On the Distribution of "Simple Stupid Bugs" in Unit Test Files: An Exploratory Study

Anthony Peruma
Rochester Institute of Technology
Rochester, New York, USA
axp6201@rit.edu

Christian D. Newman
Rochester Institute of Technology
Rochester, New York, USA
cnewman@se.rit.edu

***Abstract*—A key aspect of ensuring the quality of a software system is the practice of unit testing. Through unit tests, developers verify the correctness of production source code, thereby verifying the system's intended behavior under test. However, unit test code is subject to issues, ranging from bugs in the code to poor test case design (i.e., test smells). In this study, we compare and contrast the occurrences of a type of single-statement-bug-fix known as "simple stupid bugs" (SStuBs) in test and non-test (i.e., production) files in popular open-source Java Maven projects. Our results show that SStuBs occur more frequently in non-test files than in test files, with most fix-related code associated with assertion statements in test files. Further, most test files exhibiting SStuBs also exhibit test smells. We envision our findings enabling tool vendors to better support developers in improving the maintenance of test suites.**

I. Introduction

Unit testing is an essential strategy employed by developers to ensure the quality of their software system. Under this strategy, developers write code (i.e., test cases) that verifies the behavior of individual units of work of the system under test [1]. However, the test code written by developers is vulnerable to issues such as bugs (i.e., functional defects) and smells (i.e., bad programming practices), which impact not only the quality of the system but also the system's maintenance; specifically, the maintenance of test cases [2].

However, correcting these software issues might not always be a complicated task, with some bugs being much easier to correct than others; often requiring a change to a single statement. These are sometimes referred to as "simple stupid bugs" (SStuBs) [3]. While there exist studies that investigate defects in source code, these studies on program repair [4] do not differentiate defects occurring in test and non-test files, nor do these studies focus exclusively on SStuB-like defects. These studies mostly focus on a test case's ability to identify defects in the system under test [5]–[7]. Furthermore, studies that focus on the quality of test cases focus on test smells [8] exhibited by test files such as investigating the flakiness (i.e., non-deterministic outcome) of test cases [9], [10]. In this paper, we focus on SStuBs in test files. We explore, compare, and contrast the existence of SStuBs in test files against non-test files. Furthermore, our study also looks at the co-occurrence between SStuBs and test smells.

A. Goal & Research Questions

The goal of this study is to explore the quality of test suites from a functional and non-functional perspective, and the relationship between these viewpoints. Hence, we utilize SStuBs to represent the functional quality aspects of the test suite, and test smells for the non-functional quality aspects. As an exploratory study, we aim to have a better understanding of this problem and determine the feasibility of further research in this area. Our findings provide developers with insight to better design, implement, and maintain test suites. Additionally, our findings can improve the support provided by refactoring and other code quality tools. Hence, our study aims at answering the following research questions (RQs):

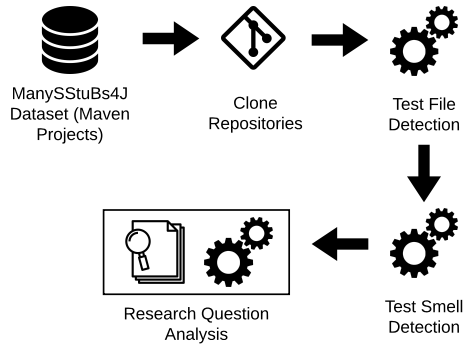

Fig. 1: Overview of our data extraction and collection process.

- **RQ_1: To what extent do SStuBs occur in test files as they do in non-test files?** This RQ compares the occurrence of SStuBs in test files and non-test files. We examine the volume and types of SStuBs occurring in files along with the developers responsible for fixing SStuBs in the two groups of file types.
- **RQ_2: To what extent do test files containing SStuB fixes also exhibit test smells?** This RQ explores the existence of test smells in files containing fixes for SStuBs. This RQ aims to understand if SStuBs can be an indicator for the presence of test smells in the code.

II. Data Extraction and Collection

Figure 1 shows a general overview of our data collection and extraction process. In the below subsections, we elaborate on the activities involved in the process. The dataset we utilize/generate is available at [12] for replication/extension.

A. Source Dataset

Our study utilizes the ManySStuBs4J dataset, which contains bug-fixing details of SStuBs extracted from popular open-source Java Maven projects [3]. Contained within this dataset is the relative path of the file containing the fix, the commit

TABLE I: Summary of the test smell detection rules utilized by TSDETECT [11].

Test Smell	Detection Rule
Assertion Roulette	A test method contains more than one assertion statement without an explanation/message (parameter in the assertion method)
Conditional Test Logic	A test method that contains one or more control statements (i.e., if, switch, conditional expression, for, foreach and while statement)
Constructor Initialization	A test class that contains a constructor declaration
Duplicate Assert	A test method that contains more than one assertion statement with the same parameters
Empty Test	A test method that does not contain a single executable statement
Exception Handling	A test method that contains either a throw statement or a catch clause
General Fixture	Not all fields instantiated within the `setUp` method of a test class are utilized by all test methods in the same test class
Ignored Test	A test method or class that contains the `@Ignore` annotation
Magic Number Test	An assertion method that contains a numeric literal as an argument
Mystery Guest	A test method containing object instances of files and databases classes
Redundant Print	A test method that invokes either the `print` or `println` or `printf` or `write` method of the `System` class
Redundant Assertion	A test method that contains an assertion statement in which the expected and actual parameters are the same
Resource Optimism	A test method utilizes an instance of a `File` class without calling the method `exists()`, `isFile()` or `notExists()` methods of the object
Sensitive Equality	A test method invokes the `toString()` method of an object
Sleepy Test	A test method that invokes the `Thread.sleep()` method
Unknown Test	A test method that does not contain a single assertion statement and `@Test(expected)` annotation parameter

id associated with the fix, and the type of SStuB fixed by the developer, among other details.

B. Repository Cloning & File Extraction

In this stage of the process, we clone the Maven repositories that contain bug-fixing commits for SStuBs. The purpose of this activity is twofold. First, we extract metadata associated with the commit, such as the author (i.e., developer) of the fix. Second, we extract the source code files associated with each SStuB-fixing commit. The original Maven dataset contains 10,231 SStuB-fixing instances distributed over 84 projects. However, at the time of cloning these repositories, only 83 of the repositories were publicly available. Hence, in this study, we analyze 10,225 SStuB-fixing instances.

C. Test File Detection

In this study, we only consider projects that utilize JUnit [13] as the testing framework since prior unit testing based research has frequently focused on JUnit (e.g., test smells [8]). JUnit recommends that developers either prepend or append the term 'Test' to the name of the production file to be tested (i.e., Test*.java or *Test.java). However, this mechanism can lead to false positives. For our study, we utilize the approach by [14] and [11] to detect test files. In this approach, we utilize JavaParser [15] to build an abstract syntax tree for each source file. We mark a file as a unit test file if the file contains JUnit import statements (i.e., `org.junit.*` or `junit.*`) and a test method. To be a test method, the method should have an annotation called `@Test` (JUnit 4), or the method name should start with 'test' (JUnit 3).

D. Test Smell Detection

For the detection of test smells we utilize TSDETECT [11], an open-source test smells detection tool. TSDETECT detects 19 test smell types and has been utilized in multiple studies [14], [16]–[20]. Provided in Table I is a summary of the detection rules for each smell type. However, for this study, we ignore the *Default Test* smell as this smell is exclusive to Android applications. We also ignore the smells *Eager Test* and *Lazy Test* as these smells require the class under test to be known; which is not known for this dataset. Hence, our analysis is limited to 16 smell types. Details about the smell types, along with examples, are available in [14].

E. Research Question Analysis

To answer our research questions, we execute a series of database queries and custom scripts on the mined data. Additionally, where applicable, we provide examples from our dataset to supplement our findings. When reporting on our research questions, we first provide details about the experiment's methodology before presenting the findings.

III. Experimental Results

A. RQ_1: To what extent do SStuBs occur in test files as they do in non-test files?

Methodology. In this RQ, we group the files containing fixes for SStuBs into two groups– test files and non-test files. The test files represent JUnit based unit testing files, while the non-test files represent the set of production files. We examine and compare details such as the distribution of SStuBs between the two groups, the developers responsible for applying the fix, and the clustering of file types in a bug-fixing commit.

Results. First, we look at the projects containing SStuBs fixing commits. From the set of 83 projects in the dataset, 57 ($\approx$ 68.67%) projects contain SStuBs occurring in both test and non-test files, while 25 ($\approx$ 30.12%) projects contain SStuBs occurring in only non-test files, and only one project contains SStuBs occurring in only test files. Our next examination shows that developers apply SStuB fixes to 5,587 Java files. From this set, 1,066 ($\approx$ 19.08%) are test files and 4,521 ($\approx$ 80.92%) are non-test files. Moreover, from a total of 10,225 instances of SStuB fixes, 1,946 ($\approx$ 19.03%) of these instances were in test files, while 8,279 ($\approx$ 80.97%) were in non-test files. Furthermore, we observe, on average, 2.20 SStuBs occur in test files, and 2.37 occur in non-test files.

Our next analysis focuses on SStuB categories. In total, there are 16 SStuB categories, the details of which are available at [3]. In Table II, we provide the top five frequently occurring SStuB categories in test and non-test files. From this table, we observe that even though the categories are the same for both file types, the ratio of occurrences of these categories

differs between file types. Further, we also observe that the categories *Change Numeric Literal* and *Change Modifier* occur at different positions relative to the other categories. These top five categories contribute to 88.23% and 81.83% of the complete set of occurrences in test and non-test file types, respectively. Additionally, we also observe that while the non-test files contain instances of all 16 categories, the test files do not exhibit the *Change Operand* category. In general, we observe 314 ($\approx$ 16.13%) SStuB instances in test files are associated with assertion statements. Examining the code statement containing the fix for *Change Numeric Literal* in test files, we observe that 112 ($\approx$ 20.4%) of the instances occur with an assertion statement (e.g., `assertEquals(2,map.size())` $\rightarrow$ `assertEquals(3,map.size())` [21]). Furthermore, 116 ($\approx$ 21.13%) instances are related to numeric values associated with time-related identifiers (e.g., `timeout=2000` $\rightarrow$ `timeout=1000` [22]). In contrast, non-test files contain five instances of SStuBs in assertion statements and 31 time-related SStuBs. For cases under the *Change Modifier* category, we observe that developers either remove or include the `static` keyword with a class or method in test files. We observe 14 cases falling under the *Same Function More Args* category are related to assertion methods in test files. In most of these cases, developers include a textual message related to the assertion condition's failure (e.g., [23]). Additionally, we also encounter instances, in test files, where the methods that are updated are related to mocking; either API's related to Mockito or custom methods that accept mocked objects as arguments (e.g., `Mockito.any()` $\rightarrow$ `Mockito.any(ProducerRecord.class)` [24]).

Our subsequent analysis is on the grouping of test and non-test files in a single commit. Since SStuBs can occur in both test and non-test files of a project, we are interested in knowing whether developers make fixes (i.e., commit) for both file types in unison or prioritize fixing one file type over another. Hence, for this analysis, we only focus on the 57 projects that contain SStuBs in test and non-test files. Our analysis shows that only one project meets this criterion. Furthermore, this project had only one commit operation that contained a test and non-test file. This phenomenon shows that developers prefer to treat SStuBs in isolation, most likely so that they can keep track of the changes and revert the change if necessary.

Finally, we examine the distribution of developers that fix SStuBs. Hence, we perform this analysis on the set of 57 projects containing test and non-test files. To detect unique developers, we utilize the same approach as [25]. The approach utilizes the SStuB fixing commit author's email to identify unique developers for a project. In total, we encounter 1,116 unique bug fixing developers. From this set, 84 ($\approx$ 7.53%) developers are responsible for fixing SStuBs occurring only in test files, while 789 ($\approx$ 70.70%) fix SStuBs occurring only in non-test files. Developers that fix SStuBs in both test and non-test files account for 243 ($\approx$ 21.77%).

Summary. This RQ shows that non-test files contain more SStuBs than test files. Additionally, we observe that the volume of SStuBs occurring in individual test and non-test files are very similar ($\approx$ 2 instances). We also observe that the majority of developers work on fixing SStuBs in non-test files than test files. Additionally, The top five popular SStuB categories for test and non-test files are the same. However, the ratio of occurrences of these categories differs. Furthermore, the code associated with SStuB fixes differs between test and non-test files; assertion statements in test files are frequently updated due to SStuB fixes.

TABLE II: Top five SStuB categories in (non-)test files.

SStuB Category	Count	Percentage
Test Files (Total Bug Type Instances: 1,946)		
Change Identifier Used	554	28.47%
Change Numeric Literal	549	28.21%
Wrong Function Name	310	15.93%
Same Function More Args	158	8.12%
Change Modifier	146	7.50%
Non-Test Files (Total Bug Type Instances: 8,279)		
Change Identifier Used	2,708	32.71%
Change Modifier	1,705	20.59%
Wrong Function Name	1,175	14.19%
Same Function More Args	600	7.25%
Change Numeric Literal	587	7.09%

B. RQ_2: To what extent do test files containing SStuB fixes also exhibit test smells?

Methodology. SStuBs represent the functional quality of a system. In contrast, test smells represent the non-functional quality of a system, both of which are important to the maintenance of the system's test suite. In this RQ, we look at the existence of test smells in test files exhibiting SStuBs. Additionally, we also investigate if the fixing of SStuBs causes a change in the number of test smell types exhibited by the file. To this extent, we utilize TSDETECT to analyze the test files containing SStuB fixes and the prior version (i.e., commit) of the file for the existence of test smells.

Results. From the set of 1,066 test files containing SStuB fixes, 1,064 files show the presence of test smells, with each file exhibiting, on average, 15.41 smell types. Examining each smell type, we observe that each of the 16 smell types occur in over 80% of the test files exhibiting SStuB fixes. Furthermore, the smell types *Assertion Roulette* and *Exception Handling* occur in all the test files. In terms of SStuB categories, we observe that *Change Numeric Literal* and *Change Identifier Used* are the top two categories occurring in 375 and 356 smelly test file instances, respectively. This is in contrast to the popularity results shown in Table II.

Next, we look at the co-occurrence of each SStuB category with each smell type. In Table III we present the number of instances (i.e., test files) where a SStuB bug category co-occurs with a test smell. Due to space constraints, we only present the top five occurring SStuB bug categories. Our analysis shows that the smells *Assertion Roulette*, *Exception Handling*, and *Magic Number Test* are the three smell types that most frequently co-occur with each of the SStuB categories. In contrast, the least co-occurring smell type is *Redundant Assertion*.

Finally, we look at the version of the test file the developer modifies just before the commit containing the SStuB fix. Our

TABLE III: Co-occurrence between the top five frequently occurring SStuB bug categories and test smell types.

SStuB Category	Assertion Roulette	Conditional Test Logic	Constructor Initialization	Empty Test	Exception Handling	General Fixture	Mystery Guest	Redundant Print	Redundant Assertion	Sensitive Equality	Sleepy Test	Duplicate Assert	Unknown Test	Ignored Test	Resource Optimism	Magic Number Test
Change Numeric Literal	375	373	354	326	375	366	344	346	304	355	364	372	372	368	355	373
Change Identifier Used	356	352	346	327	356	341	326	336	319	347	339	351	353	346	328	356
Wrong Function Name	192	189	183	170	192	177	170	177	160	184	178	188	190	183	172	192
Same Function More Args	110	110	109	97	110	106	104	102	100	109	105	110	110	107	104	110
Change Modifier	95	94	94	91	95	93	93	92	78	94	93	94	95	95	95	95
Other Categories	164	163	162	155	164	161	158	157	153	162	159	163	164	163	159	164

objective is to determine if the smell count decreases when the developer fixes a SStuB. Our findings show that most SStuB fixes do not result in a decrease in smell count. More specifically, 853 ($\approx$ 80.17%) instances do not show a change in smell count, while 182 ($\approx$ 17.11%) instances show a decrease and 29 ($\approx$ 2.73%) instances show an increase. Looking at the instances that show a decrease, we observe that, on average, 3.22, smell types are removed between the two versions of the file. We observe that the smells *Assertion Roulette* and *Exception Handling* do not show a reduction, while the smell *Redundant Assertion* is a popular smell that reduces.

Summary. Test files exhibiting SStuBs are very likely to contain test smells, with the *Assertion Roulette* and *Exception Handling* smell types frequently occurring in such files. Additionally, test files that exhibit certain types of SStuBs, such as *Change Numeric Literal*, also exhibit test smells. However, developers rarely fix these smells when addressing SStuBs.

IV. Discussion

As an exploratory study, our research aims to understand the extent to which SStuBs occur in test files and their relationship to test smell so as to provide direction to research areas that support developers in designing and maintaining test suites.

From $\mathbf{RQ}_1$, we observe that SStuBs tend to occur more frequently in non-test files than test files. However, this should not signify that test files are of better quality than non-test files; this might be a situation where developers might not always be addressing these types of defects in test suites. It is interesting to note that developers usually address test and non-test SStuBs in separate commits. This is interesting since a change in code in a non-test file would require an appropriate change to the test code to ensure that the test case passes [26]. Committing a non-test change independently of a test change would most usually cause the test case to fail in an automated build/test environment. Hence, it is most likely that such systems either do not use an automated build system or lack sufficient code coverage. Additionally, our observation of most developers working on fixing non-test SStuBs over test SStuBs can be a further indicator that developers prioritize non-test files over test files; and might need to adhere to a test-driven development approach [27]. However, further research into this area is warranted to understand the rationale as to why developers treat test files differently from non-test files. Additionally, there needs to be research into tools that support developers with automatically finding/recommending changes to test files based on SStuB fixes applied to non-test files.

$\mathbf{RQ}_1$ also shows that the code related to SStuB fixes tend to differ between test and non-test files. We observe that the code usually associated with SStuB fixes in test files are frequently associated with assertion statements, such as in the case of the *Change Numeric Literal* SStuB category. The frequent occurrence of issues related to assertion statements should not be surprising as asserts are one of the most fundamental parts of a test case. Furthermore, in $\mathbf{RQ}_2$, we observe the frequent occurrence of the smell *Assertion Roulette*; which corroborates with findings from prior test smell studies [2], [9]. These findings show that the occurrence of a specific SStuB category in the test suite indicates the occurrence of specific test smells in the same file, and thereby help developers in addressing multiple issues in the code that would otherwise be missed. A similar finding reported by Spadini et al. [9] shows an association between smelly tests and defect-proneness of the production code under test. For example, in our dataset, we observe instances where the *Change Numeric Literal* SStuB category occurs due to changes made to the `Thread.sleep()` value, which is also an indicator of the *Sleepy Test* smell (e.g., [28]). This specific smell can lead to unexpected results as the processing time for a task differs when executed in various environments and configurations; most likely, the developer experienced this situation due to the sleep duration change. Lastly, our findings on the non-removal of smells also corroborate with research by Tufano et al. [29]. Once more, all our findings can help tool/IDE vendors better equip their devices to better support developers to improve the functional and non-functional aspects of their test suites.

V. Threats to Validity

Even though the projects in our dataset are some of the most popular open-source Maven-based Java systems, the results may not generalize to systems written in other languages. Furthermore, we confine our analysis to the JUnit testing framework. However, prior unit testing-based research has frequently focused on JUnit [8]. Our selection of tsDetect is due to its ability to detect multiple smell types and better detection performance [20]. Finally, even though non-test files have more SStuBs than test files, the per-file rate of SStuBs of test and non-test files are very similar. This aspect is interesting and requires more in-depth analysis, such as the possibility that SStuBs are not found at the same rate in test and non-test files.

VI. Conclusion & Future Work

In this study, we explore the occurrence of SStuBs in test files and their relation to test smells. We observe that SStuBs occur more frequently in non-test files than test files and that most of the fix related code differs between test and non-test files. Finally, we show that test files exhibiting SStuBs also exhibit test smells and tend to co-occur with specific types of SStuBs. These findings show that there is indeed scope for future research in this area, especially around the maintenance of test suites. Future work in this area includes investigating the flakiness of test suites brought about by SStuBs.

REFERENCES

[1] R. Pressman and D. Bruce R. Maxim, *Software Engineering: A Practitioner's Approach*. McGraw-Hill Education, 2014.

[2] G. Bavota, A. Qusef, R. Oliveto, A. De Lucia, and D. Binkley, "An empirical analysis of the distribution of unit test smells and their impact on software maintenance," in *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, pp. 56–65, 2012.

[3] R.-M. Karampatsis and C. Sutton, "How often do single-statement bugs occur? the manysstubs4j dataset," in *Proceedings of the International Conference on Mining Software Repositories (MSR 2020)*, 2020.

[4] L. Gazzola, D. Micucci, and L. Mariani, "Automatic software repair: A survey," *IEEE Transactions on Software Engineering*, vol. 45, no. 1, pp. 34–67, 2019.

[5] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic patch generation learned from human-written patches," in *2013 35th International Conference on Software Engineering (ICSE)*, pp. 802–811, 2013.

[6] S. Shamshiri, R. Just, J. M. Rojas, G. Fraser, P. McMinn, and A. Arcuri, "Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges (t)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 201–211, 2015.

[7] M. M. Almasi, H. Hemmati, G. Fraser, A. Arcuri, and J. Benefelds, "An industrial evaluation of unit test generation: Finding real faults in a financial application," in *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, pp. 263–272, 2017.

[8] V. Garousi and B. Küçük, "Smells in software test code: A survey of knowledge in industry and academia," *Journal of Systems and Software*, vol. 138, pp. 52 – 81, 2018.

[9] 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)*, pp. 1–12, 2018.

[10] G. Grano, F. Palomba, D. Di Nucci, A. De Lucia, and H. C. Gall, "Scented since the beginning: On the diffuseness of test smells in automatically generated test code," *Journal of Systems and Software*, vol. 156, 2019.

[11] A. Peruma, K. Almalki, C. D. Newman, M. W. Mkaouer, A. Ouni, and F. Palomba, "Tsdetect: An open source test smells detection tool," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, (New York, NY, USA), p. 1650–1654, Association for Computing Machinery, 2020.

[12] "Dataset." https://doi.org/10.5281/zenodo.4608719.

[13] "Junit." https://junit.org/.

[14] A. Peruma, K. Almalki, C. D. Newman, M. W. Mkaouer, A. Ouni, and F. Palomba, "On the distribution of test smells in open source android applications: An exploratory study," in *Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering*, CASCON '19, (USA), p. 193–202, IBM Corp., 2019.

[15] "Javaparser." http://javaparser.org/.

[16] M. Schvarcbacher, D. Spadini, M. Bruntink, and A. Oprescu, "Investigating developer perception on test smells using better code hub-work in progress," in *2019 Seminar Series on Advanced Techniques and Tools for Software Evolution, SATTOSE 2019*, 2019.

[17] D. Spadini, M. Schvarcbacher, A.-M. Oprescu, M. Bruntink, and A. Bacchelli, "Investigating severity thresholds for test smells," in *Proceedings of the 17th International Conference on Mining Software Repositories*, MSR '20, 2020.

[18] D. J. Kim, "An empirical study on the evolution of test smell," in *Proceedings of the 42nd International Conference on Software Engineering: Companion Proceedings*, ICSE '20, 2020.

[19] A. Peruma, C. D. Newman, M. W. Mkaouer, A. Ouni, and F. Palomba, "An exploratory study on the refactoring of unit test files in android applications," in *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, ICSEW'20, (New York, NY, USA), p. 350–357, Association for Computing Machinery, 2020.

[20] A. Panichella, S. Panichella, G. Fraser, A. A. Sawant, and V. J. Hellendoorn, "Revisiting test smells in automatically generated tests: Limitations, pitfalls, and opportunities," in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 523–533, 2020.

[21] "Configurationpropertiesreportendpointserializationtests.java." https://github.com/spring-projects/spring-boot/commit/0757d24.

[22] "Beanrecipientlisttimeouttest.java." https://github.com/apache/camel/commit/cb6f6e2.

[23] "Evictiontest.java." https://github.com/hazelcast/hazelcast/commit/7fa8196.

[24] "Kafkaproducertest.java." https://github.com/apache/camel/commit/82bd0ba.

[25] A. Peruma, M. W. Mkaouer, M. J. Decker, and C. D. Newman, "Contextualizing rename decisions using refactorings, commit messages, and data types," *Journal of Systems and Software*, vol. 169, p. 110704, 2020.

[26] E. Burns and W. Prakash, *Hudson Continuous Integration in Practice*. McGraw-Hill Education, 2013.

[27] K. Beck, *Test-driven Development: By Example*. Addison-Wesley signature series, Addison-Wesley, 2003.

[28] "Exponentialbackoffmsgretrymanagertest.java." https://github.com/apache/storm/commit/9134320.

[29] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. D. Penta, A. De Lucia, and D. Poshyvanyk, "When and why your code starts to smell bad (and whether the smells go away)," *IEEE Transactions on Software Engineering*, vol. 43, no. 11, pp. 1063–1088, 2017.