Towards a Taxonomy of Bug Tracking Process Smells: A Quantitative Analysis

Khushbakht Ali Qamar

Department of Computer Engineering
Bilkent University
Ankara, Turkey
ali.qamar@bilkent.edu.tr

Emre Sülün

Department of Computer Engineering
Bilkent University
Ankara, Turkey
emre.sulun@bilkent.edu.tr

Eray Tüzün

Department of Computer Engineering
Bilkent University
Ankara, Turkey
eraytuzun@cs.bilkent.edu.tr

Abstract—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.

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 inspect bug reports collected from six projects. Among these projects, four of them are Jira-based (MongoDB Core Server, Evergreen, Confluence Server & Data Center, Jira Server & Data Center) and the other two are Bugzilla-based (GCC and Wireshark). We observed that a considerable amount of bug tracking process smells exist in all projects with varying ratios.

Index Terms—the bug tracking system, process mining, conformance checking, anti-patterns, bug tracking smells, process smell

I. INTRODUCTION

Bug tracking (BT) is a methodology for reporting and keeping track of the bugs in software products. In a Bug Tracking System (BTS), each item is followed and tracked through to completion with informative reports which display the progress. 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 literature¹. Practitioners' non-compliance with the best practices impedes the benefits of the BT process and should be avoided.

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 OSS 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 the best of our knowledge, this is the first systematic study to collect bad practices in the process of BT. 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 RQ, 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 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). We aim to analyze the frequency of each process smell over time to answer this research question.

The empirical results of our study indicate that the BT process smells introduced in our taxonomy exist in every project with different ratios. Also, we observed that over time, the occurrence of bug tracking process smells in some software projects are decreased. The reason for this might be associated with the advancements in BT tools and improved best practices for the BT process.

This study was supported by The Scientific and Technological Research Council of Turkey (TUBITAK) 1505 program. Project Number: 5200078

¹<https://bugs.eclipse.org/bugs/page.cgi?id=bug-writing.html>

The following section discusses the related work. Section III explains our research methodology. Section IV demonstrates our BT process smells taxonomy and the technique for detecting each type of smell. Section V gives the empirical evaluation of BT process smells on six projects. Section VI addresses the validity threats of this study and lastly, Section VII presents our conclusion and future work.

II. RELATED WORK

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 [5], [6] puts 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 Gupta et al. [7], 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. [8] 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. [9] 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. An interactive approach was introduced by Knab et al. [10] 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. [11] 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. [12] presume that high-quality bug reports are addressed faster as compared to the 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.

III. RESEARCH METHODOLOGY

In our research, we follow a mixed-methods-based approach. We used this research methodology to support our qualitative analysis with quantitative results [13]. 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 Section V). To identify the bad practices, we used Multivocal Literature Review (MLR) as described in Section III-A through III-D.

A. Review Method

We selected MLR as a review method following the guidelines of Garousi et al. [14]. Generally in an SLR, the search

Fig. 1. Flow Diagram of MLR

is only limited to the academic literature [15]. However, it is seen that many software developers do not publish their work formally in academic forums [16], that is to say if we are not considering the gray literature (GL), then we might limit the 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 [17] 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 1.

B. 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 "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 [15], 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.

C. 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. [14].

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*.

D. 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 online² for ensuring the transparency of our study.

IV. TAXONOMY OF BUG TRACKING PROCESS SMELLS (RQ1)

RQ1 is about compiling the bad practices which are being followed by developers 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 and this leads us to 12 BT process smells. A summary of the proposed BT process smells taxonomy is given in Table I. In this section, each smell is introduced in the following format: (1) a detailed description of the smell (2) our smell detection methodology.

A. 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. [18]. 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 [19], [20]. 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 [21]. Thus, we consider it a process smell and call it *unassigned bugs*. Potential impacts of not assigning a developer to a bug could be loss of traceability [22] and delays in bug resolution [23].

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.

B. 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 [24]. 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 [25]. It also affects the related software development tasks such as prediction of bug locations, recommendation of bug fixes, and software cost [25].

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.

C. 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 [26]. 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 (six 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 [27].

Smell Detection Method: We compare the date the bug was created and the date of the first activity in the bug history. If the first activity is at least six months after the bug is created, we consider it a smell.

D. 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*" [28]. Whenever a bug is assigned to any team, and it is not specified which member of the team is going to solve

²<https://figshare.com/s/f8ce1820d9a371a73071>

TABLE I
SUMMARY TABLE OF BUG TRACKING PROCESS SMELLS

Smell Name	Description
Unassigned Bugs	No assignee, but the bug is fixed and closed
No Link to Commit Bug-Fixing Commit	If a proper bug is closed without any link to bug-fixing commit in the reference section
Ignored Bugs	Bugs that are left open for a long time or bugs that have incomplete resolution
Bugs Assigned to a Team	The bugs which are assigned to a team but not a particular assignee
Missing Priority	The priority of the bug has not been set
Not Referenced Duplicates	Duplicate bugs that do not have the reference to their original bug
Missing Environment Information	Environment information (Version, OS, and Component of the product) about the bug is missing
Missing Severity	The severity of the bug has not been set
Reassignment of Bug Assignee	Fixer for the bug is assigned more than once to a certain bug
No Comment Bugs	A resolved bug with no comments
Non-Assignee Resolver of Bug	When a bug is resolved by any person other than assignee(s)
Closed-Reopen Ping Pong	Bugs which are reopened several times

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 [29]. The problem of *team bug-assignment* was introduced by Jonsson et al., [29], [30] in which instead of a single developer, bugs are redirected to one of the many accessible teams.

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.

E. Missing Priority

During the life cycle of a bug, the bug's priority plays a very important role in its resolution [31]. 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 [32], [33]. 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 affect on the system. We are considering not prioritized bugs as a process smell and call it *missing priority*. The potential *impact* of this process smell is on the development-oriented decisions (time and resource allocation) [34].

Smell Detection Method: We check whether the priority field is valid or not. Some of the invalid priority strings are *None*, *Not Evaluated*, and *"-"*.

F. 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. [35], [36]. 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 [37].

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.

G. Missing Environment Information

During the BT process, it is necessary to know where the bug has been encountered [33] as bugs often only happen in certain environments [38]. 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 [21]).

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 [32].

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.

H. 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 [39]. 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. [21], [33]. 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 [40]. 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 [41].

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.

I. Reassignment of Bug Assignee

Research shows that the time-to-correction for a bug is increased by the reassignments of developers to a bug [20]. 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].

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 once, then we count it as a smell.

J. No Comment Bugs

Generally, one of the important sources of information for developers during the software development life cycle is the bug reports and the comments that are associated with bug reports [1], [33]. When stakeholders or developers do not understand the bug then at that time comments play a vital part [42].

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 [43]. In proposing the Bug Reopen predictor [44] features like description features, comment features, and meta-features are being used. For identifying the blocking bugs [45], 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 reopen prediction [12], [46].

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.

K. 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 [22].

Smell Detection Method: First, we check whether the bug is assigned and closed. If so, we compare the assignee and the person who resolved the bug.

L. 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 reproduced by the developer or improperly tested fix). Reopened bugs reduce the overall software quality, increased maintenance expenses, as well as unnecessary rework by busy developers [47]. 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 [48]. 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 [49]. Reopened bugs also increase development costs, affect the quality of product,

TABLE II
THE DETAILS OF PROJECTS

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

prediction of release dates reduce the team morale leading to poor productivity [48].

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.

V. RESULTS (RQ2)

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.

A. Dataset

We analyzed six large-scale software projects that have a publicly available issue tracking systems. The projects are GCC³, Wireshark⁴, Confluence Server and Data Center⁵, Jira Server and Data Center⁶, MongoDB Core Server⁷, and Evergreen⁸. 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).

We utilized the Perceval tool [50] to fetch the issue tracking histories of those 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 II. Also, we share the project history datasets in our replication package⁹.

³<https://gcc.gnu.org/bugzilla/>

⁴<https://bugs.wireshark.org/bugzilla/>

⁵<https://jira.atlassian.com/projects/CONFSERVER>

⁶<https://jira.atlassian.com/projects/JRASERVER>

⁷<https://jira.mongodb.org/projects/SERVER>

⁸<https://jira.mongodb.org/projects/EVG>

⁹<https://figshare.com/s/3f402ee60f46ea1fe99c>

Fig. 2. Empirical Evaluation Steps

B. 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. However, as some issues are marked as enhancements in the *Severity* field, we filtered them out. In Jira, we only included the issues whose type is bug or defect. The total number of bugs and issues are shown in Table II.

C. 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 2.

Based on the generated matrix, we analyze the density of the smells for each project. Table III 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 Section IV.

A visual demonstration of Table III is given in Figure 3. 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

TABLE III
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	25020	821	0	2671	17	0	66952	2832	0	20407	1202	0	22504	89	0	12655	765	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	23546	2295	0	2032	656	0	67339	2445	0	18725	2884	0	16550	6043	0	13361	59	0
Reopen Ping Pong	25213	628	0	2559	129	0	68116	1668	0	21309	300	0	21915	678	0	13420	0	0
Resolver is not Assignee	6893	3498	15450	491	75	2122	18662	3342	47780	4163	2055	15391	8616	1414	12563	88	123	13209
Unassigned	6225	2665	16951	1316	6	1366	21110	16696	31978	5759	1975	13875	12733	351	9509	188	7656	5576

Fig. 3. Heatmap of Smell Ratio

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 system. Therefore, we mark those smells as not applicable to those projects and the corresponding cells in Figure 3 are left blank. We also evaluated how the number of BT smells changes over time. Figure 4 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. While searching the mail lists, we found out that Wireshark developers adapted Git and Gerrit tools in December 2013¹⁰. Links between commits and bugs are connected after the adaptation of those tools.

Another decrease in the number of BT process smells can be observed in Confluence Server & Data Center project. Figure 6

Fig. 4. Smell Ratio Change of All Projects by Year

shows how the number of *Missing Severity* smells decreases over time. After Atlassian added a new severity field to their projects in August 2016¹¹, 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 belongs 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.

D. Comparison of Bug Tracking Tools

The results of the empirical analysis are given in Table III 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

¹⁰<https://www.wireshark.org/lists/wireshark-dev/201312/msg00217.html>

¹¹<https://www.atlassian.com/blog/announcements/realigning-priority-categorization-public-bug-repository>

Fig. 5. Smell Ratio Change of Wireshark Project

Fig. 6. Smell Ratio Change of Confluence Server & Data Center Project

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%. The *closed-reopen ping pong* does not exist for one of the Bugzilla projects (*Wireshark*). 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

Fig. 7. Percentage of Bug Affected by Different No. of Smells

avoided during the BT process as it increases maintenance costs [49].

Figure 7 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 7 implies that the majority of bugs have one or two smells.

E. Time-Based Analysis

We also demonstrated how the number of BT process smells changes over time. Figure 4 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 III. Consequently, it is observed that over the years, best practices for the BT process have been adopted. Figure 4 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.

VI. THREATS TO VALIDITY

Although our analysis provides some useful and interesting results, there are a number of risks to our analysis's validity that must be examined. There are three 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 leave 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 field, 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 few assumptions on the configurable parameters and definitions (e.g. ignored bugs time duration is 6 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 are planning to survey experienced developers for expert opinions about the thresholds and definitions of our proposed BT process smells.

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 code¹².

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.

VII. CONCLUSION AND FUTURE WORK

In this study, 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 can summarize the main contributions of our study as follows:

- Proposed a novel taxonomy of BT process smells (Table I), based on a multivocal literature review.
- Performed an empirical analysis 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.

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, we are planning to expand our taxonomy with potential root causes and perform a similar empirical evaluation on a more diverse set of BT tools such as GitHub Issues.

¹²<https://figshare.com/s/3f402ee60f46ea1fe99c> and <https://figshare.com/s/f8ce1820d9a371a73071>

To observe the perceived importance of each BT process smell, we are also planning to survey among experienced developers working in the industry.

REFERENCES

- [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: ACM Press, 2008, p. 47. [Online]. Available: <http://portal.acm.org/citation.cfm?doid=1454247.1454265>
- [2] T. Zimmermann, R. Premraj, J. Sillito, and S. Breu, "Improving bug tracking systems," in *2009 31st International Conference on Software Engineering - Companion Volume*. IEEE, 2009, pp. 247–250. [Online]. Available: <http://ieeexplore.ieee.org/document/5070993/>
- [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, no. 5, pp. 618–643, sep 2010. [Online]. Available: <http://ieeexplore.ieee.org/document/5487527/>
- [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)*. IEEE, feb 2014, pp. 174–183. [Online]. Available: <http://ieeexplore.ieee.org/document/6747167/>
- [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: ACM, may 2014, pp. 658–661. [Online]. Available: <https://dl.acm.org/doi/10.1145/2591062.2591080>
- [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: ACM Press, 2014, pp. 1–10. [Online]. Available: <http://dl.acm.org/citation.cfm?doid=2590748.2590749>
- [7] 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: ACM Press, 2014, pp. 122–131. [Online]. Available: <http://dl.acm.org/citation.cfm?doid=2597073.2597081>
- [8] 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*, 2006, pp. 39–48.
- [9] 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*. IEEE, 2007, pp. 113–120.
- [10] P. Knab, M. Pinzger, and H. C. Gall, "Visual patterns in issue tracking aata," in *International Conference on Software Process*. Springer, 2010, pp. 222–233.
- [11] 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)*. IEEE, 2006, pp. 127–134.
- [12] P. Hooimeijer and W. Weimer, "Modeling bug report quality," in *Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering*, 2007, pp. 34–43.
- [13] 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)*. IEEE, may 2017, pp. 499–500. [Online]. Available: <http://ieeexplore.ieee.org/document/7965402/>
- [14] 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. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0950584918301939>
- [15] S. Keele *et al.*, "Guidelines for performing systematic literature reviews in software engineering," Citeseer, Tech. Rep., 2007.
- [16] R. Glass and T. DeMarco, *Software Creativity 2.0*, ser. Online access: EBSCO Computers & Applied Sciences Complete. Developer.* Books, 2006. [Online]. Available: <https://books.google.com.tr/books?id=DozsD0zx5wC>

- [17] M. Ivarsson and T. Gorschek, "A method for evaluating rigor and industrial relevance of technology evaluations," *Empirical Software Engineering*, vol. 16, no. 3, pp. 365–395, jun 2011.
- [18] 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, 2002, pp. 1–21. [Online]. Available: <https://www.ics.uci.edu/~wscacchi/Software-Process/Readings/Mozilla-study.pdf>
- [19] 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: ACM Press, 2011, pp. 1–7. [Online]. Available: <http://dl.acm.org/citation.cfm?doid=2020390.2020402>
- [20] 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*, ser. ESEC/FSE '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 111–120. [Online]. Available: <https://doi.org/10.1145/1595696.1595715>
- [21] S. Hill, "How to write a good bug report?" 2015, accessed: 2021-03-01. [Online]. Available: <https://leantesting.com/write-good-bug-report/>
- [22] J. Bridges, "8 steps for better issue management," 2019, accessed: 2021-04-25. [Online]. Available: <https://www.projectmanager.com/training/managing-project-issues/>
- [23] 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: ACM Press, 2006, p. 361. [Online]. Available: <http://portal.acm.org/citation.cfm?doid=1134285.1134336>
- [24] 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: ACM Press, 2010, p. 97. [Online]. Available: <http://portal.acm.org/citation.cfm?doid=1882291.1882308>
- [25] 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*. IEEE, 2013, pp. 89–98.
- [26] M. Ricklin, "Defect tracking best practices," 2009, accessed: 2021-03-01. [Online]. Available: <https://www.stickyminds.com/article/defect-tracking-best-practices-0>
- [27] 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: ACM Press, 2016, pp. 43–52. [Online]. Available: <http://dl.acm.org/citation.cfm?doid=2973839.2973844>
- [28] 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, no. 2083, p. 20160112, dec 2016. [Online]. Available: <https://royalsocietypublishing.org/doi/10.1098/rsta.2016.0112>
- [29] L. Jonsson, "Increasing anomaly handling efficiency in large organizations using applied machine learning," in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, may 2013, pp. 1361–1364. [Online]. Available: <http://ieeexplore.ieee.org/document/6606717/>
- [30] 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*. IEEE, apr 2012, pp. 437–446. [Online]. Available: <http://ieeexplore.ieee.org/document/6200136/>
- [31] 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*. IEEE, mar 2013, pp. 70–73. [Online]. Available: <http://ieeexplore.ieee.org/document/6569718/>
- [32] S. Admin, "10 Tips of writing effective bug reports," 2014, accessed: 2021-03-01. [Online]. Available: <https://www.softwaretestingclass.com/10-tips-of-writing-efficient-defect-report/>
- [33] M. Mitrev, "The anatomy of a good bug report," 2020, accessed: 2021-03-01. [Online]. Available: <https://devrix.com/tutorial/good-bug-report/>
- [34] 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.
- [35] 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, no. 3, p. e1821, mar 2017. [Online]. Available: <http://doi.wiley.com/10.1002/smr.1821>
- [36] B. Kucuk and E. Tuzun, "Characterizing duplicate bugs: An empirical analysis," in *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2021, pp. 661–668.
- [37] N. Bettenburg, R. Premraj, T. Zimmermann, and Sunghun Kim, "Duplicate bug reports considered harmful ... really?" in *2008 IEEE International Conference on Software Maintenance*. IEEE, 2008, pp. 337–345. [Online]. Available: <http://ieeexplore.ieee.org/document/4658082/>
- [38] E. Dmytriev, "Best practices for effective defect tracking," 2016, accessed: 2021-03-01. [Online]. Available: <https://www.linkedin.com/pulse/best-practices-effective-defect-tracking-eugene-dmytriev-l-i-o-n>
- [39] 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: ACM, jun 2020, pp. 17–20. [Online]. Available: <https://dl.acm.org/doi/10.1145/3377816.3381738>
- [40] 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. [Online]. Available: <https://linkinghub.elsevier.com/retrieve/pii/S0164121216000765>
- [41] T. Menzies and A. Marcus, "Automated severity assessment of software defect reports," in *2008 IEEE International Conference on Software Maintenance*. IEEE, sep 2008, pp. 346–355. [Online]. Available: <http://ieeexplore.ieee.org/document/4658083/>
- [42] LuminosLabs, "Writing good bug report," 2020, accessed: 2021-03-01. [Online]. Available: <https://www.luminoslabs.com/insights/writing-good-bug-report/>
- [43] 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)*. IEEE, may 2007, pp. 499–510. [Online]. Available: <http://ieeexplore.ieee.org/document/4222611/>
- [44] X. Xia, D. Lo, E. Shihab, X. Wang, and B. Zhou, "Automatic, high accuracy prediction of reopened bugs," *Automated Software Engineering*, vol. 22, no. 1, pp. 75–109, mar 2015. [Online]. Available: <http://link.springer.com/10.1007/s10515-014-0162-2>
- [45] 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, sep 2018, pp. 44–58. [Online]. Available: <https://linkinghub.elsevier.com/retrieve/pii/S0164121218300530>
- [46] 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*. IEEE, oct 2012, pp. 225–234. [Online]. Available: <http://ieeexplore.ieee.org/document/6385118/>
- [47] 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*. IEEE, oct 2010, pp. 249–258. [Online]. Available: <http://ieeexplore.ieee.org/document/5645566/>
- [48] AnalyseIT247, "Reopening issues after they have been resolved or closed," 2016, accessed: 2021-03-01. [Online]. Available: <https://www.analyseit247.com/reopening-issues/>
- [49] 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, no. 5, pp. 741–773, may 2016. [Online]. Available: <https://academic.oup.com/comjnl/article-lookup/doi/10.1093/comjnl/bxv114>
- [50] 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 Proceedings*. New York, NY, USA: ACM, may 2018, pp. 1–4. [Online]. Available: <https://dl.acm.org/doi/10.1145/3183440.3183475>