Revisiting the Impact of Antipatterns on Fault-Proneness: A Differentiated Replication

Aurel Ikama[1], Vincent Du[1], Philippe Belias[1], Biruk Asmare Muse[1], Foutse Khomh[1]
[1] *Dep. of Computer and Software Engineering, Polytechnique Montréal, Montréal*

***Abstract*—Anti-patterns manifesting on software code through code smells have been investigated in terms of their prevalence, detection, refactoring, and impact on software quality attributes. In particular, leveraging heuristics to identify fault-fixing commits, Khomh et al. have found that anti-patterns and code smells have an impact on the fault-proneness of a software system. Similarly, Saboury et al. have found a relationship between anti-patterns occurrences and fault-proneness, using the SZZ heuristic to identify fault-fixing commits and fault-inducing changes. However, recent studies question the accuracy of this popular SZZ heuristic, and thus the validity of empirical studies that leverage it.**

This motivated us to conduct a differentiated replication of the work by Khomh et al.. We particularly focused on the impact of anti-patterns on fault-proneness as it is the only dependent variable that may be affected by noise in the collected faults data. In our differentiated replication study, (1) we expanded the number of subject systems from 5 to 38, (2) utilized a manually validated dataset of bug-fixing commits from the work of Herbold et al., and (3) answered research questions from Khomh et al., that are related to the relationship between anti-patterns occurrences and fault-proneness. (4) We added an additional research question to investigate if combining results from several SZZ approaches could help reduce the impact of noise. Our findings show that the impact of the noise generated by the SZZ algorithm is negligible for the studied subject systems; meaning that the reported relation observed on noisy data still holds on the clean data. However, we also observed that combining results from several SZZ approaches do not reduce this noise, quite the contrary.

***Keywords*—** Anti-patterns, Fault-proneness, SZZ algorithm

I. Introduction

Anti-patterns are poor solutions to recurring design problems in the context of software engineering. Anti-patterns manifest themselves in the source code through code smells [1]. Ever since Brown et al. [2] introduced their catalog of 40 anti-patterns, there have been many studies on the detection [3], [4], [5], [6], [7], refactorings [8], [9], prevalence [10], and evolution [11], [12] of anti-patterns and code smells, and their impacts on maintenance effort [13]. Researchers have also examined the impact of these anti-patterns on performance [14], [15], fault proneness [16], [17], [18] and change proneness [17].

Studies that examined the relationship between anti-patterns or code smells and faults often follow different approaches to identify bug fix commits and the corresponding bug introducing commits. The state-of-the-art algorithm to identify bug fixes and bug introducing commits is the SZZ algorithm, which was introduced by Śliwerski et al. [19]. The algorithm works in two phases. In the first phase, bug fix commits are identified using regular expressions that search the issue/bug number from the commit messages and link that number to issue tracking data. Then the modified lines of the identified bug-fix commit are extracted. In the next phase, SZZ uses the git blame command to obtain a set of commits that previously modified the extracted lines. The bug-inducing commits are identified by examining the commits that touched the extracted lines in the previous versions. Commits that previously modified the target lines are considered as potential bug-inducing commits. Then, among these candidates, commits that were created before the target bug was reported are removed from the list of candidates. Recent studies question the accuracy of the SZZ algorithm and the validity of empirical studies involving SZZ [20], [21], [22]. Most recently, Herbold et al. [22] found that only half of the bug fixing commits identified by SZZ are actually bug-fixing commits. Furthermore, they found that SZZ misses 20% of bug-fix commits. Such noise in the data could affect the conclusions of studies that rely on them. Given the availability of manually validated data set of bug-fix and bug-inducing commits from Herbold et al., we were motivated to assess the extent to which previous studies that relied on heuristics could be affected by noises contained in their dataset. In particular, We conducted a differentiated replication study of the impact of anti-patterns in change and fault proneness by Khomh et al. [17].

Khomh et al. conducted an empirical study to investigate the impact of anti-patterns on the change- and fault-proneness of source code classes. The authors detected 13 anti-patterns from Brown [2] and Fowler [1] in 54 releases of ArgoUML, Eclipse, Mylyn, Rhino, and analyzed the change- and fault -proneness of the detected anti-patterns. They used the DECOR [3] detection tool to extract the anti-patterns from the subject systems and relied on bug fixing changes information from Bugzilla to investigate fault-proneness. They answered six research questions on the relationship between anti-patterns, kind of anti-patterns, and change-proneness and fault-proneness. They also considered class sizes as a co-founding factor. Their results show that classes involved in anti-patterns are more change- and fault-prone than classes that are not involved in anti-patterns. Among the studied anti-patterns, MessageChain had a significant impact on change-proneness in all subject systems. The authors also found that the relation between anti-patterns and fault-proneness is not as strong as the relation between anti-patterns and change-proneness. While the authors used manually validated bug fix changes, they mentioned that their dataset could still have some miss-classified bug fix issues. Hence, we hypothesize that the presence of noise could affect the conclusions of the study on fault-proneness. In addition to that, the study only considered 5 subject systems which affect the generalization of their findings. The results of this replication study contribute to improving the generalizability of the reported impact of anti-patterns on fault-proneness. We hope that this replication study will highlight the need for re-evaluating the findings of all the previous studies that leveraged heuristics such as SZZ as part of their methodologies, allowing for the elaboration of more robust results.

The remainder of the paper is organized as follows. Section II describes our study method. We report our case study results in Section III followed by discussion of the findings in Section IV. Next, we describe the threats to validity in Section V and related work in Section VI. Finally, we conclude the paper in Section VII.

II. Study Design

The objective of this study is to investigate the extent to which the noise introduced due to the limitations of SZZ impacts fault detection and to conduct a differentiated replication of Khomh et al.'s work

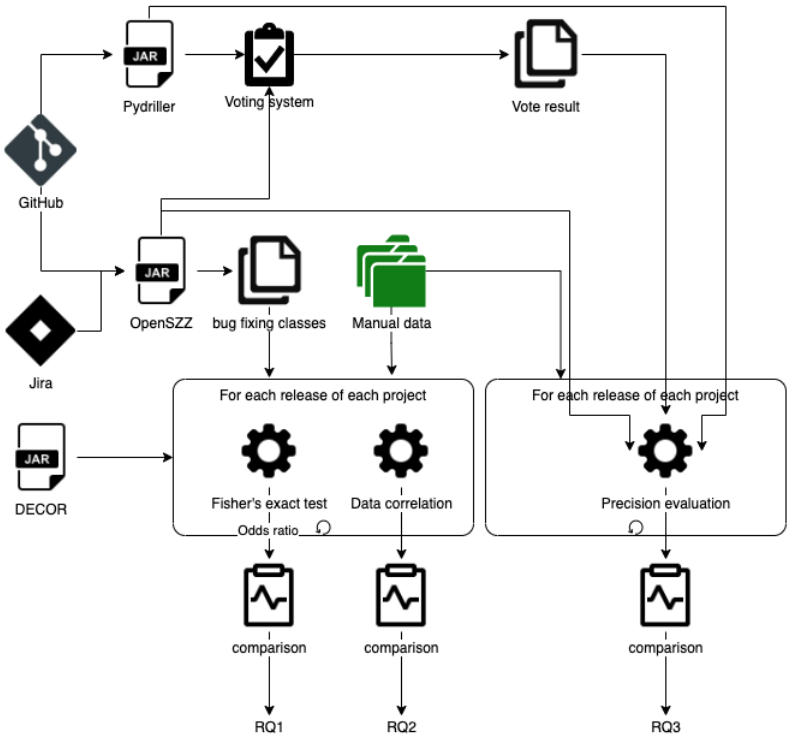

Figure 1: Overview of the study method

on the relationship between antipatterns and fault-proneness using at least 2 releases of 38 subject systems (listed in Table II). More particularly We investigated the following three research questions.

RQ1:What is the impact of using SZZ generated bug fixing changes on the relation between antipatterns and fault-proneness?

In this research question, we investigate how the relationship between antipatterns and fault proneness is affected by using the SZZ generated bug fix commits compared to using the dataset of Herbold et al. [22]. This research question is similar to RQ3 of Khomh et al.'s paper [17].

RQ2: What is the impact of using SZZ generated bug fixing changes on the relationship between specific types of antipatterns and fault-proneness?

In this research question, we investigate the relationship of specific antipatterns with fault-proneness using the SZZ generated bug fix commits and using the dataset of Herbold et al. [22]. This research question is similar to RQ4 in Khomh et al.'s paper [17].

RQ3: Does combining the results of multiple SZZ detection approaches using majority vote reduces the gap with the ground truth?

In this research question, we investigate if combining different implementations of SZZ improves bug fix commit identification and makes the result closer to using the bug fix commits obtained from Herbold's manually curated dataset.

A. Data collection

We used 38 subject systems similar to the work of Herbold et al. [23] as we rely on their manually validated dataset of bug-fixing commits. Note that in this paper the reference data are those collected manually. Herbold et al. picked the subject systems from the Apache Software Foundation [24] that have reached a certain level of maturity in order to be considered as a top-level project. A subject system is considered as mature if the system uses Git as version control system, uses Java as main language, uses Jira as issues tracking system, etc. The whole criteria for the inclusion of projects is shown in Table I.

B. Data Processing

In this sub-section, we describe the details of the data collection and analysis approach followed to answer our different research questions. This approach is depicted in Figure 1. In the following, we elaborate on each data processing step.

1) *SZZ bug detection:* We use the following two SZZ tools to generate bug-fixing changes.
OpenSZZ (Precision = 83%) [25] is an open-source implementation of the SZZ algorithm. The OpenSZZ executable takes the project GitHub Url and Jira Url as input and finds all classes that were involved in at least one bug fixing commit as output. We run OpenSZZ on each release of each project and saved the result of each run in a JSON file.
Pydriller [26] is a framework written in python with which one can easily extract any type of information from any GitHub directory. Its ***get_commits_last_modified_lines*** function uses the SZZ algorithm to determine the set of commits containing a bug from a certain commit passed to it as a parameter. We use Pydriller on each project to detect classes that were in at least one bug-inducing commit and saved them in a list.

2) *Antipattern detection:* With **DECOR** (Precision=60.5%, Recall=100%) [3] we were able to detect the antipatterns present in the classes of the different versions of each project. We then extracted the different antipatterns with the associated classes.

3) *Impact evaluation:* To evaluate the impact of the noise introduced by SZZ on the relationship between antipatterns and

Table I: Criteria for the inclusion of projects use in [23] Table 3

Criterion	Rational
Uses Git	Most projects either already use a Git repository, or provide a Git mirror of an SVN repository
Java as main language	Our static analysis only works for Java code
Uses Jira	The Jira of the Apache Software Foundation is the main resource for tracking issues of most Apache projects
At least two years old	Project as a sufficient development history
Not in incubator stage	Project has been fully accepted by the Apache Software Foundation
>100 Issues in Jira	Project is mature and actively uses Jira
>1000 Commits	Project has a sufficient development history
>100 Files	Project should have a reasonable size
Activity since 2018-01-01	Project is still active in both Jira and Git

fault-proneness, we analyzed the bug fixing changes generated by OpenSZZ, then those collected manually on the studied systems. For the analyses, we applied the same methodology as Khomh et al. [17] (**Fisher's exact test** and **calculation of Odds Ratio (OR)**) on a set of antipatterns generated by DECOR. The comparison of the two analyses allows us to answer **RQ1**.
Note that we execute Fisher's exact test twice with different inputs: once with smelly classes generated by DECOR and faulty classes generated by OpenSZZ, then another time with the same smelly classes generated by DECOR and faulty classes that are manually validated. So the Odds Ratios generated by the two Fisher's exact tests are compared and analyzed to assess the impact of using automatically generated faults (SZZ) on the relation between antipattern and fault-proneness.

4) *Correlation analysis:* From the intersection of the list of classes containing the different bugs generated by OpenSZZ, and the list of classes associated with the antipatterns generated by DECOR, we computed the logistic regression model [17].
We then counted for each antipattern, the number of times that all the analyzed versions have a p-value higher than $t = 75\%$. This is the value at which the p-value is significant in the logistic regression. This means that there is a correlation between the antipattern and the bugs. We count in each analyzed version the number of significant p-values. As stated by Khomh et al. in [17], *"If these classes are more likely to change in more than t releases, then we say that this antipattern has a significant impact on increasing the change-proneness"*. Similarly, for fault-proneness, if the classes are more likely to experience a fault in more than t releases, we conclude that the antipattern has a significant impact on increasing the fault-proneness. Finally, we replicate the process by replacing OpenSZZ with the manually validated data. The comparison of the two results allows us to answer **RQ2**.

5) *Voting system and precision evaluation:* We ran both SZZ tools (OpenSZZ and Pydriller) on each of the projects and collected information about the classes containing the bugs (provided by each tool). Then, the voting process goes as follows, a class contains a bug if and only if both tools detected that class when we ran them separately. Finally, we evaluate the detection accuracy for the combination of tools (resulting from the voting) and the detection accuracy for each tool separately and compare them with the manual detection, to answer **RQ3**.

C. Replication Package

We made our dataset, data collection, and data analysis scripts publicly available at [27], to allow for replications and extensions of our work. The manual data published by Herbold et al. are available at [28].

III. Case Study Results

RQ1: What is the impact of using SZZ generated bug fixing changes on the relation between antipatterns and fault-proneness?

The odds ratios generated by Fisher's exact test for each release of the 3 first projects are shown in Table IV (see replication package for all results [27]). Each row shows, for each system, a release number and the odds ratio of classes participating in at least one antipattern in that release to exhibit at least one fault-fixing change before the next release. Releases where Fisher's exact test result is not statistically significant (p-value > 0.05) are annotated with *, and releases, where Fisher's exact test didn't output any results (due to insufficient input data, i.e., no faulty classes in that release), are represented by a hyphen "-".
The results show that, in all releases where the result is statistically significant, the odds ratios generated with manually validated faults and that generated with SZZ detected faults followed the same trend (all superior to 1 and in the same order of magnitude), which means that both results indicate a significant difference of proportions between fault-proneness of classes participating and not participating in antipatterns. The odd ratio values are between 2.14 and 22.99. We, therefore, conclude that using the SZZ generated dataset yields results similar to those obtained on manually validated bug data. Overall, the findings confirm previous results by Khomh et al. regarding the relationship between antipatterns and fault-proneness.

RQ2: What is the impact of using SZZ generated bug fixing changes on the relationship between specific types of antipatterns and fault-proneness?

Figures 2 and 3 present respectively the logistic regression results for the relationships between fault-proneness and antipattern types for the manual data and the data from OpenSZZ.

We have represented the results for each data type as in the paper [17]; "a cell in the table indicates the number (and percentage) of versions, for a given system, in which class participation in a given antipattern is significantly correlated with fault-proneness".

We observe that for both types of data (manual and OpenSZZ), the classes participating in the antipatterns LongMethod, ComplexClass, and to a lesser extent LongParameterList are more fault-prone than the other classes.

We note that we also find more antipatterns linked to faults in the data obtained from OpenSZZ than in the manual data, this could be explained by the noise coming from the automatically collected data. Indeed, the consensus applied to label the faults in the manual data allows eliminating the false positives.

However, this noise has no impact on the conclusion on this question, which is to say that despite data collected manually or automatically, there is a relationship between antipatterns and fault-proneness but this relationship differs according to the systems and versions and does not concern all antipatterns.

Finally we find the same specific types namely *ComplexClass*, *LongMethod* and *AntiSingleton* in the Khomh et al. [17] results and in the manual data. However, the antipatterns *LongParameterList*, *ClassDataShouldBePrivate*, and *RefusedParentBequest* are correlated

Table II: Apache projects and releases used for our empirical study

project	releases
ant-ivy	1.4.1, 2.0.0, 2.1.0, 2.2.0, 2.3.0, 2.4.0
archiva	1.0, 1.1, 1.2, 1.3, 2.0.0, 2.1.0, 2.2.0
calcite	1.0.0, 1.1.0, 1.2.0, 1.3.0, 1.4.0, 1.5.0, 1.6.0, 1.7.0, 1.8.0, 1.9.0, 1.10.0, 1.11.0, 1.12.0, 1.13.0, 1.14.0, 1.15.0
cayenne	3.0.0, 3.1.0
commons-bcel	6.0, 6.1, 6.2
commons-beanutils	1.7.0, 1.8.0, 1.9.0
commons-codec	1.2, 1.3, 1.5, 1.6, 1.7, 1.8, 1.9, 1.10
commons-collections	1.0, 2.0, 2.1, 3.0, 3.1, 3.2, 3.3, 4.0, 4.1
commons-compress	1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9, 1.10, 1.11, 1.12, 1.13, 1.14, 1.15, 1.16
commons-configuration	1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9, 1.10, 2.2
commons-dbcp	1.0, 1.1, 1.2, 1.3, 1.4, 2.0, 2.1, 2.2.0, 2.3.0, 2.4.0, 2.5.0
commons-digester	1.4, 1.5, 1.6, 1.7, 1.8, 3.0, 3.1, 3.2
commons-io	1.0, 1.1, 1.2, 1.3, 1.4, 2.0, 2.1, 2.2, 2.3, 2.4, 2.5
commons-jcs	1.0, 1.1, 1.3, 2.0, 2.1, 2.2
commons-jexl	2.0, 2.1
commons-lang	2.0, 2.1, 2.2, 2.3, 2.4, 2.5, 2.6, 3.0, 3.1, 3.2, 3.3, 3.4, 3.5, 3.6, 3.7
commons-math	1.0, 1.1, 1.2, 2.0, 2.1, 2.2, 3.0, 3.1, 3.2, 3.3, 3.4, 3.5
commons-net	1.0.0, 1.1.0, 1.2.0, 1.3.0, 1.4.0, 2.0, 2.1, 2.2, 3.0, 3.1, 3.2, 3.3, 3.4, 3.5, 3.6
commons-scxml	0.5, 0.6, 0.7, 0.8, 0.9
commons-validator	1.1.0, 1.2.0, 1.3.0, 1.4.0, 1.5.0, 1.6.0
commons-vfs	1.0, 2.0, 2.1, 2.2
deltaspike	0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 1.0.0, 1.1.0, 1.2.0, 1.3.0, 1.4.0, 1.5.0, 1.6.0, 1.7.0, 1.8.0
eagle	0.3.0, 0.4.0
giraph	1.0.0, 1.1.0
gora	0.1, 0.2, 0.3, 0.4, 0.5, 0.6
jspwiki	1.4.0, 1.5.0, 1.6.0, 1.7.0, 1.8.0, 2.0.36, 2.2.19, 2.4.56, 2.6.0, 2.8.0, 2.9.0, 2.10.0
knox	0.3.0, 0.4.0, 0.5.0, 0.6.0, 0.7.0, 0.8.0, 0.9.0, 0.10.0, 0.11.0, 0.12.0, 0.13.0, 0.14.0, 1.0.0
kylin	0.6.1, 0.7.1, 1.0, 1.1, 1.2, 1.3, 1.5.0, 1.6.0, 2.0.0, 2.1.0, 2.2.0
lens	2.6.0, 2.7.0
mahout	0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 0.10.0, 0.11.0, 0.12.0
manifoldcf	0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9, 1.10, 2.0, 2.1, 2.2, 2.3, 2.4, 2.5, 2.6, 2.7, 2.8, 2.9, 2.10
nutch	0.7, 0.8, 0.9, 1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9, 1.10, 1.11, 1.12, 1.13, 1.14, 2.0, 2.1, 2.2, 2.3
opennlp	1.7.0, 1.8.0
parquet-mr	1.0.0, 1.1.0, 1.2.0, 1.3.0, 1.4.0, 1.5.0, 1.6.0, 1.7.0, 1.8.0, 1.9.0
santuario-java	1.4.5, 1.5.9, 2.0.0, 2.1.0
systemml	0.9, 0.10, 0.11, 0.12, 0.13, 0.14, 0.15, 1.0.0, 1.1.0
tika	0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 0.10, 1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9, 1.10, 1.11, 1.12, 1.13, 1.14, 1.15, 1.16, 1.17
wss4j	1.5.0, 1.6.0, 2.0.0, 2.1.0

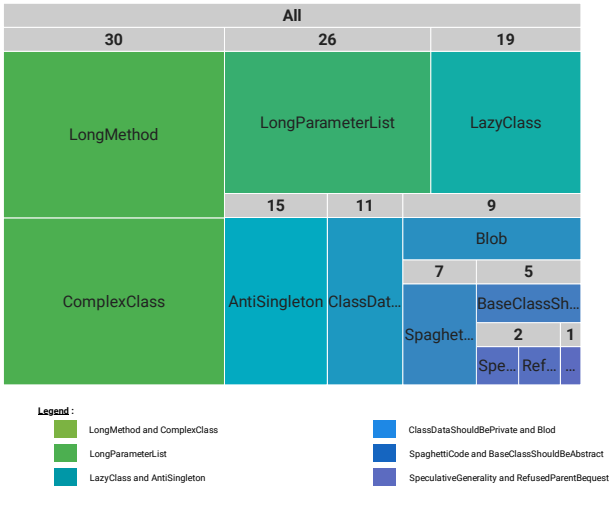

Figure 2: Results of logistic regression for relationships between fault-proneness and antipattern types for data from OpenSZZ

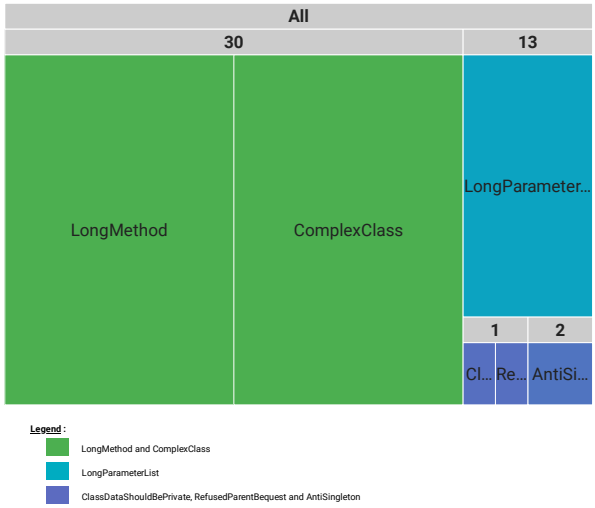

Figure 3: Results of logistic regression for relationships between fault-proneness and antipattern types for the manual data

Table III: Percentage of detection using each tool separately and by combining them

Project name	Combination of tools	Pydriller detection	Openszz detection
ant-ivy	83.33%	100.00%	83.33%
archiva	29.23%	96.92%	29.23%
calcite	86.11%	99.69%	86.11%
cayenne	50.85%	98.31%	50.85%
commons-bcel	40.00%	100.00%	40.00%
commons-beanutils	81.82%	100.00%	81.82%
commons-codec	50.00%	100.00%	50.00%
commons-collections	38.46%	100.00%	38.46%
commons-compress	53.06%	100.00%	53.06%
commons-configuration	56.36%	100.00%	56.36%
commons-dbcp	40.63%	100.00%	40.63%
commons-digester	8.33%	100.00%	8.33%
commons-io	60.87%	100.00%	60.87%
commons-jcs	54.17%	100.00%	54.17%
commons-jexl	16.67%	100.00%	16.67%
commons-lang	56.06%	84.85%	56.06%
commons-math	27.12%	100.00%	27.12%
commons-net	55.77%	100.00%	55.77%
commons-scxml	52.94%	100.00%	52.94%
commons-validator	42.86%	100.00%	42.86%
commons-vfs	35.71%	98.21%	35.71%
deltaspike	66.67%	100.00%	66.67%
eagle	35.00%	100.00%	35.00%
giraph	29.73%	100.00%	29.73%
gora	59.09%	100.00%	59.09%
jspwiki	11.54%	100.00%	11.54%
knox	51.77%	97.87%	51.77%
kylin	0.00%	99.43%	0.00%
lens	92.31%	100.00%	92.31%
mahout	50.00%	99.55%	50.00%
manifoldcf	83.69%	100.00%	83.69%
nutch	79.31%	94.83%	82.76%
opennlp	42.86%	100.00%	42.86%
parquet-mr	21.69%	100.00%	21.69%
santuario-java	51.35%	100.00%	51.35%
systemds	0.00%	100.00%	0.00%
tika	62.20%	97.64%	62.20%
ws-wss4j	31.58%	100.00%	31.58%

Table IV: Odds ratio comparison for projects ant-ivy, archiva, and calcite

Fault proneness								
ant-ivy			archiva			calcite		
Releases	Odds ratios (manual)	Odds ratios (szz)	Releases	Odds ratios (manual)	Odds ratios (szz)	Releases	Odds ratios (manual)	Odds ratios (szz)
1.4.1	3.52*	13.1	1	3.99*	14.87	1.0.0	3.17	1.05*
2.0.0	9.25	8.83	1.1	11.66	-	1.1.0	4.38	3.8
2.1.0	11.31	17.11	1.2	7.48	-	1.2.0	4.84	3.45
2.2.0	10.74	7.78	1.3	5.32	3.7	1.3.0	4.02	4.12
2.3.0	6.6	13.88	2.0.0	-	-	1.4.0	3.09	3.78
2.4.0	15.5	7.68	2.1.0	-	4.49	1.5.0	3.93	3.94
			2.2.0	-	6.02	1.6.0	3.74	3.37
						1.7.0	4.22	2.12
						1.8.0	6.2	3.12
						1.9.0	5.57	3.18
						1.10.0	5.92	4.4
						1.11.0	6.18	2.36
						1.12.0	7.02	2.14
						1.13.0	7.45	3.56
						1.14.0	7.43	5.18
						1.15.0	7.85	3.63

with defects in the analysis with the manual data but are not reported by Khomh et al.

RQ3: Does combining the results of multiple SZZ detection approaches using majority vote reduces the gap with the ground truth?

As discussed in Section II, to answer this question we first ran each tool on the entirety of each project. Secondly, we used the voting system explained in section II-A. Thirdly, we compare the detection result for each tool separately with the detection made by the combination of the tools to see how the noise evolves. To evaluate the impact of noise when leveraging the two tools simultaneously during fault identification, we used the manually validated dataset as our ground truth. For each project, we combined the manual detection of each release we decided to study in one file for code purposes. For example for the ant-ivy project, we combined all the files containing bug fixes in one file from version 1.4.1 to version 2.4.0. Next, we wrote a script so that for each class in that file, it will go search if that class was detected by OpenSzz tool, PyDriller, and the combination of the two and calculate the percentage of the precision detection.
Table III shows the result of our experiment. As we can see the combination of tools is less efficient than the tools used separately. The reason for that is the quality of the tools that have been combined. We can see that compared to PyDriller, OpenSzz is not very efficient. So by combining those two we see that instead of reducing the noise we are adding it. Indeed, we observe that the accuracy of detection with the combination of tools is lower than that with each tool separately. For example, we can see that Pydriller alone has a better detection accuracy compared to the one resulting from the combination of the tools.

IV. Discussion

The findings of Khomh et al. [17] show that there is a relationship between antipatterns and fault-proneness using bug fixing commits obtained using heuristics. They go further by determining which kinds of anti-patterns are most related to fault-proneness. Our differentiated replication with a higher number of subject systems obtained from [22] confirmed their conclusions. Indeed, despite the noise in the SZZ data found in the study of Herbold et al. [23], we found our findings in alignment with the findings of Khomh et al. regarding the relationship between antipatterns and fault-proneness. It shows that the noise of SZZ generated bug fixing commits has a negligible impact on the analysis of the relationship between anti-patterns and fault-proneness. However, we did not, as in the replicated paper, study the types of changes involved in the relationship with antipatterns or the relationship between the presence of antipatterns and class size, because our objective was limited to determining whether the difference between the data collected from the SZZ algorithm and those collected manually impacts the analysis of the relationship between antipatterns and fault-proneness. We observe that the impact of the data discrepancy is insufficient to invalidate or contradict the results of a study on antipatterns using the SZZ algorithm. Moreover, in this study we added the information that a combination of SZZ tools does not reduce the gap with manual data, it rather increases it. In light of the work of Herbold et al. we contribute that the noise created by bug fixing commits in the use of SZZ is negligible in the analysis of the impact of the relationship between antipatterns and fault-proneness. Since combining results from multiple tools does not seem to mitigate the amount of noise, we recommend that more effort should be invested in the development of better bug-fix and bug-inducing commit detection tools. Future works should also consider experimenting with more advanced statistical data aggregation techniques. Overall, the impact of SZZ generated bug fixing commits noise seems to be negligible in the relation between anti-patterns and fault-proneness, however, the choice of the implementation of the SZZ algorithm could influence this relation. Hence, Other studies must be done in other contexts in order to further validate our findings. It will also be necessary to study other aspects of the impact of this noise such as refactoring.

V. Threats to validity

This section discusses threats to the validity of our replication study following the guidelines for case study research.

A. Threats to construct validity

The use of DECOR [3] for antipattern detection. The accuracy of DECOR is not perfect, hence it could miss some antipatterns and therefore affect our conclusions. However, this is the state of the art detection approach, with 100% recall [29], with concrete implementation and also deployed in similar papers (including the original study by Khomh et al.), examples ([30], [31], [32], [33], [34]). Another threat to construct validity concerns the accuracy of OpenSZZ and PyDriller. We have used the official implementations of these tools to avoid introducing accuracy issues beyond those already reported by the authors of the tools. Moreover, using these official implementations allows for consistent comparisons with previous studies that also relied on these implementations.

B. Threats to internal validity

We did not claim any causation as our analysis is on the relationship between antipatterns and fault-proneness. Hence, our study is not subjected to threats to internal validity. Our replication study reinforces the conclusion that antipatterns do impact the fault-proneness of classes and that certain kinds of antipatterns have a greater impact than others. However, our study cannot say anything about the reasons for classes to have antipatterns and, consequently, the reasons for faults to occur in these classes. We only empirically verified that classes with antipatterns are more fault-prone than others, thus confirming previous findings.

C. Threats to External validity

These threats concern the possibility to generalize our results. One of the conclusions we made with our experiment is that the combination of SZZ tools in order to reduce the noise is not a good idea. However, the generalization of our findings needs to take into account two factors. The first factor is the quality of the tool used. As we saw PyDriller was far more effective than OpenSZZ so combining them using our vote system will only lead to accepting the detection of OpenSZZ. The second factor is the number of tools used. Indeed during our experiment, we only used two tools. One could envision using more than two tools and also experiment with more advanced statistical data aggregation techniques. Future work might consider replicating our study with more than two SZZ tools and more advanced aggregation techniques.

D. Threats to reliability validity

To minimize potential threats to reliability, we analyzed open-source projects available on GitHub and provide a replication package that contains our dataset and analysis scripts [27].

VI. Related Work

A. Replication studies in software engineering

Replication of empirical studies in software engineering has provided important contributions to the validation and generalization of original works. Cruz et al. [35] showed that the trend of published replication studies in software engineering in general and empirical studies in software engineering is increasing. Gómez et al. [36] conducted a literature review of software engineering replication studies and proposed a classification of replication studies into literal, conceptual, and operational. The aim of literal replication is to have as exact experiment settings as possible with the primary paper. The aim of the conceptual replication is to take the findings of the primary paper and verify them in completely different experimental

settings. The core findings of the original paper are evaluated in the new experimental setting. The aim of operational (differentiated) replication is to vary some experiment dimensions like increasing the number of the subject systems, overcoming some threats to validity introduced by the primary study, and deploying improved experiment data sources. There are many studies that perform differentiated or operational replications (e.g., [37], [38], [39], [40], [41]). Rahman et al. conducted a differentiated replication study about security smells in Ansible and Chef scripts. The primary paper [42] was done by the same authors but the experimental settings are modified in the replication study by changing the subject systems, introducing a new detection tool, and specification of smells.

Fucci et al. [39] conducted a differentiated replication study that proposes a technique to automatically distinguish between code comprehension, code review, and prose review tasks measuring the body signals of participants. The objective of the experiment is to use machine learning to identify what tasks developers are working on measuring brain-, hurt- and skin-related signals. The replicated study modified the original research questions as well as the experiment protocol. The authors improved the accuracy of the detection in the original paper.

Di Penta et al. [41] conducted a differentiated replication study improving the experimental design and generalizability of the primary paper by Bavota et al. [43]; investigating the relationship between refactoring and bugs. The authors extended the findings of the primary paper and also found that some of the bug-inducing commits reported by SZZ are not correct, i.e., the bugs were in the system before the reported bug-inducing commits. The above studies demonstrate the benefits of differentiated replication studies (like our study) on improving proposed techniques, generalization, and validation of findings.

B. Studies on the limitations of SZZ

The problem with SZZ has been studied by Herbold et al. [23]. Their study uncovered potential problems in different parts of the SZZ algorithm. In fact, from their conducted empirical study on 398 releases of 38 Apache projects, they've found that only half of the bug fixing commits determined by SZZ are bug fixing. If a six-month time frame is used in combination with SZZ to determine which bugs affect a release, one file is incorrectly labeled as defective for every file that is correctly labeled as defective. They concluded that problems with inaccurate defect labels are a severe threat to the validity of the state of the art of defect prediction. In our study, we want to assess the impact of inaccurate defect labels on another popular software engineering investigation, i.e., the relationship between antipatterns and fault-proneness.

C. Studies on the impact of Antipatterns

The impact of anti-patterns on software quality has been well investigated. Specifically, the impacts of anti-patterns on maintenance effort [13], on performance [14], [15], on fault proneness, [16], [17], [18] and on change proneness [17] are some example studies. Johannes et al. [16] show, for example, that files containing code smells participate in more faults than those without code smells in JavaScript programs. Zazworka et al. [18] find that design debt has a negative impact on software quality. Hecht et al. [14] reported the negative impact of Android code smell on performance, and Spadini et al. [44] showed that test smells increase the change- and fault-proneness of tests and thus the quality of the software.

The relation between antipatterns and fault-proneness has already been studied in a previous paper by Khomh et al. [17], where they analyzed the relationship between antipatterns and fault-fixing issues, and the influence of kinds of antipatterns on fault-proneness. To detect antipatterns, they've used DECOR (Defect dEtection for CORrection) which is capable of detecting 13 types of antipatterns.

Abidi et al. [31] performed an empirical study on 98 releases of JNI systems and investigated the prevalence and impacts of multi-language design smells on software quality. They reported that multi-language smells are prevalent in open-source projects and that they persist throughout the releases of the systems. They also reported that some kinds of smells are more prevalent than others. Their results suggest that multi-language smells can often be more associated with bugs than files without smells.

Muse et al. [30] performed an empirical study on the prevalence and impact of SQL design smells. They reported that SQL smells are prevalent and persist in the studied open-source projects and that they have a weak association with bugs.

Abbes et al. [34] investigated the impact of occurrences of design smells on developers' understandability of the code during program comprehension and maintenance tasks. They conducted three experiments to collect data about the performance of developers and study the impact of Blob and Spaghetti Code anti-patterns and their combinations. They conclude that the occurrence of one antipattern does not significantly impact comprehension while the combination of the two antipatterns negatively impacts program comprehension.

VII. Conclusion

In this paper, we conducted a differentiated replication of the work of Khomh et al. with the aim to examine the potential impact that noise contained in fault data obtained using heuristics can have on the reported relation between anti-patterns and fault-proneness. Our results provide empirical evidence that noise contained in bug fixing data obtained with SZZ has a negligible impact on the analysis of the relationship between antipatterns and fault-proneness. We performed Fisher's exact test and logistic regression using faults (generated using SZZ algorithm and retrieved from manually validated data) and antipatterns (generated using DECOR) in different releases of 38 Apache open source systems, then we compared the generated results to answer our research questions. We showed, through the first research question RQ1, that using the faults generated with the SZZ algorithm to assess the relation between antipatterns and fault-proneness doesn't impact the result observed with using manually validated faults. We also showed with RQ2 that the noise in the SZZ generated faults has no impact on the relationship between particular kinds of antipatterns and fault-proneness. We also studied with RQ3 the effect of merging data collected using different SZZ tools on reducing the gap between automatically generated faults and manually validated faults. We found that instead of reducing the noise, the voting merging technique actually increases the noise in fault detection. The results of this replication study contribute to improving the generalizability of the reported impact of anti-patterns on fault-proneness.

References

[1] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, "Refactoring: Improving the design of existing code, 1999," *Google Scholar Google Scholar Digital Library Digital Library*.

[2] W. H. Brown, R. C. Malveau, H. W. S. McCormick, and T. J. Mowbray, *AntiPatterns: refactoring software, architectures, and projects in crisis*. John Wiley & Sons, Inc., 1998.

[3] N. Moha, Y.-G. Gueheneuc, L. Duchien, and A.-F. Le Meur, "Decor: A method for the specification and detection of code and design smells," *IEEE Transactions on Software Engineering*, vol. 36, no. 1, pp. 20–36, 2009.

[4] S. Hassaine, F. Khomh, Y.-G. Guéhéneuc, and S. Hamel, "Ids: An immune-inspired approach for the detection of software design smells," in *2010 Seventh International Conference on the Quality of Information and Communications Technology*. IEEE, 2010, pp. 343–348.

[5] F. Khomh, S. Vaucher, Y.-G. Guéhéneuc, and H. Sahraoui, "Bdtex: A gqm-based bayesian approach for the detection of antipatterns," *Journal of Systems and Software*, vol. 84, no. 4, pp. 559–572, 2011.

[6] A. Maiga, N. Ali, N. Bhattacharya, A. Sabané, Y. Guéhéneuc, and E. Aimeur, "Smurf: A svm-based incremental anti-pattern detection approach," in *2012 19th Working Conference on Reverse Engineering*, 2012, pp. 466–475.

[7] F. Palomba, "Textual analysis for code smell detection," in *Proceedings of the 37th International Conference on Software Engineering-Volume 2*. IEEE Press, 2015, pp. 769–771.

[8] N. Tsantalis, "Evaluation and improvement of software architecture: Identification of design problems in object-oriented systems and resolution through refactorings," *Diss. Ph. D. dissertation, Univ. of Macedonia*, 2010.

[9] G. Szőke, C. Nagy, L. J. Fülöp, R. Ferenc, and T. Gyimóthy, "Faultbuster: An automatic code smell refactoring toolset," in *2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2015, pp. 253–258.

[10] F. Palomba, G. Bavota, M. Di Penta, F. Fasano, R. Oliveto, and A. De Lucia, "On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation," *Empirical Software Engineering*, vol. 23, no. 3, pp. 1188–1221, 2018.

[11] S. Olbrich, D. S. Cruzes, V. Basili, and N. Zazworka, "The evolution and impact of code smells: A case study of two open source systems," in *Proc. of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*. IEEE, 2009, pp. 390–400.

[12] R. Peters and A. Zaidman, "Evaluating the lifespan of code smells using software repository mining," in *2012 16th European Conference on Software Maintenance and Reengineering*. IEEE, 2012, pp. 411–416.

[13] D. I. K. Sjøberg, A. Yamashita, B. C. D. Anda, A. Mockus, and T. Dybå, "Quantifying the effect of code smells on maintenance effort," *IEEE Transactions on Software Engineering*, vol. 39, no. 8, pp. 1144–1156, Aug 2013.

[14] G. Hecht, N. Moha, and R. Rouvoy, "An empirical study of the performance impacts of android code smells," in *Proceedings of the international conference on mobile software engineering and systems*, 2016, pp. 59–69.

[15] R. Morales, R. Saborido, F. Khomh, F. Chicano, and G. Antoniol, "Earmo: An energy-aware refactoring approach for mobile apps," *IEEE Transactions on Software Engineering*, vol. 44, no. 12, pp. 1176–1206, 2017.

[16] D. Johannes, F. Khomh, and G. Antoniol, "A large-scale empirical study of code smells in javascript projects," *Software Quality Journal*, pp. 1–44, 2019.

[17] F. Khomh, M. Di Penta, Y.-G. Guéhéneuc, and G. Antoniol, "An exploratory study of the impact of antipatterns on class change-and fault-proneness," *Empirical Software Engineering*, vol. 17, no. 3, pp. 243–275, 2012.

[18] N. Zazworka, M. A. Shaw, F. Shull, and C. Seaman, "Investigating the impact of design debt on software quality," in *Proceedings of the 2nd Workshop on Managing Technical Debt*. ACM, 2011, pp. 17–23.

[19] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" *ACM sigsoft software engineering notes*, vol. 30, no. 4, pp. 1–5, 2005.

[20] D. A. Da Costa, S. McIntosh, W. Shang, U. Kulesza, R. Coelho, and A. E. Hassan, "A framework for evaluating the results of the szz approach for identifying bug-introducing changes," *IEEE Transactions on Software Engineering*, vol. 43, no. 7, pp. 641–657, 2016.

[21] E. C. Neto, D. A. Da Costa, and U. Kulesza, "The impact of refactoring changes on the szz algorithm: An empirical study," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2018, pp. 380–390.

[22] S. Herbold, A. Trautsch, F. Trautsch, and B. Ledel, "Problems with szz and features: An empirical study of the state of practice of defect prediction data collection," *Empirical Software Engineering*, vol. 27, no. 2, pp. 1–49, 2022.

[23] ——, "Issues with szz: An empirical assessment of the state of practice of defect prediction data collection," *arXiv preprint arXiv:1911.08938*, 2019.

[24] "Apache software fondation," http://www.apache.org.

[25] V. Lenarduzzi, F. Palomba, D. Taibi, and D. A. Tamburri, "Openszz: A free, open-source, web-accessible implementation of the szz algorithm," in *Proceedings of the 28th International Conference on Program Comprehension*, ser. ICPC '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 446–450. [Online]. Available: https://doi.org/10.1145/3387904.3389295

[26] D. Spadini, M. Aniche, and A. Bacchelli, *PyDriller: Python Framework for Mining Software Repositories*, 2018.

[27] "Replication package," https://github.com/vincedu/Anti-Pattern_Fault-Proneness.

[28] "Manual data," https://zenodo.org/record/5675024/files/release-level-data.tar.gz?download=1.

[29] N. Moha, Y.-G. Guéhéneuc, A.-F. Le Meur, L. Duchien, and A. Tiberghien, "From a domain analysis to the specification and detection of code and design smells," *Formal Aspects of Computing*, vol. 22, no. 3-4, pp. 345–361, 2010.

[30] B. A. Muse, M. M. Rahman, C. Nagy, A. Cleve, F. Khomh, and G. Antoniol, "On the prevalence, impact, and evolution of sql code smells in data-intensive systems," in *Proceedings of the 17th International Conference on Mining Software Repositories*, 2020, pp. 327–338.

[31] M. Abidi, M. S. Rahman, M. Openja, and F. Khomh, "Are multi-language design smells fault-prone? an empirical study," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 30, no. 3, pp. 1–56, 2021.

[32] Z. A. Kermansaravi, M. S. Rahman, F. Khomh, F. Jaafar, and Y.-G. Guéhéneuc, "Investigating design anti-pattern and design pattern mutations and their change-and fault-proneness," *Empirical Software Engineering*, vol. 26, no. 1, pp. 1–47, 2021.

[33] D. I. Sjøberg, A. Yamashita, B. C. Anda, A. Mockus, and T. Dybå, "Quantifying the effect of code smells on maintenance effort," *IEEE Transactions on Software Engineering*, vol. 39, no. 8, pp. 1144–1156, 2012.

[34] M. Abbes, F. Khomh, Y.-G. Gueheneuc, and G. Antoniol, "An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension," in *Software maintenance and reengineering (CSMR), 2011 15th European conference on*. IEEE, 2011, pp. 181–190.

[35] M. Cruz, B. Bernárdez, A. Durán, J. A. Galindo, and A. Ruiz-Cortés, "Replication of studies in empirical software engineering: A systematic mapping study, from 2013 to 2018," *IEEE Access*, vol. 8, pp. 26 773–26 791, 2020.

[36] O. S. Gómez, N. Juristo, and S. Vegas, "Understanding replication of experiments in software engineering: A classification," *Information and Software Technology*, vol. 56, no. 8, pp. 1033–1048, 2014.

[37] D. Fucci and B. Turhan, "On the role of tests in test-driven development: a differentiated and partial replication," *Empirical Software Engineering*, vol. 19, no. 2, pp. 277–302, 2014.

[38] A. Vetro, W. Bohm, and M. Torchiano, "On the benefits and barriers when adopting software modelling and model driven techniques-an external, differentiated replication," in *2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 2015, pp. 1–4.

[39] D. Fucci, D. Girardi, N. Novielli, L. Quaranta, and F. Lanubile, "A replication study on code comprehension and expertise using lightweight biometric sensors," in *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. IEEE, 2019, pp. 311–322.

[40] A. Rahman, M. R. Rahman, C. Parnin, and L. Williams, "Security smells in ansible and chef scripts: A replication study," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 30, no. 1, pp. 1–31, 2021.

[41] M. Di Penta, G. Bavota, and F. Zampetti, *On the Relationship between Refactoring Actions and Bugs: A Differentiated Replication*. New York, NY, USA: Association for Computing Machinery, 2020, p. 556–567. [Online]. Available: https://doi.org/10.1145/3368089.3409695

[42] A. Rahman, C. Parnin, and L. Williams, "The seven sins: Security smells in infrastructure as code scripts," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 164–175.

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

[44] D. Spadini, F. Palomba, A. Zaidman, M. Bruntink, and A. Bacchelli, "On the relation of test smells to software code quality," in *2018 IEEE international conference on software maintenance and evolution (ICSME)*. IEEE, 2018, pp. 1–12.