917	0	14924	2006	637	45	23325	0	46459	6962	0	14647	17953	2188	2452	219	0	13201
Ignored	741	25100	0	823	1864	1	42283	27501	0	588	21021	0	10487	12106	0	9967	3453	0
Missing Environment	24010	1831	0	2256	432	0	69784	0	0	20415	1194	0	20270	2323	0	13420	0	0
Missing Priority	25841	0	0	2688	0	0	69784	0	0	21609	0	0	22591	2	0	13420	0	0
Missing Severity	8494	17347	0	0	2688	0	69784	0	0	8066	13543	0	0	22593	0	13420	0	0
No Comment	19854	2951	3036	1621	306	761	57660	8	12116	13299	4492	3818	20612	1135	846	12510	27	883
No Link to Commit	0	0	25841	1433	331	924	3997	34223	31564	0	0	21609	9495	3589	9509	2677	5285	5458
Not Referenced Duplicate	2302	456	23083	182	51	2455	8277	0	61507	2124	230	19255	2499	434	19660	2972	0	10448
Reassignment	24772	1069	0	2487	201	0	69196	588	0	19952	1657	0	19801	2792	0	13416	4	0
Reopen Ping Pong	24661	1180	0	2562	126	0	66750	3034	0	20855	754	0	21630	963	0	12775	645	0
Resolver is not Assignee	8019	2821	15001	1867	508	313	18662	3342	47780	5215	1172	15222	16195	3139	3259	88	123	13209
Unassigned	6225	2665	16951	1316	6	1366	21110	16696	31978	5759	1975	13875	12733	351	9509	188	7656	5576

5.3 Mining Bug Tracking Process Smells

Following the retrieval of project histories and filtering irrelevant issue types, we run the smell detection scripts which generate a matrix of bugs and process smells. In the matrix, each row corresponds to a bug, and the columns represent the process smells in addition to the bug metadata such as unique bug identifier and date created. The flow of the evaluation is visualized in Figure 5.1.

Based on the generated matrix, we analyze the density of the smells for each project. Table 5.2 displays the total number of bug reports that have the appropriate process smells for each project. The *True* column expresses how many of the eligible bugs have the corresponding process smell, the *False* column expresses how many of the eligible bugs do not have the corresponding process smell and the *NA (not applicable)* column expresses how many bugs are not eligible for that process smell.

For example, in the Evergreen project, 637 bug reports are assigned to a team, 2006 bug reports are not assigned to a team. Also, 45 bug reports are marked as *NA* for that smell. A bug report is marked as *NA* if it does not hold the precondition of the corresponding process smell. For example, to be considered an *Assigned to a Team* smell, a bug report must be assigned. That is, 45 bug reports are neither assigned to a team nor a person. All preconditions are given in the smell detection methods of each process smell in Chapter 4.

A visual demonstration of Table 5.2 is given in Figure 5.2. It shows the ratio of process smells for each project, the smell ratio is normalized according to the number of bugs. For example, in the Evergreen project, 637 bugs are assigned to a team and the total number of bugs that are eligible for this smell is $583+2060 = 2643$. Thus, the ratio is $637/2643 = 0.24$ which is shown in lighter blue in the figure.

Since Atlassian projects (Jira and Confluence) are closed source projects, we could not find any link from their issue tracking systems to a version control

[7] `https://figshare.com/s/3f402ee60f46ea1fe99c`

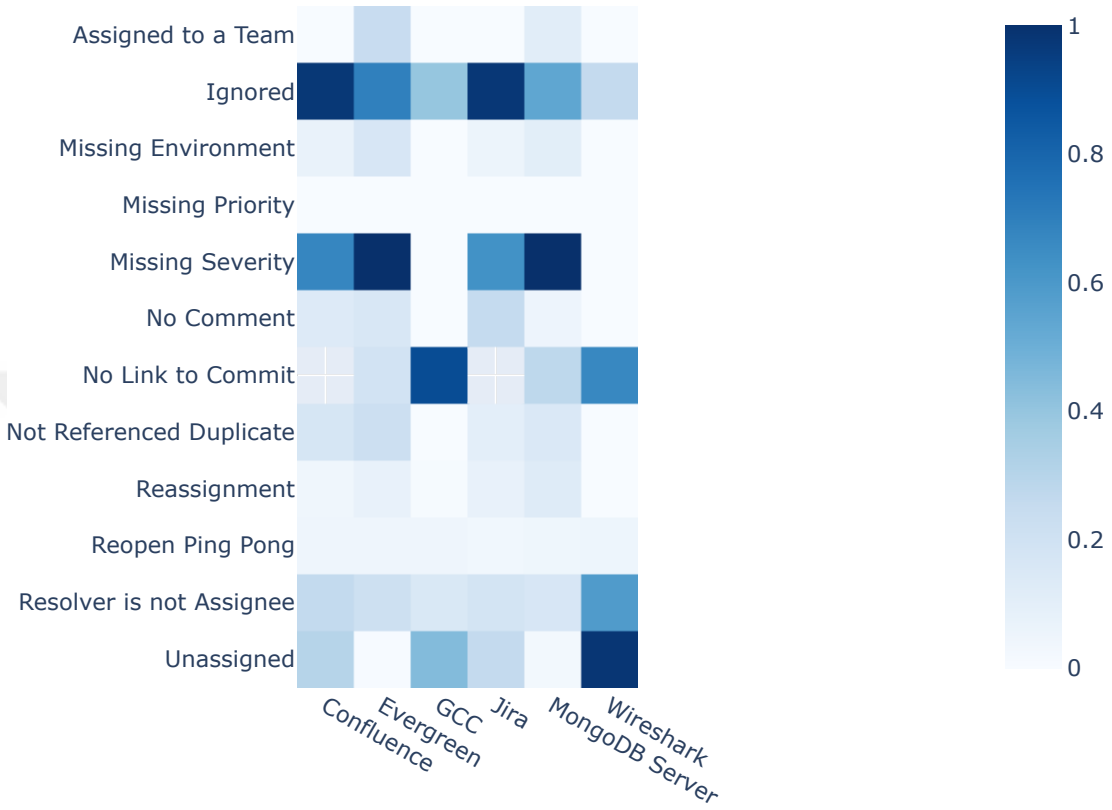

Figure 5.2: Heatmap of Smell Ratio

system. Therefore, we mark 'No Link to Commit' as not applicable to those projects and the corresponding cells in Figure 5.2 are left blank. We also evaluated how the number of BT smells changes over time. Figure 5.3 shows the change of the smell ratio for each project over the years. It can be observed that the number of process smells tends to decrease over time for almost all the projects.

To understand the possible reasons for the decrease, we investigated the change of all BT process smells in a project. For instance, *No Link to Bug-Fixing Commit* smell has a sudden drop in the Wireshark project starting from 2013, which is shown in Figure 5.4. While searching the mail lists, we found out that Wireshark developers adapted Git and Gerrit tools in December 2013[8]. Links between commits and bugs are connected after the adaptation of those tools.

[8]`https://www.wireshark.org/lists/wireshark-dev/201312/msg00217.html`

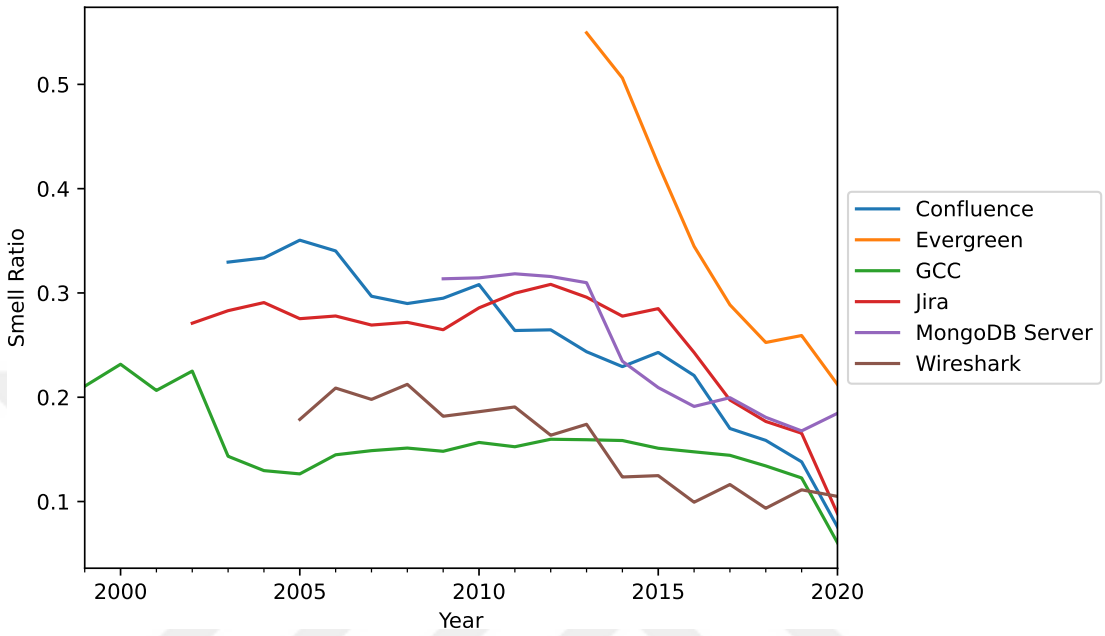

Figure 5.3: Smell Ratio Change of All Projects by Year

Another decrease in the number of BT process smells can be observed in Confluence Server & Data Center project. Figure 5.5 shows how the number of *Missing Severity* smells decreases over time. After Atlassian added a new severity field to their projects in August 2016[9], the ratio of this process smell declined rapidly. However, the decline does not begin in 2016. The reason behind the drop before 2016 probably depends on the lifetime of the bugs because we count a bug as belonging to the year it was created. In other words, even though a bug is created before 2016, its severity field is updated after 2016 and that is why we observe the decline.

For the same project, no smell data is available for *Resolver is not Assignee* smell after 2018. To understand the reason behind that, we analyzed the bug transition history of this project before and after 2018. Our analysis shows the project stopped using the *Resolved* state and bugs are no longer transitioned to the *Resolved* state.

[9]`https://www.atlassian.com/blog/announcements/realigning-priority-categorization-public-bug-repository`

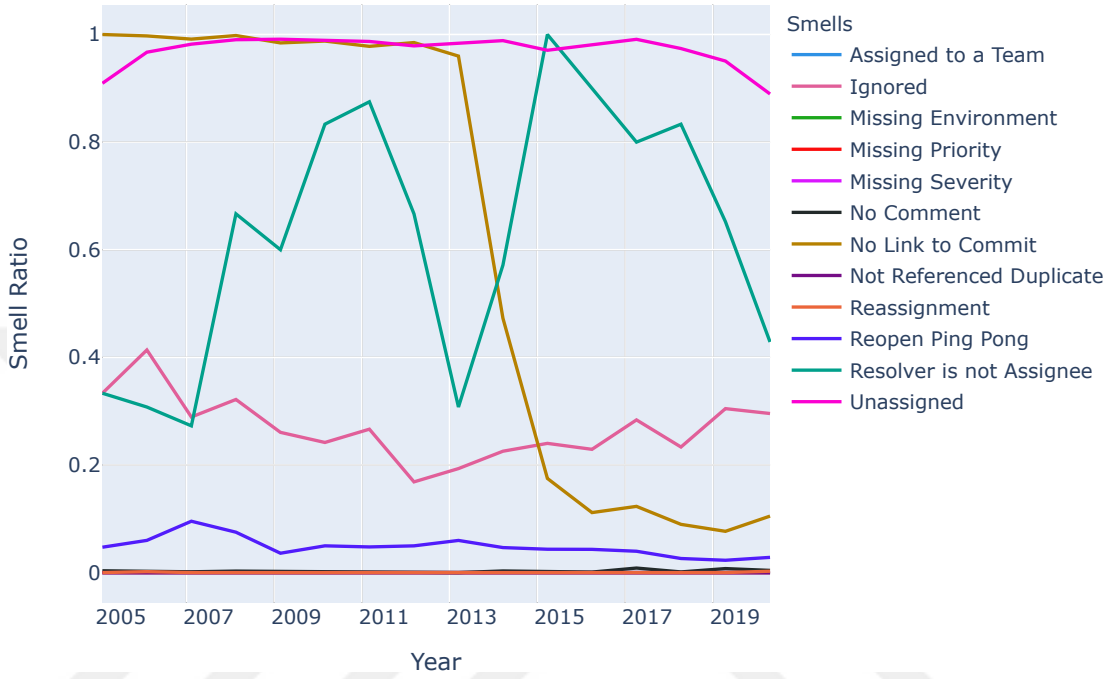

Figure 5.4: Smell Ratio Change of Wireshark Project

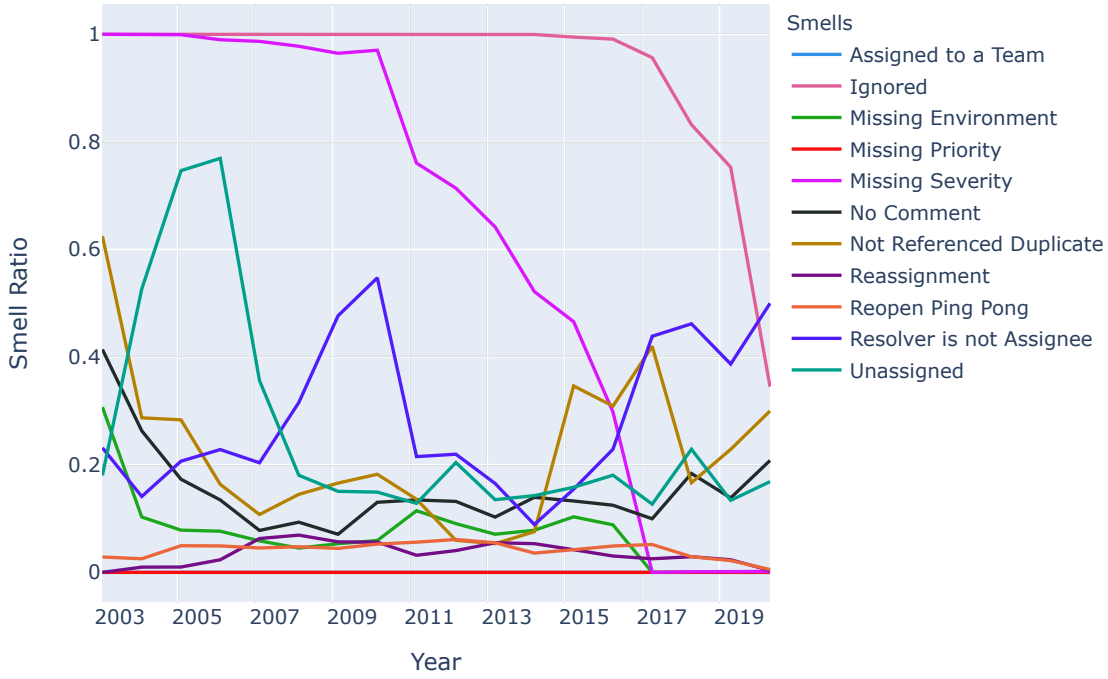

Figure 5.5: Smell Ratio Change of Confluence Server & Data Center Project

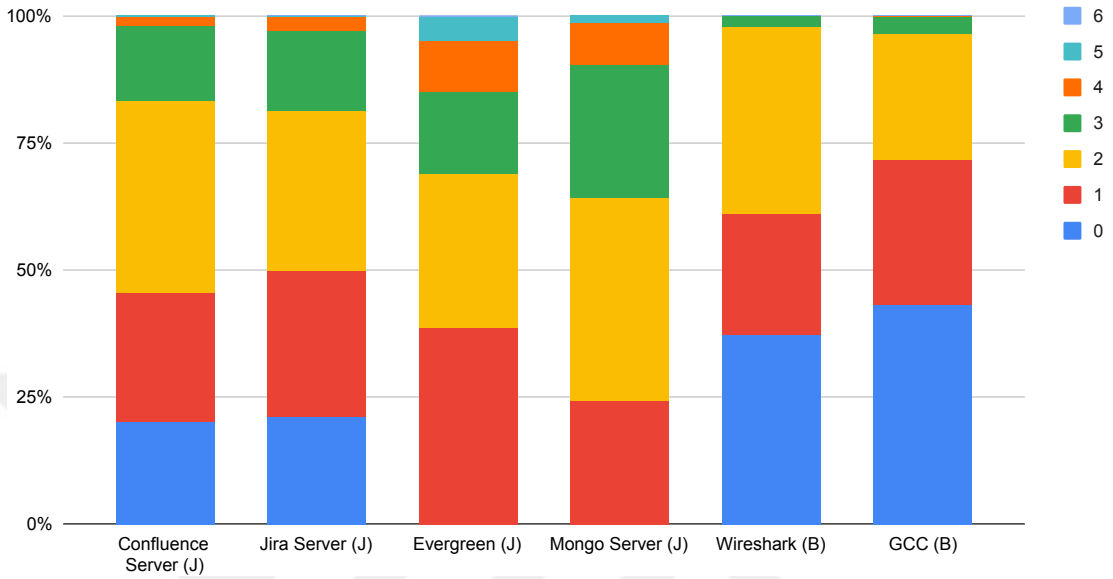

Figure 5.6: Percentage of Bug Affected by Different No. of Smells

5.4 Comparison of Bug Tracking Tools

The results of the empirical analysis are given in Table 5.2 which implies that each BT process smell occurs with varying ratios in all six projects. For two Jira projects (*Confluence Server & Data Center and Jira Server & Data Center*), *No link to Bug-Fixing Commit* gives no results. Due to the privacy policy of these projects, we could not access the commit links. In two of the Jira projects (*Evergreen & MongoDB Core Server*), there is no severity field. However, in the other two Jira projects (*Confluence Server & Data Center and Jira Server & Data Center*) severity is defined in terms of custom-field as 'Symptom Severity'. On the other hand, since severity is a mandatory field that needs to be set while submitting a bug report in Bugzilla's projects, the smell ratio is 0%. We observed that the ratio of *closed-reopen ping pong* is comparatively less than other smell categories. This potentially shows that reopening the bug is avoided during the BT process as it increases maintenance costs [55].

Figure 5.6 shows the percentage of bugs affected by the multiple smells. The numeric values from 0 to 6 indicate the number of smells. The bugs in *Evergreen* and *MongoDB Core Server* projects have at least one smell. Also, no bug has

more than six smells in any project. Figure 5.6 implies that the majority of bugs have one or two smells.

5.5 Time-Based Analysis

We also demonstrated how the number of BT process smells changes over time. Figure 5.3 shows the change of the process smell ratio for each project over the years. If we look at these concerning the BT tools, then we can say that over time BT tools evolved and get more advanced. As it is already mentioned, for obtaining the severity of Jira projects, the custom user-defined field *Symptom Severity* was analyzed. Some Jira Projects like *Mongo DB Core Server* & *Evergreen* lacks the severity field as shown in Table 5.2. Consequently, it is observed that over the years, best practices for the BT process have been adopted. Figure 5.3 shows that some projects were slow in adopting best practices like *GCC* & *MongoDB Core Server*. The graph shows that practices being followed within the organizations related to these projects evolved slowly over time. Whereas, in other projects, *Confluence Server & Data Center*, *Jira Server & Data Center*, *Evergreen*, and *Wireshark* the best practices were adopted quickly to avoid bad practices during the BT process over time.

Chapter 6

Synthesizing Practitioner's Survey

To know the perception of software practitioners about our proposed taxonomy of bug tracking process smells. For this purpose, we conducted a survey with 30 software practitioners. In our survey, each process smell is defined in detail in the survey, along with a real-life example. The respondents are then asked a series of questions related to each process smell.

The majority of respondents agree with our list of BT process smells, according to the survey results (Table 6.1). We questioned our survey participants on how often they encountered this smell during the BT process. To get an expert opinion about the potential root causes of all the process smells, we have enlisted the potential root causes and asked practitioners to choose among them as per experience and if we have missed any factor, they can tell us in open-ended questions. Similarly, for the impacts of BT process smells, we questioned our participants by enlisting potential impacts and asked them to write any other impact they could think of in open-ended questions. Moreover, we asked the practitioners if they were aware of these process smells before (Table 6.1) and if there are any actions taken in their companies to prevent the occurrence of these process smells. Furthermore, we asked them the thresholds for some of our process

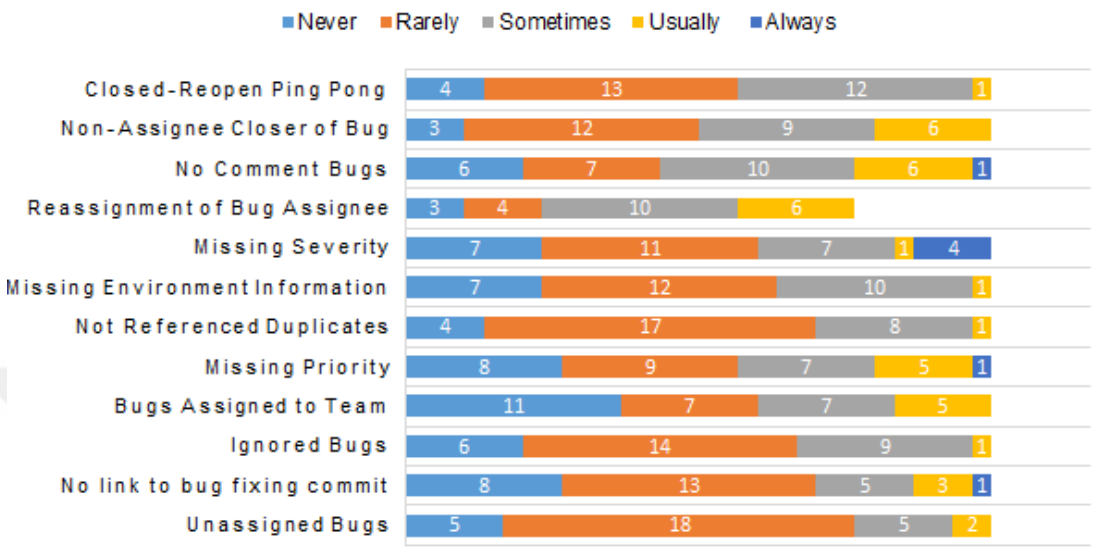

Figure 6.1: Distribution of answer to the question: *How often do you encounter a process smell in your company?*

smells, we asked software practitioners. We adjusted our thresholds according to the practitioner's feedback. The distributions of responses to the Likert Scale questions were also used to demonstrate the survey members' differing viewpoints which are given in Figure 6.1 and 6.2.

This chapter uses quotes from open-ended survey questions to express practitioners' perceptions of all the process smells. Also, based on the survey, we have mentioned the actions taken by different organizations in order to avoid the occurrence of process smells. We explain the perception of practitioners for each process smell in the following sections. Each section has their reason of disagreement with the process smell, how often they encounter the smell during BT process, actions are taken in companies to avoid the smell, potential root causes (the number in bracket shows the number of responses), and additional comments from the respondents.

6.1 Unassigned Bugs

Out of 30, 29 respondents agree with this process smell which implies this process smells practically exists within software development companies. One practitioner who does not agree to this process smell states his reason as: *"I only came across unassigned bugs when the bug report was a duplicate/not needed or the bug was not reproducible (bc. it was fixed by another). It's possible that such bugs were resolved during triage (for example by the PM) so I don't think they are indicators of wrong practice."*

Afterward, we asked practitioners about how often they encountered unassigned bugs during BT process, most of the practitioners said 'Rarely' (Figure 6.1). To be aware of the actions taken against this process smell in different companies, we asked our respondents if they are aware of any actions taken in order to avoid this process smell, 18/30 respondents have said yes(Figure 6.2). Some of those actions are mentioned below;

"Defined SOPs/steps to reports bugs"

"This doesn't happen in our company. The assignee is a mandatory field to change the status of the bug to assign."

"Unassigned bugs cannot progress to certain life-cycle stages"

"Bugs are monitored by team leads and are assigned to developers on a daily basis. A red flag is raised by the manager if unassigned bugs are in a bulk"

"Bug cannot be opened without an assignee. At first. it should be assigned to the technical lead or product architect"

We asked practitioners their expert opinion about the potential root causes of this smell and some of the potential root causes are as follows;

- Availability of developer (11/30)

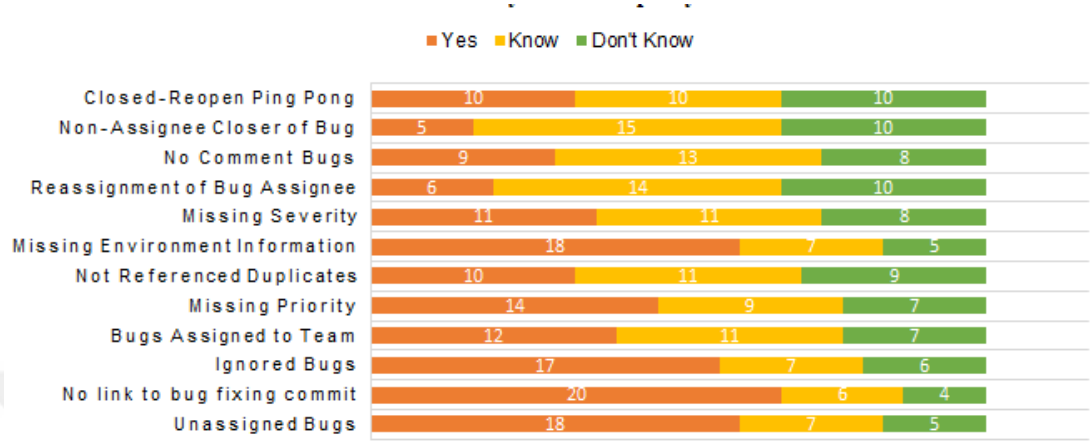

Figure 6.2: Distribution of answer to the question: *Are there any actions taken to prevent smell in your Company?*

- Time pressure on the triager (14/30)
- Triager couldn't find the expert developer for the given bug (12/30)

In open-ended questions, some potential root causes given by practitioners are: Multiple developers work on development and wrong risk management/planning or insufficient process rules.

6.2 No Link to Bug-Fixing Commit

28 out of 30 respondents agree with this process smell. One of the reasons for not agreeing with the process smell is *"Usually developers put version numbers (which include the build number from the CI platform) in the issue details which is used as a reference"*

The majority of respondents said they encountered this process smell 'Rarely' during BT process (Figure 6.1). 20/30 respondents said there are actions taken in their companies to avoid this process smell (Figure 6.2). Some of those actions are mentioned below;

"Configuration management department set hooks, in this way, if the developer does not indicate issue number such as Jira#Project_Abbv-Issue_Number in the commit message, the system does not allow to push his code to the remote repository."

"Our tracking system automatically fetches the related commit since we use the ticket ids in our commit descriptions."

"We have imposed that bugs cannot be fixed without a proper commit branch, message and pull request"

We asked practitioners their expert opinion about the potential root causes of this smell and some of the potential root causes are as follows;

- Developer forgets to mention the commit link while fixing a bug (22/30)
- Committing the link is not straightforward in the BT system (11/30)
- Weak understanding of the BT system (10/30)

Some of the potential root causes suggested by respondents in open-ended questions are; bug tooling not enforcing it and multiple issues are resolved at same the time and not all of them are code bugs.

6.3 Ignored Bugs

29 out of 30 respondents agree with this process smell. The ones who do not agree with the process smell stated *"Bugs might be reported that have no real impact on the functionality of the software. Such bug reports should be pruned once in a while."*

For ignored bugs, we also asked the practitioners about the threshold value we are using in our empirical analysis to declare a bug as an ignored bug. Given

the multiple options (3 months, 6 months, 1 year, other) 17 respondents voted for 3 months, while some of the respondents gave their suggestions for 2 weeks. We have changed our analysis as per the survey response and it is 3 months now. Moreover, in an open-ended question respondents mentioned that the time limit for ignored bugs depends on;

"Depends on the content of the bug, the effect of it on the code, the project phase/urgency, the impact of the bug"

The majority of the respondents selected 'Rarely' when we asked how frequently they encountered ignored bugs(Figure 6.1). Regarding actions taken in companies for avoiding ignored bugs, 17/30 respondents responded positively (Figure 6.2) and some of those actions are as follows:

"All bugs are regularly reviewed by product managers. (Though they are not that good at pruning the bug list)."

"We deploy configuration control boards (CCB) which go through bugs and other issues for resolutions, possible state changes, and reassignments when needed."

We asked practitioners their expert opinion about the potential root causes of this smell and some of the potential root causes are as follows;

- Incorrect severity indication (11/30)
- Vague or inadequate bug description (17/30)
- Incorrect prioritization(17/30)
- Overlooked the bug(12/30)

One other potential root cause mentioned by one of the respondents is: maybe the reported bug is not really a bug (faulty triaging).

6.4 Bugs Assigned to Team

25 out of 30 respondents agree with our smell definition. Some of the reasons for those who do not agree with this process smell are as follows *"I believe it is not a bug smell because those kinds of bugs are generally assigned to that particular team by other teams. It's team responsibility to assign to a specific member and it might take some time to do so."*

"A bug detected in a component can be assigned to the relevant team due to collective code ownership. The team itself can progress this bug in the sprint plan. In the sprint plan, a volunteer can take ownership of this bug."

"I believe that a bug might be assigned to the teams (in certain circumstances). it might not be a very straightforward task or even people from different areas in the same team (f/e, b/e, etc.) might have to be involved in the bug solving process. I don't see this scenario as a smell. If you consider the accountability, you still assign the ticket to some developer(s). Actually, even better, you account for more than one developer, the whole team, which is a safer method. Of course, not all bugs should be assigned to the teams. But I think there might be some bugs that can be solved by multiple devs."

The majority of respondents (11/30) said they 'Never' encountered this smell during the BT process (Figure 6.1). We also asked practitioners if it is true, according to their perception, that bugs that are assigned to a team are not fixed or receive any comments in a short period resulting in a long time to fix. 10/30 respondents agree to it and 10/30 respondents said 'maybe'. Regarding actions taken in different companies for avoiding ignored bugs, 12/30 said yes (Figure 6.2) and the actions taken are as follow

"Having weekly ticket refining meetings where tickets are assigned to the team members before the start of the sprint planning."

"We can't assign bugs to a team. They need to be assigned to a single person. Usually that's the project lead who can then propagate it further to their team

members"

We asked practitioners their expert opinion about the potential root causes of this smell and some of the potential root causes are as follows;

- Mistakenly assigned by a triager (11/30)
- Unavailability of developer (18/30)
- Expertise of triager while assigning bug reports (system knowledge) (12/30)

6.5 Missing Priority

25 out of 30 practitioners believe that missing priority corresponds to bad practice in the bug tracking process. One of the reasons for not considering it a smell is, *"There are times when all of the bugs are declared high priority. And the developer has to figure out on its own which ones should be fixed on priority."*

The majority of respondents said they have 'Rarely' encountered this process smell, shown in Figure 6.1. afterward, we also asked practitioners if there are any actions taken in their companies to avoid this process smell, 14/30 respondents have said yes (Figure 6.2). Some of those actions are mentioned below;

"We have made it mandatory to assign priority tag while posting a bug"

"PMs always assign priorities if missed by the QA engineer"

"CCB: Group of leads comes together to prioritize the bugs"

We asked practitioners their expert opinion about the potential root causes of this smell and some of the potential root causes are as follows;

- Expertise or experience of triager (23/30)

- Wrong bug severity (10/30)
- Overlooked by a triager (15/30)

6.6 Not Referenced Duplicates

28 out of 30 respondents believe that this corresponds to bad practices in the bug tracking process.

The majority of respondents have said they have 'Rarely' encountered this process smell (Figure 6.1). 10/30 respondents have told us the actions their companies are taking to avoid this process smell (Figure 6.2), some of them are as follows

"Review of bug reports."

"Use of similar issues finding capability of BTS, mandatory selection of bugs in case of 'is duplicate' link selected."

"Grouping the different types of bugs in BTS. So that you would need to search fewer of the previous bugs to find the duplicate if it exists."

We asked practitioners their expert opinion about the potential root causes of this smell and some of the potential root causes are as follows;

- Person marking duplicate is new to the system (17/30)
- Person marking duplicate is unaware of previous bugs (24/30)
- Poor search feature of bug tracking system to find master bug report (11/30)

Another important potential cause stated by one of the respondents is "One bug can cause multiple features to crash. Then each feature failure can become one separate bug, hence resulting in a duplicate bug."

6.7 Missing Environment Information

All 30 respondents believe that missing environment information in bug reports corresponds to bad practice in the bug tracking process. The majority of respondents (Figure 6.1) said they have 'Rarely' encountered this smell during the BT process. afterward, we also asked our respondents if there are any actions taken in their companies to avoid this process smell, 18/30 respondents have said yes (Figure 6.2). Some of those actions are mentioned below;

"We try to mention 'how to reproduce' the bug with the description"

"Bug reports are being reviewed" "It is mandatory to add environment info while posting bugs"

"The test tool can automatically collect environment information"

We asked practitioners their expert opinion about the potential root causes of this smell and some of the potential root causes are as follows;

- Reporter forgets to mention (21/30)
- Reporter doesn't know the environment details(20/30)
- Reporter was short on time to mention this information (18/30)

6.8 Missing Severity

28 out of 30 respondents believe that missing environment information in bug reports corresponds to bad practices in the bug tracking process. Other respondents have mentioned their reasons for not considering it a process smell, which are as follows *"People give high priority when severity is high. Therefore, they end up using only one."*

"We only have priority fields in our issues (using Jira, I guess it is modifiable), and I have never realized that it is missing until now."

Most of the respondents said they have 'Rarely' encountered this smell (Figure 6.1). Regarding actions taken in organizations for avoiding missing severity bugs, 11/30 said yes (Figure 6.2) and some of those actions are

"Bug reports are reviewed and re-prioritized regularly."

"A group of experienced staff comes together to fix the severity of the open bugs."

"The severity field is mandatory while creating a bug"

We asked practitioners their expert opinion about the potential root causes of this smell and some of the potential root causes are as follows;

- A large number of bug reports submitted everyday increase triagers' workload (20/30)
- Triager overlooks (16/30)

6.9 Reassignment of Bug Assignee

24 out of 30 respondents believe that reassignment of the bug assignee corresponds to bad practices in the bug tracking process. A few of the reasons for not considering it a smell as stated by a respondent is *"This becomes a smell if a bug is reassigned more than three times. Before that, some organizations have a trend in which all bugs are assigned to the lead first who then reassigns them to the appropriate developer"*

"Bug reassignment could be a legit action taken in case of unavailability of developer or complex nature of the bug."

"My organization actually uses reassignment as a marker of progress (reporter assigns to triager, triager assigns to dev, dev assigns to reporter/tester after fixing it, and so on). Not sure if this is a good way to go about it but so far it seems to work"

"I think there are valid reasons for a bug report to be reassigned, maybe the initial dev is not experienced enough to solve the issue, or they had to be assigned to another report or project and therefore can not continue working on this one or the bug is just difficult to solve"

In our survey, we also asked about the threshold for counting this process smell i.e. how many times the reassignment of bug assignee should be considered as a smell. Previously we were considering the single reassignment as a process smell, but it was one of the potential threats to the validity of our analysis. Therefore, we decided to take an expert opinion regarding the threshold value. As per survey results, 13 respondents said the threshold should be twice and 12 respondents said it should be thrice. However, respondents also mentioned that the threshold depends on factors like the size of the team and how the bug tracking life cycle is laid out in any organization. As per the majority results, we are selecting 'Twice' as a threshold for this process smell. Most of the respondents said they have encountered this process smell 'Rarely' (Figure 6.1). Regarding actions taken in companies for avoiding reassignment of bugs, 6/30 said yes (Figure 6.2), and some of those actions are

"Bugs are reassigned only under very special circumstances"

"Discussion in the team of that particular project and assign the bug to respective developer"

We asked practitioners their expert opinion about the potential root causes of this smell and some of the potential root causes are as follows;

- Reassignment of some fields cause other fields to be reassigned (9/30)
- Triager might not be knowing the suitable developer for the bug(22/30)

- No developer recommendation system integrated into Bug tracking system (15/30)
- New bug report correction (When a bug report is submitted, some fields could be wrongly assigned) (11/30)
- Admin batch operations (Administrators also reassign some fields in the bug reports to better organize the project) (7/30)

One of the respondents also mentioned another potential root cause which is the availability reasons of the developer (the initially assigned developer had to switch to another task / took a vacation).

6.10 No Comment Bugs

24 out of 30 respondents believe that the bug reports with no comments correspond to the bad practice during the bug tracking process. Some of the reasons for not agreeing with our smell definition are as follows *"Some bugs are trivial and require no comments."*

"I have rarely encountered a bug report which does not have any descriptions. Also, I feel like verbose descriptions having extra/unnecessary details are almost always better than having shorter descriptions which are missing crucial details"

"I think some of that information can be found on the description field - in my experience, the lack of this field causes more issue than not having comments on the report"

As you can see from Figure 6.1, the majority of respondents say they encountered this process smell 'Sometimes'. Then we also asked our respondents if there are any actions taken in their organizations to avoid this process smell, 9/30 respondents have said yes (Figure 6.2). Some of those actions are mentioned below;

"Developers and QA engineers have regular phone calls"

"Comments are mandatory for a developer while marking the bug as fixed so that the actual fix can be traced in future and its area of impact be defined"

"We have a reporting mechanism to detect these situations"

"Team members are constantly reminded that comments are to be added by day end."

We asked practitioners their expert opinion about the potential root causes of this smell and some of the potential root causes are as follows;

- Due to incorrect severity and priority, the bug is being ignored (14/30)
- Developers/Contributors forgets to comment (17/30)
- A vast number of bug reports often contain excessive description and comments, which may become a burden to the developers (13/30)

6.11 Non-Assignee Closer of Bug

19 out of 30 respondents believe that the described scenario about the non-assignee resolver of a bug corresponds to bad practice in the bug tracking process. The reasons provided by respondents for not agreeing with the smell definition are *"Some teams are so well integrated that they can easily take over bugs from other members of the team"*

"Some companies work that way, sometimes due to lots of new hires"

"It's OK if someone who closed the bug is the team lead/mentor/manager of the developers. Sometimes, in the meeting, the team lead shares the screen and does these actions in front of the whole team. Also, the bug is assigned to the developer for resolution by the triager. If the developer is not able to solve it,

s/he may reassign or toss it to other developers: in this case, s/he would reassign to someone else instead of closing it"

"Bugs can be resolved by the QA engineer or the developer. I don't think this is an indicator of bad practice in a company but could be considered as such in open source projects."

Figure 6.1 shows that the majority of respondents have said they have encountered no comment bugs 'Sometimes' during BT process. Afterward, we also asked our respondents if there are any actions taken in their companies to avoid this process smell, 5/30 respondents have said yes (Figure 6.2). Some of those actions are mentioned below;

"PMs make sure the assigned developers are closing bugs"

"Developers are encouraged to mark the bug as fixed themselves"

"It is a good scenario to assign it to the same person, but in real life, it is necessary to proceed with redundancy."

We asked practitioners their expert opinion about the potential root causes of this smell and some of the potential root causes are as follows;

- Assignee forgot to close the bug (20/30)
- Bug was originally resolved by someone else (16/30)
- In most big projects, a bug can be closed or re-open later, is reserved for higher-level or administrative roles that only the core developers have (11/30)
- The bug is assigned to the developer for resolution by the triager. If the developer is not able to solve it, he may reassign or toss it to other developers. (14/30)
- In quest of finding the right expertise of developer (5/30)

6.12 Closed Reopen Ping-Pong

22 out of 30 respondents believe that the described scenario about closed-reopen ping pong of a bug corresponds to bad practice in the bug tracking process. And the reason for not agreeing to our this process smell is stated by respondents as

"I agree that this is a smell but I don't think this is completely related to BTS. For example, 'Insufficient unit testing' is not even in the scope of BTS. In case the bug occurs again, you would reopen the old one or open a new bug(duplicate). Either way, you say that it is a smell. What should developers do in this case? This happens in real life. Then, re-occurring bugs are always a smell"

In the survey, we also asked about the threshold for considering it a smell i.e. do the respondents agree with our assumption that if the bug is reopened at least once it is a smell, or should we change this threshold. The majority (17/30) of the respondents said once is fine while others say once is a very strict threshold, maybe twice is better. Some of the other concerns of respondents were *"It depends on the severity of the bug"*

"1 is good and maybe a frequency can be added as a parameter as well. re-opened bugs in 3 months for example. Some bugs are difficult to generate by the Devs/QA."

The majority of respondents said they 'Rarely' encountered this smell during the BT process (Figure 6.1). Regarding actions taken in organizations for avoiding reassignment of bugs, 10/30 said yes (Figure 6.2) and some of those actions are

"If a bug is reopened, the developer and tester shall come together to assess the bug"

"Mandatory unit tests that run automatically before each commit, the practice of implementing and updating unit tests, bug reports containing detailed descriptions, communication between staff."

Table 6.1: Survey Results

Smell Name	**Do you agree with the definition of BT process smell?**	**Which of these smells you were aware before our survey?**
Unassigned Bugs	96.7%	80%
No Link to Bug-Fixing Commit	93.3%	66.7%
Ignored Bugs	96.7%	86.7%
Bugs Assigned to a Team	83.3%	60%
Missing Priority	83.3%	76.7%
Not Referenced Duplicates	93.3%	73.3%
Missing Environment Information	100%	76.7%
Missing Severity	93.3%	63.3%
Reassignment of Bug Assignee	80%	66.7%
No Comment Bugs	80%	66.7%
Non-Assignee Resolver of Bug	66.3%	56.7%
Closed-Reopen Ping Pong	93.3%	66.7%

"If a bug is reopened 2-3 times then a concern is raised so that it does not happen again" We asked practitioners their expert opinion about the potential root causes of this smell and some of the potential root causes are as follows;

- Insufficient unit testing (26/30)
- Ambiguous bug specifications (20/30)
- Changed bug scope (15/30)
- Poorly/incorrectly fixed bugs (26/30)
- Tester doing is really bad job (If the bug wasn't fixed – they shouldn't close it but reactivate it back to the developer) (16/30)

Chapter 7

Hypothesis Testing

To statistically analyze the effect of process smells on time to resolution (TTR) and bug reopening count, we propose to use statistical tests. The reason we choose TTR and reopening count for statistical impact analysis out of all other factors is these two parameters can easily be quantified from the dataset we have. For TTR, we calculated the total time (in seconds) from where the bug was opened to the time it was resolved and closed. Similarly, for reopening count, we are counting for how many times the bug is reopened by counting the number of reopened states of the bug during its life cycle. As per our analysis, some process smells affect the TTR of the bug. Hypothetically, the bugs with process smells are likely to have greater resolution time and more reopens as compared to the bugs which have no process smells. However, some of the process smells have no effect on the resolution and reopening of the bug. For example, if the bug has no comment that means there is no time spent on the discussion related to the bug and the bug is fixed quickly.

In order to statistically analyze the effect of process smells on TTR and reopen counts of bugs, we used *Mann-Whitney U Test*. We used a non-parametric statistical test because when we visualized our data (using box plot), we observed it is not normally distributed and we are comparing two independent groups i.e. Bugs with process smell and Bugs with no process smell. We tested each alternative

Table 7.1: p-values for statistical tests of TTR and smelling bug instances for six projects. Expected Results (Green Color), Unexpected Results (Yellow Color)

	Confluence	**Jira**	**GCC**	**Evergreen**	**Mongo DB**	**Wireshark**
Unassigned Bugs	1.00e+00	5.32e-01	1.17e-149	9.98e-01	8.35e-02	9.99e-01
No Link to Commit	NaN	NaN	2.91e-15	7.98e-01	1.43e-01	9.97e-01
Ignored Bugs	6.27e-17	3.09e-08	0.00e+00	2.52e-05	2.10e-122	0.00e+00
Bugs Assigned to a Team	NaN	NaN	NaN	1.40e-15	5.46e-258	NaN
Missing Priority	NaN	NaN	NaN	NaN	6.16e-01	NaN
Not Referenced Duplicate	9.74e-01	9.44e-01	NaN	3.65e-01	9.63e-01	NaN
Missing Environment Info	1.00e+00	1.00e+00	NaN	1.00e+00	1.00e+00	NaN
Missing Severity	1.00e+00	1.00e+00	NaN	NaN	NaN	NaN
Reassignment of Assignee	1.06e-27	2.41e-39	2.15e-62	4.53e-56	0.00e+00	5.62e-02
No Comment Bugs	1.00e+00	4.68e-98	9.94e-01	9.99e-01	1.67e-01	9.99e-01
Non-Assignee Resolver	2.25e-208	2.60e-67	5.98e-135	3.82e-14	1.82e-221	5.27e-02
Closed-Reopen Ping Pong	9.93e-97	4.54e-24	1.00e+00	1.09e-09	1.33e-81	7.67e-05

hypothesis proposed for different BT process smells by setting the significance level, $\alpha = 0.05$ (with a level of 95% confidence).

In Table 7.1 and 7.2, the p-values for statistical analysis of TTR and reopen count of bugs having process smells are given respectively. The coloring scheme is used to show the expected and unexpected statistical results. The green color is used to show the expected results (the results that are in accordance with our hypothesis) while the yellow color is representing the unexpected statistical results. NaN indicates that for this particular process smell we cannot run our statistical test because it cannot be compared i.e. either all bugs have smells or none of the bugs has that particular smell.

7.1 TTR Vs Process Smells

To analyze the effect of process smells on the TTR, we first calculated the required parameter against each bug. For TTR, we calculated the time of resolution of the bug (in seconds) i.e. how long did the bug take from opening till closing. afterward, we defined our null and alternative hypotheses. Considering the process smells' definitions and their potential impacts on the software development process, some of the process smells are likely to have no effect on TTR. afterward, we proposed null and alternative hypotheses for each process smell which are as follows;

$$H_{0_a} = Bugs\ with < \textbf{\textit{smell type}} > have\ no\ effect\ on\ TTR$$

$$H_{1_a} = Bugs\ with < \textbf{\textit{smell type}} > have\ greater\ TTR$$

The effect of process smells on the resolution time of the bug is statistically significant as we can observe it from the p-values obtained in table 7.1. However, we also got some unexpected results for some of the process smells and projects. For 'Missing Environment Information', 'Missing Priority' and Missing Severity' the p-values are highlighted with yellow color which means we get unexpected p-values. The TTR should be higher if severity or environment information is missing but the p-values we obtained are higher than 0.05 which means we cannot reject the H_{0_a}. However, if we look at the 'Missing Environment Smell' ratio in Table 5.6, it is very low as compared to other smells. Therefore, we could not say that effect does not exist rather the strength of the parameter could fall. The same is case with the bugs with no priority, the priority ratio is less than 1% for the MongoDB project. For the bugs that do not have severity information they could possibly take a longer time to resolve as the developer would not have any information about severity.

If we observe the p-values for 'Unassigned Bugs' of all projects except for GCC are greater than 0.05 which means we cannot reject the H_{0_a}. Hypothetically, unassigned bugs should have higher TTR. However, it could be a possibility

that sometimes the bugs are of trivial nature and they are resolved immediately without assigning to any developer.

For process smells like 'No Link to Bug-Fixing Commit' and 'Not Referenced Duplicates', the TTR should not likely be affected. These process smells have no potential impact on the resolution of bugs. Therefore, p-values for these two smell categories are higher than 0.05 showing that we are 'Accepting' our null hypothesis (H_{0_a}) i.e. these two process smell have no effect on TTR . For GCC project, the 'No Link to Commit' has an exceptionally high ratio of almost 90% (Table 5.6), it might be the reason for unexpected statistical value.

For all other smells like 'Bugs Assigned to Team', 'Ignored Bugs', 'Reassignment of Bug Assignee', 'Non Assignee Resolver', and 'Closed- Reopen Ping-Pong' ideally, TTR should be greater and the p-values obtained are all less than 0.05 which means we can reject the H_{0_a}. They have statistically significant results with no exceptions showing that bugs having these process smells are likely to have higher TTR.

For 'No Comment Bugs' the TTR is not likely to be greater. As no comments means less discussion and quick bug fixing implying that the null hypothesis (H_{0_a}) should be accepted and p-values obtained are all greater than 0.05 except for Jira project.

Overall, we have observed that the Jira project gave more unexpected statistical results (Table 7.1) than any other project. We tried to look for the project's BT guidelines and we found out that one of the reasons could be for this particular project they are using an additional metric during their bug reporting called User Impact Score (UIS)[1], which is derived uniquely for each bug. It considers the number of users who are affected, current interest, the severity of the bug, and the percentage of people that are affected per instance. Hereby, we can conclude that the effect of bug tracking process smells on TTR is statistically significant.

[1]https://confluence.atlassian.com/support/atlassian-data-center-and-server-bug-fix-policy-201294573.html

Table 7.2: p-values for statistical tests of reopen count and smelling bug instances for six. Expected Results (Green Color), Unexpected Results (Yellow Color)

	Confluence	**Jira**	**GCC**	**Evergreen**	**Mongo DB**	**Wireshark**
Unassigned Bugs	1.00	9.99e-01	9.99e-01	7.30e-01	9.91e-01	9.81e-01
No Link to Commit	NaN	NaN	9.99e-01	8.90e-01	1.92e-03	5.23e-02
Ignored Bugs	NaN	NaN	NaN	NaN	NaN	NaN
Bugs Assigned to a Team	NaN	NaN	NaN	9.99e-01	9.95e-01	NaN
Missing Priority	NaN	NaN	NaN	NaN	6.17e-01	NaN
Not Referenced Duplicate	2.61e-01	6.87e-01	NaN	7.75e-01	6.07e-01	NaN
Missing Environment Info	9.99e-01	9.99e-01	NaN	9.38e-01	9.99e-01	NaN
Missing Severity	4.38e-01	2.72e-06	NaN	NaN	NaN	NaN
Reassignment of Assignee	5.09e-125	7.89e-97	7.65e-28	5.08e-12	3.54e-39	4.51e-06
No Comment Bugs	1.00e+00	1.00e+00	7.26e-01	9.98e-01	1.00e+00	8.53e-01
Non-Assignee Resolver	3.76e-38	3.23e-16	4.21e-30	1.09e-01	2.11e-10	6.44e-01
Closed-Reopen Ping Pong	0.00e+00	0.00e+00	0.00e+00	0.00e+00	0.00e+00	0.00e+00

7.2 Reopen Count Vs Process Smells

BT process smells have an effect on the reopening of the bug. To analyze whether this effect is statistically significant or not we have done statistical analysis using the reopened counts of bugs with process smells and bugs with no process smells. For reopening count, we are counting how many times the bug is being reopened during its life cycle for each bug. afterward, we defined our null and alternative hypotheses. The null and alternative hypotheses statements are

$$H_{0_b} = \textit{Bugs with} < \textbf{\textit{smell type}} > \textit{have no effect on reopen count}$$

$$H_{1_b} = \textit{Bugs with} < \textbf{\textit{smell type}} > \textit{have greater reopen count}$$

For 'Ignored Bugs' we cannot run a statistical test for reopen counts because, by definition, ignored bugs are the bugs that are never resolved. Therefore, we cannot consider reopens in this scenario and we are writing it NaN (Table 7.2). 'No Link to Bug-Fixing Commit', 'Not Referenced Duplicates', Missing Priority' and 'Missing Severity' process smells do not affect the reopen count of the bug i.e. H_{0_b} should be accepted for these process smells. The p-value of 'No Link to Bug-Fixing Commits' for MongoDB project is unexpectedly less which means we have to reject the H_{0_b}. Similarly, for 'Missing Severity' the p-values are less than 0.05 which means we have to reject the H_{0_b}.

For process smells like 'Unassigned Bugs', ' Bugs Assigned to a Team', 'Reassignment of Bug Assignee', 'No Comments', 'Non-Assignee Resolver' and 'Closed-Reopen Ping-Pong' the reopen counts should be greater i.e. H_{0_b} should be rejected. The bugs having these process smells have higher chances of getting reopened at any later time. The statistical results in Table 7.2 shows that some of the p-values we obtained are unexpected i.e. greater than 0.05 and we failed to reject the null hypothesis. For 'Unassigned Bugs' and 'No Comment Bugs' p-values we obtained for all projects are greater than 0.05 and are failed to support our alternative hypothesis. Unassigned bugs might be the ones that are of trivial nature and are resolved straight forwardly without being assigned to any developer and such bugs do not need reopens. The p-values for process smell 'Non-Assignee Resolver' are unexpectedly greater for Evergreen and Wireshark projects failing to reject the null hypothesis.

'Not Referenced Duplicates' does not cause the reopening of bug. Therefore, we should accept H_{0_b} in this case. The p-values obtained are greater than 0.05 which means our finding is statistically significant.

Statistical analysis for reopen counts might not be very reliable. As in many projects, there is a possibility that instead of reopening the same bug again and again, a new bug is being opened. Therefore, we cannot come up with a conclusion here.

Chapter 8

Discussion

8.1 Context Dependency of Smells

The occurrence of BT process smells depends on various factors. In this chapter, we are going to discuss all those factors that play an important role in the BT process smells' occurrence. Firstly, the difference among the BTS can be one of the major reasons of having process smells in any project as the different organization uses different BT tools and every tool provides its own features. There are many well-known BT tools available like Jira, Bugzilla, Github Issues, etc. Each BT tool differs from the other. For example, Jira offers customization of fields i.e. users can define/add custom fields according to their requirements for any project. They can make any field mandatory and delete any field as per their requirement like the *severity* field is not defined in Jira by default. Therefore, if the user does not add the severity field while using Jira as their BT tool, it is going to make bug reports smelly. Whereas, if any organization is using Bugzilla as a BT tool they have a pre-defined set of bug fields. In Bugzilla, severity is a mandatory field. Therefore, the process smell would not occur. In short, the BT tool is one of the important factors causing process smells.

Secondly, organizational practices can be a potential factor in the occurrence

of BT process smells. As organizations follow their own set of rules and best practice for the BT process. For example, in some organizations bugs are marked resolved by team leads only. While in any other organization anyone who is working on the bug can mark the bug as resolved/closed. As in our survey, one of the respondents writes in an open-ended question; *"some organizations have a trend in which all bugs are assigned to the lead first who then reassigns them to the appropriate developer"* (which causes reassignment of bugs). Moreover, organizations have defined their own bug states, state transitions, and paths that a bug should follow as per their requirements. Therefore, organizational practices being followed during project development could be a cause of the occurrence of the BT process smells in these projects.

Thirdly, the difference among the projects. During our analysis, we have seen different state diagrams of the different projects that were using Jira as their BT tool. We have observed the unusual transitions from one state to another state during the bug life cycle. Even though all of the projects we analyzed were using Jira still their state diagrams were very different from each other. One reason could be that there is no fixed defined path (or transition from one state to another) that a bug should follow during its life cycle.

8.2 Practical Advice to Practitioners

Software development relies heavily on the BT process. In order to fix bugs more efficiently, the BT process should be effective. While fixing the bug, the first most important thing is to report the bug effectively. If the bug is not reported correctly, developers will most likely dismiss the bug saying it is not reproducible. Writing a good and effective bug report is a skill.

For a better bug tracking process and BTS, we have given some potential practical advice for practitioners.

- To simplify the bug tracking process, there should be a proper guideline that

practitioners would use to achieve milestones in the bug tracking process.

- When utilizing BTS, one of the first things practitioners need to do is set field templates. It is possible to achieve this by making key bug report fields obligatory (e.g. assignee, severity, environment information). This method can help prevent many process smells.

- Practitioners should try to reproduce a bug numerous times before reporting it. The bug should be repeatable. The information provided should be sufficient to reproduce the bug reliably. Even if the bug is not always reproducible, the practitioner can still report it by indicating that it occurs on a regular basis.

- Creating a BT process flow within an organization/project. Practitioners should consider how bugs spread throughout their company. This will eliminate lost reports, ensuring that everyone knows who is responsible for what, and ensure that everyone is aware of the procedures to follow. A bug should be required to follow a set path from step to step as it progresses through the organization.

- Practitioners should look for the same or similar bugs in other modules. The code for numerous similar modules may be reused by the developer. As a result, there's a higher chance that a problem in one module will propagate to other modules that are similar. Practitioners may also search for a more severe version of the bug they found.

- Duplicates should be avoided. Always try to seek existing reports in your BTS as a first step.

- Professional BTS should be used, and newcomers should be trained on how to utilize a BTS efficiently. Workshops on the BT process should be held, and practitioners should be instructed on how to use BTS and how to prevent the smells associated with the BT process.

Chapter 9

Threats to Validity

This chapter discusses the threats to the internal, construct, and external validity of our study. Although our analysis provides some useful and interesting results, there are a number of threats to our analysis's validity that must be examined. There are four major risks to our analysis's internal validity.

Organizational Practices: As different organizations follow their own set of rules and best practices, a few of our process smells (Bugs Assigned to Team and Non-Assignee Bug Resolver) can be subject to discussion. For example, if bugs are marked resolved by only the team leads in a project, we cannot claim that non-assignee resolver of bug smell exists. Because it is not a mismatch between the assignee and resolving person due to organizational discrepancies, it is simply the organizational rule. These kinds of smells' detection methods should be adapted to organizations and systems. Regarding the *Missing Environment Information* smell, bug reporters may prefer indicating the environment information into the bug description and leaving the corresponding fields blank. We count such a case as missing environment information, although it may not be, since that information might be mentioned inside the bug description. However, we believe this is still a bad practice as it prevents people from searching and filtering environment information.

Tool Dependency: For the projects that use Jira, one might argue that *the Missing Severity* smell should not be counted since Jira does not provide a severity field by default. Despite the lack of severity fields, Jira still provides mechanisms to add custom fields. It is the project members' responsibility to customize the tool. Since severity is an important property of a bug, not tracking this property should still be counted as a process smell.

Configurable Thresholds: Within the smell detection methods, we made a few assumptions on the configurable parameters and definitions (e.g. ignored bugs time duration is six months and as for reassignment of bug, if a bug is reassigned more than once). These thresholds are subject to discussion and could be configured depending on the project. Also, we surveyed experienced software practitioners for expert opinions about the thresholds and definitions of our proposed BT process smells and adjust our thresholds as per survey results.

Practitioner's Survey: We also discovered a number of threats associated with our study's survey. One of the possible threats to the legitimacy of the survey is the smell definitions may be misinterpreted by the respondents. To address these concerns, a full description of each process smell is provided, along with a real-life example. The respondents were instructed to contact the authors if they had any questions about the survey. We did a pilot test questionnaire before sending the survey to the software practitioners and clarified the items that could potentially pose an issue. Many respondents effectively supplied many responses in one response to the open-ended questions. When asked about the actions taken in their companies for avoiding process smells, they mentioned some of the actions. All of the comments provided by the participants were taken into consideration during the survey synthesis in order to stay faithful to their feedback. We're sharing the responses in chapter 6. The survey was sent out to the authors' network and was aimed at those who actively use or have expertise with bug tracking systems. Despite the fact that the survey participants come from different companies and have different job titles, general conclusions cannot be drawn. As a result, we can not assume that the findings will apply to all software product systems that use bug tracking systems. The survey's sample is a convenience sample that was sent to the authors' network, as previously stated.

Also, the survey is a bit longer and takes around 45-50 mins.

To improve the study's replicability, we share the traceability table from primary studies of process smell categories for the MLR, the dataset that we used to mine BT process smells, and the source code[1].

Construct validity shows how well the outcomes on the instrument are indicative of the theoretical concept and how well the measure 'behaves' in a way that is consistent with theoretical hypotheses. We described two measures in our research, TTR which indicates the resolution time of the bug and reopen count which shows how many times the bug is reopened during its life-cycle. For TTR, we intend to measure the time a bug takes to resolve. Therefore, we calculated the time from where the bug is opened till the time when the bug is closed. similarly, for reopen counts, we counted the number of times the bug was in the reopened state during its life cycle. Despite the fact that these assumptions are fair, these measures may not accurately reflect the intended measure. Moreover, reopening counts might not be reliable.

External validity threats are a concern with respect to the generalization of results, i.e. to which extent we can generalize our results. To mitigate these threats we conducted our study on six projects from two different tools, i.e. Jira & Bugzilla. To improve the generalizability, we are planning to conduct our study on more projects and tools in the future.

[1] `https://figshare.com/s/3f402ee60f46ea1fe99c` and `https://figshare.com/s/f8ce1820d9a371a73071`

Chapter 10

Conclusion and Future Work

In this study, based on the results of an MLR, we proposed a taxonomy of 12 process smells in the bug tracking process. To observe their presence in practice, we conducted an empirical evaluation of BT process smells by mining bug reports of six projects from Jira & Bugzilla (GCC, Wireshark, Jira Server & Data Center, Confluence Server & Data Center, MongoDB Core Server, and Evergreen). We conducted a survey with software practitioners to get experts' opinions about the BT process smells. We also run a statistical test to analyze whether the impacts of BT process smells on TTR and reopen counts are statistically significant or not. We can summarize the main contributions of our study as follows:

- Proposed a novel taxonomy of BT process smells (Table 4.1), based on a multivocal literature review.
- Performed an empirical analysis with six OSS projects to demonstrate that all the process smells occur in software bug reports with varying ratios.
- Observed that over time, the occurrence of some specific BT process smells in software projects is decreased. The reason for this improvement might be associated with the advancements in BT tools and improved best practices for the BT process.

- Each process smell is assessed in our study by obtaining the opinions of domain experts i.e. software practitioners.
- Analyzed the statistical impact of process smells on TTR and reopen count of bugs.

The implications of our study for researchers and practitioners are three-fold. First, our proposed taxonomy can be used as a baseline to be extended by researchers. Second, for practitioners, the BT process can be enhanced by introducing suitable tooling for an improved BT process. Finally, our proposed taxonomy could be used for developing automated (or semi-automated) BT process smell detection systems by mining software repositories.

As future work, our taxonomy can be expanded to include more process smells and their empirical analysis. We can also expand this study to perform a similar empirical evaluation on a more diverse set of BT tools such as GitHub Issues. We can perform the impact analysis of each process smell on the BT process individually. Another future direction is the implementation of some professional tools to detect and analyze BT process smells. These tools can be configured for different platforms.

Bibliography

[1] B. Dit and A. Marcus, "Improving the readability of defect reports," in *Proceedings of the 2008 International Workshop on Recommendation Systems for Software Engineering - RSSE '08*, (New York, New York, USA), p. 47, ACM Press, 2008.

[2] T. Zimmermann, R. Premraj, J. Sillito, and S. Breu, "Improving bug tracking systems," in *2009 31st International Conference on Software Engineering - Companion Volume*, pp. 247–250, IEEE, 2009.

[3] T. Zimmermann, R. Premraj, N. Bettenburg, S. Just, A. Schroter, and C. Weiss, "What makes a good bug report?," *IEEE Transactions on Software Engineering*, vol. 36, pp. 618–643, sep 2010.

[4] X. Xia, D. Lo, M. Wen, E. Shihab, and B. Zhou, "An empirical study of bug report field reassignment," in *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, pp. 174–183, IEEE, feb 2014.

[5] M. Gupta, "Nirikshan: process mining software repositories to identify inefficiencies, imperfections, and enhance existing process capabilities," in *Companion Proceedings of the 36th International Conference on Software Engineering*, (New York, NY, USA), pp. 658–661, ACM, may 2014.

[6] M. Gupta and A. Sureka, "Nirikshan: mining bug report history for discovering process maps, inefficiencies and inconsistencies," in *Proceedings of the 7th India Software Engineering Conference on - ISEC '14*, (New York, New York, USA), pp. 1–10, ACM Press, 2014.

[7] K. A. Qamar, E. Sülün, and E. Tüzün, "Towards a taxonomy of bug tracking process smells: A quantitative analysis," in *2021 47th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pp. 138–147, 2021.

[8] H. Rocha, G. de Oliveira, M. T. Valente, and H. Marques-Neto, "Characterizing bug workflows in mozilla firefox," in *Proceedings of the 30th Brazilian Symposium on Software Engineering*, SBES '16, (New York, NY, USA), p. 43–52, Association for Computing Machinery, 2016.

[9] M. Gupta, A. Sureka, and S. Padmanabhuni, "Process mining multiple repositories for software defect resolution from control and organizational perspective," in *Proceedings of the 11th Working Conference on Mining Software Repositories - MSR 2014*, (New York, New York, USA), pp. 122–131, ACM Press, 2014.

[10] C. A. Halverson, J. B. Ellis, C. Danis, and W. A. Kellogg, "Designing task visualizations to support the coordination of work in software development," in *Proceedings of the 2006 20th Anniversary Conference on Computer supported Cooperative Work*, pp. 39–48, 2006.

[11] M. D'Ambros, M. Lanza, and M. Pinzger, "A bug's life: Visualizing a bug database," in *2007 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis*, pp. 113–120, IEEE, 2007.

[12] P. Knab, M. Pinzger, and H. C. Gall, "Visual patterns in issue tracking data," in *International Conference on Software Process*, pp. 222–233, Springer, 2010.

[13] A. J. Ko, B. A. Myers, and D. H. Chau, "A linguistic analysis of how people describe software problems," in *Visual Languages and Human-Centric Computing (VL/HCC'06)*, pp. 127–134, IEEE, 2006.

[14] P. Hooimeijer and W. Weimer, "Modeling bug report quality," in *Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering*, pp. 34–43, 2007.

[15] J. Anvik, L. Hiew, and G. C. Murphy, "Who should fix this bug?," in *Proceeding of the 28th International Conference on Software Engineering - ICSE '06*, vol. 2006, (New York, New York, USA), p. 361, ACM Press, 2006.

[16] J. Anvik and G. C. Murphy, "Determining implementation expertise from bug reports," in *Fourth International Workshop on Mining Software Repositories (MSR'07:ICSE Workshops 2007)*, pp. 2–2, IEEE, may 2007.

[17] E. Doğan and E. Tuzun, "Towards a taxonomy of code review smells," *Information and Software Technology*, vol. 142, no. 106737, 2022.

[18] M. Di Penta and D. A. Tamburri, "Combining quantitative and qualitative studies in empirical software engineering research," in *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pp. 499–500, IEEE, may 2017.

[19] V. Garousi, M. Felderer, and M. V. Mäntylä, "Guidelines for including grey literature and conducting multivocal literature reviews in software engineering," *Information and Software Technology*, vol. 106, pp. 101–121, 2019.

[20] S. Keele *et al.*, "Guidelines for performing systematic literature reviews in software engineering," tech. rep., Citeseer, 2007.

[21] R. Glass and T. DeMarco, *Software Creativity 2.0*. Online access: EBSCO Computers & Applied Sciences Complete, Developer.* Books, 2006.

[22] M. Ivarsson and T. Gorschek, "A method for evaluating rigor and industrial relevance of technology evaluations," *Empirical Software Engineering*, vol. 16, pp. 365–395, jun 2011.

[23] B. Kitchenham and S. L. Pfleeger, "Principles of survey research: part 5: populations and samples," *ACM SIGSOFT Software Engineering Notes*, vol. 27, no. 5, pp. 17–20, 2002.

[24] H. B. Mann and D. R. Whitney, "On a test of whether one of two random variables is stochastically larger than the other," *The annals of mathematical statistics*, pp. 50–60, 1947.

[25] C. Reis and R. D. M. Fortes, "An overview of the software engineering process and tools in the mozilla project," in *Proceedings of the Open Source Software Development Workshop*, no. figure 1, pp. 1–21, 2002.

[26] I. Aljarah, S. Banitaan, S. Abufardeh, W. Jin, and S. Salem, "Selecting discriminating terms for bug assignment: a formal analysis," in *Proceedings of the 7th International Conference on Predictive Models in Software Engineering - Promise '11*, (New York, New York, USA), pp. 1–7, ACM Press, 2011.

[27] G. Jeong, S. Kim, and T. Zimmermann, "Improving bug triage with bug tossing graphs," in *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE '09, (New York, NY, USA), p. 111–120, Association for Computing Machinery, 2009.

[28] S. Hill, "How to write a good bug report?," 2015. Accessed: 2021-03-01.

[29] J. Bridges, "8 steps for better issue management," 2019. Accessed: 2021-04-25.

[30] A. Bachmann, C. Bird, F. Rahman, P. Devanbu, and A. Bernstein, "The Missing Links: Bugs and Bug-Fix Commits," in *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE '10*, (New York, New York, USA), p. 97, ACM Press, 2010.

[31] T. F. Bissyandé, F. Thung, S. Wang, D. Lo, L. Jiang, and L. Reveillere, "Empirical evaluation of bug linking," in *2013 17th European Conference on Software Maintenance and Reengineering*, pp. 89–98, IEEE, 2013.

[32] M. Ricklin, "Defect tracking best practices," 2009. Accessed: 2021-03-01.

[33] H. Rocha, G. de Oliveira, M. T. Valente, and H. Marques-Neto, "Characterizing bug workflows in mozilla firefox," in *Proceedings of the 30th Brazilian Symposium on Software Engineering - SBES '16*, (New York, New York, USA), pp. 43–52, ACM Press, 2016.

[34] L. Floridi, "Faultless responsibility: on the nature and allocation of moral responsibility for distributed moral actions," *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, vol. 374, p. 20160112, dec 2016.

[35] L. Jonsson, "Increasing anomaly handling efficiency in large organizations using applied machine learning," in *2013 35th International Conference on Software Engineering (ICSE)*, pp. 1361–1364, IEEE, may 2013.

[36] L. Jonsson, D. Broman, K. Sandahl, and S. Eldh, "Towards automated anomaly report assignment in large complex systems using stacked generalization," in *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, pp. 437–446, IEEE, apr 2012.

[37] N. Kaushik, M. Amoui, L. Tahvildari, W. Liu, and S. Li, "Defect prioritization in the software industry: challenges and opportunities," in *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, pp. 70–73, IEEE, mar 2013.

[38] S. Admin, "10 Tips of writing effective bug reports," 2014. Accessed: 2021-03-01.

[39] M. Mitrev, "The anatomy of a good bug report," 2020. Accessed: 2021-03-01.

[40] J. Kanwal and O. Maqbool, "Bug prioritization to facilitate bug report triage," *Journal of Computer Science and Technology*, vol. 27, no. 2, pp. 397–412, 2012.

[41] K. Aggarwal, F. Timbers, T. Rutgers, A. Hindle, E. Stroulia, and R. Greiner, "Detecting duplicate bug reports with software engineering domain knowledge," *Journal of Software: Evolution and Process*, vol. 29, p. e1821, mar 2017.

[42] B. Kucuk and E. Tuzun, "Characterizing duplicate bugs: An empirical analysis," in *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 661–668, 2021.

[43] N. Bettenburg, R. Premraj, T. Zimmermann, and Sunghun Kim, "Duplicate bug reports considered harmful ... really?," in *2008 IEEE International Conference on Software Maintenance*, pp. 337–345, IEEE, 2008.

[44] E. Dmytriiev, "Best practices for effective defect tracking," 2016. Accessed: 2021-03-01.

[45] J. Arokiam and J. S. Bradbury, "Automatically predicting bug severity early in the development process," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: New Ideas and Emerging Results*, (New York, NY, USA), pp. 17–20, ACM, jun 2020.

[46] T. Zhang, J. Chen, G. Yang, B. Lee, and X. Luo, "Towards more accurate severity prediction and fixer recommendation of software bugs," *Journal of Systems and Software*, vol. 117, pp. 166–184, jul 2016.

[47] T. Menzies and A. Marcus, "Automated severity assessment of software defect reports," in *2008 IEEE International Conference on Software Maintenance*, pp. 346–355, IEEE, sep 2008.

[48] LuminosLabs, "Writing good bug report," 2020. Accessed: 2021-03-01.

[49] P. Runeson, M. Alexandersson, and O. Nyholm, "Detection of duplicate defect reports using natural language processing," in *29th International Conference on Software Engineering (ICSE'07)*, pp. 499–510, IEEE, may 2007.

[50] X. Xia, D. Lo, E. Shihab, X. Wang, and B. Zhou, "Automatic, high accuracy prediction of reopened bugs," *Automated Software Engineering*, vol. 22, pp. 75–109, mar 2015.

[51] H. Valdivia-Garcia, E. Shihab, and M. Nagappan, "Characterizing and predicting blocking bugs in open source Projects," in *Journal of Systems and Software*, vol. 143, pp. 44–58, sep 2018.

[52] F. Zhang, F. Khomh, Y. Zou, and A. E. Hassan, "An empirical study on factors impacting bug fixing time," in *2012 19th Working Conference on Reverse Engineering*, pp. 225–234, IEEE, oct 2012.

[53] E. Shihab, A. Ihara, Y. Kamei, W. M. Ibrahim, M. Ohira, B. Adams, A. E. Hassan, and K.-i. Matsumoto, "Predicting re-opened bugs: a case study on the eclipse project," in *2010 17th Working Conference on Reverse Engineering*, pp. 249–258, IEEE, oct 2010.

[54] AnalyseIT247, "Reopening issues after they have been resolved or closed," 2016. Accessed: 2021-03-01.

[55] T. Zhang, H. Jiang, X. Luo, and A. T. Chan, "A literature review of research in bug resolution: tasks, challenges and future directions," *The Computer Journal*, vol. 59, pp. 741–773, may 2016.

[56] S. Dueñas, V. Cosentino, G. Robles, and J. M. Gonzalez-Barahona, "Perceval: software project data at your will," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, (New York, NY, USA), pp. 1–4, ACM, may 2018.

[57] E. K. and S. A., "How to Write a Bug Report," 2016. Accessed: 2021-03-01.

[58] Instabug, "How to Write a Bug Report: The Ideal Bug Report," 2020. Accessed: 2021-03-01.

[59] Arena, "Defect Management—4 Steps to Better Products & Processes," 2020. Accessed: 2021-03-01.

[60] S. Kim and M. D. Ernst, "Prioritizing Warning Categories by Analyzing Software History," in *Fourth International Workshop on Mining Software Repositories (MSR'07:ICSE Workshops 2007)*, pp. 27–27, IEEE, may 2007.

[61] G. Canfora and L. Cerulo, "Supporting change request assignment in open source development," in *Proceedings of the 2006 ACM symposium on Applied computing - SAC '06*, vol. 2, (New York, New York, USA), p. 1767, ACM Press, 2006.

[62] J. Anvik and G. C. Murphy, "Reducing the effort of bug report triage: Recommenders for development-oriented decisions," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 20, no. 3, pp. 1–35, 2011.

[63] D. Lee, "How to write a bug report that will make your engineers love you," 2020. Accessed: 2021-03-01.

[64] S. N. Ahsan, J. Ferzund, and F. Wotawa, "Automatic Software Bug Triage System (BTS) Based on Latent Semantic Indexing and Support Vector Machine," in *2009 Fourth International Conference on Software Engineering Advances*, pp. 216–221, IEEE, sep 2009.

[65] S. Kim and M. D. Ernst, "Which warnings should I fix first?," in *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering - ESEC-FSE '07*, (New York, New York, USA), p. 45, ACM Press, 2007.

[66] T. Kremenek and D. Engler, "Z-Ranking: Using Statistical Analysis to Counter the Impact of Static Analysis Approximations," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 2694, (Berlin, Heidelberg), pp. 295–315, Springer, 2003.

[67] Kualitee, "The Good, The Bad and The Ugly Bug: Bug Tracking Best Practices," 2019. Accessed: 2021-03-01.

[68] Axosoft, "Bug Tracking Best Practices Guide," 2020. Accessed: 2021-03-01.

[69] software Testing Help, "How To Write A Good Bug Report? Tips And Tricks," 2007. Accessed: 2021-03-01.

[70] T. Coders, "Best Practices: Defect Reporting Techniques," 2020. Accessed: 2021-03-01.

[71] Musescore, "How to write a good bug report: step-by-step instructions," 2015. Accessed: 2021-03-01.

[72] L. v. Belle, "How to Write a Bug Report that Will Make Your Developers Happy," 2021. Accessed: 2021-03-01.

[73] Apache, "Issue Writing Guidelines," 2021. Accessed: 2021-03-01.

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

[75] M. Christensen, "How to Write an Effective Bug Report That Actually Gets Resolved (and Why Everyone Should)," tech. rep.

[76] A. Lamkanfi, S. Demeyer, E. Giger, and B. Goethals, "Predicting the severity of a reported bug," in *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*, pp. 1–10, IEEE, may 2010.

[77] R. Jindal, R. Malhotra, and A. Jain, "Prediction of defect severity by mining software project reports," *International Journal of System Assurance Engineering and Management*, vol. 8, pp. 334–351, jun 2017.

[78] K. Somasundaram and G. C. Murphy, "Automatic categorization of bug reports using latent Dirichlet allocation," in *Proceedings of the 5th India Software Engineering Conference on - ISEC '12*, (New York, New York, USA), pp. 125–130, ACM Press, 2012.

[79] A. Sajedi-Badashian and E. Stroulia, "Guidelines for evaluating bug-assignment research," *Journal of Software: Evolution and Process*, vol. 32, p. 2250, sep 2020.

[80] V. Akila, G. Zayaraz, and V. Govindasamy, "Effective bug triage–a framework," *Procedia Computer Science*, vol. 48, pp. 114–120, 2015.

[81] 360Logica, "How to write a good bug report? Tips and Tricks," 2012. Accessed: 2021-03-01.

Appendix A

Literature Sources

As the result of the MLR, 25 gray literature sources and 35 academic studies are shared with the corresponding smells in Tables A.1 and A.2.

Table A.1: MLR Results with the Related Smells

Source	Type	Unassigned Bugs	No Link to Commit	Ignored Bugs	Bugs Assigned to Team	Missing Priority	Not Referenced Duplicates
An overview of the software engineering process and tools in the Mozilla project [25]	White	✓					
Who should fix this bug? [15]	White	✓					
How to write a good bug report? [28]	Gray	✓	✓				
How to Write a Bug Report [57]	White					✓	
The missing links: bugs and bug-fix commits [30]	White		✓				
An empirical study on factors impacting bug fixing time [52]	White			✓			
How to Write a Bug Report: The Ideal Bug Report [58]	Gray					✓	
Towards automated anomaly report assignment in large complex systems using stacked generalization [36]	White				✓		
Increasing anomaly handling efficiency in large organizations using applied machine learning [35]	White				✓		
Defect prioritization in the software industry: challenges and opportunities [37]	White					✓	
The Anatomy Of a Good Bug Report [39]	Gray					✓	
10 Tips of Writing effective Bug Reports [38]	Gray					✓	
Detecting duplicate bug reports with software engineering domain knowledge [41]	White						✓
Characterizing bug workflows in mozilla firefox [33]	White			✓			
How long will it take to fix this bug? [2]	White	✓					
Towards more accurate severity prediction and fixer recommendation of software bugs [46]	White	✓					
Defect Management:4 Steps to Better Products & Processes [59]	Gray					✓	
Prioritizing warning categories by analyzing software history [60]	White					✓	
Supporting change request assignment in open source development [61]	White	✓					
Determining implementation expertise from bug reports [16]	White	✓					
Reducing the effort of bug report triage: Recommenders for development-oriented decisions [62]	White	✓					
How to write a bug report that will make your engineers love you [63]	Gray					✓	
Selecting discriminating terms for bug assignment: A formal analysis [26]	White	✓					
Automatic software bug triage system (BTS) based on Latent Semantic Indexing and Support Vector Machine [64]	White	✓					
Improving bug triage with bug tossing graphs [27]	White	✓					
Which warnings should I fix first? [65]	White					✓	
Z-Ranking: Using statistical analysis to counter the impact of static analysis approximations [66]	White					✓	
Duplicate bug reports considered harmful... really? [43]	White						✓
8 Steps for Better Issue Management [29]	Gray	✓					
Best Practices for Effective Defect Tracking [44]	Gray					✓	
10 Tips of Writing Efficient Defect Report [38]	Gray					✓	
Defect Tracking Best Practices [32]	Gray	✓				✓	
The Good, The Bad and The Ugly Bug: Bug Tracking Best Practices [67]	Gray					✓	
Bug Tracking Best Practices Guide [68]	Gray	✓		✓			
How To Write A Good Bug Report? Tips And Tricks [69]	Gray	✓				✓	
Best Practices: Defect Reporting Techniques [70]	Gray					✓	
How to write a good bug report: step-by-step instructions [71]	Gray					✓	
How to write a bug report that will make your developers happy [72]	Gray					✓	
Issue Writing Guidelines [73]	Gray	✓					
How to write a good bug report? Tips and Tricks []	Gray	✓				✓	

Table A.2: MLR Results with the Related Smells

Source	Type	Missing Environment Info	Missing Severity	Reassignment of bug Bug Assignee	No Comment Bugs	Non Assignee Resolver of Bugs	Closed Reopen Ping-Pong
Who should fix this bug? [15]	White			✓			
How to write a good bug report? [28]	Gray	✓	✓				
How to Write a Bug Report [57]	Gray		✓				
How to Write a Bug Report: The Ideal Bug Report [58]	Gray	✓	✓				
The Anatomy Of a Good Bug Report [39]	Gray	✓			✓		
Predicting eclipse bug lifetimes [74]	White				✓		
Improving Bug Tracking Systems [2]	White	✓					
Best Practices for Effective Defect Tracking [44]	Gray	✓					
How to Write an Effective Bug Report That Actually Gets Resolved (and Why Everyone Should) [75]	Gray	✓	✓				
What makes a good bug report? [3]	White	✓					
Predicting the severity of a reported bug [76]	White		✓				
Prediction of defect severity by mining software project reports [77]	White		✓				
Towards more accurate severity prediction and fixer recommendation of software bugs [46]	White		✓				
Automatic categorization of bug reports using latent dirichlet allocation [78]	White			✓			
Writing Good Bug Report [48]	Gray				✓		
Improving the readability of defect reports [1]	White				✓		
Predicting re-opened bugs: A case study on the eclipse project [53]	White						✓
Reopening Issues After They Have Been Resolved Or Closed [75]	Gray						✓
Guidelines for evaluating bug-assignment research [79]	Gray					✓	
Effective bug triage–a framework [80]	Gray					✓	
An empirical study of bug report field reassignment [4]	Gray			✓			
Detection of duplicate defect reports using natural language processing [49]	Gray				✓		
Automatically predicting bug severity early in the development process [45]	Gray		✓				
How to write a bug report that will make your engineers love you [63]	Gray	✓	✓				
Improving bug triage with bug tossing graphs [27]	White			✓			
8 Steps for Better Issue Management [29]	Gray					✓	
Best Practices for Effective Defect Tracking [44]	Gray	✓	✓				
10 Tips of Writing Efficient Defect Report [38]	Gray		✓				
Bug Tracking Best Practices Guide [68]	Gray	✓	✓				
How To Write A Good Bug Report? Tips And Tricks [69]	Gray	✓	✓				
Best Practices: Defect Reporting Techniques [70]	Gray		✓				
How to write a good bug report: step-by-step instructions [71]	Gray	✓	✓				
How to write a bug report that will make your developers happy [72]	Gray	✓	✓				
Issue Writing Guidelines [73]	Gray	✓					
How to write a good bug report? Tips and Tricks [81]	Gray	✓	✓				

Appendix B

Code and Reproducibility

The source code for this study, the data we used for empirical analysis as well as the questions from the practitioners survey, are publicly shared online at Figshare [1].

[1] https://figshare.com/s/3f402ee60f46ea1fe99c