Noname manuscript No.
(will be inserted by the editor)

The Secret Life of Test Smells - An Empirical Study on Test Smell Evolution and Maintenance

Dong Jae Kim · Tse-Hsun (Peter) Chen · Jinqiu Yang

Received: —/ Accepted: —

Abstract In recent years, researchers and practitioners have been studying the impact of test smells in test maintenance. However, there is still limited empirical evidence on why developers remove test smells in software maintenance and the mechanism employed for addressing test smells. In this paper, we conduct an empirical study on 12 real-world open-source systems to study the evolution and maintenance of test smells and how test smells are related to software quality. Results show that: 1) Although the number of test smell instances increases, test smell density decreases as systems evolve. 2) However, our qualitative analysis on those removed test smells reveals that most test smell removal (83%) is a by-product of feature maintenance activities. 45% of the removed test smells relocate to other test cases due to refactoring, while developers deliberately address the only 17% of test smells, consisting of largely *Exception Catch/Throw* and *Sleepy Test*. 3) Our statistical model shows that test smell metrics can provide additional explanatory power on post-release defects over traditional baseline metrics (an average of 8.25% increase in AUC). However, most types of test smells have a minimal effect on post-release defects. Our study provides insight into developers' perception of test smells and current practices. Future studies on test smells may consider focusing on the specific types of test smells that may have a higher correlation with defect-proneness when helping developers with test code maintenance.

Keywords Test Smell · Empirical Study · Software Quality

1 Introduction

In modern software development, developers need to continuously implement changes to the software system to keep up with the consumers' ever-growing

Dong Jae Kim · Tse-Hsun (Peter) Chen · Jinqiu Yang
Department of Computer Science and Engineering, Concordia University, Montreal, Canada
E-mail: {k_dongja, peterc, jinqiuy}@encs.concordia.ca

demands. As a software system evolves, tremendous collaborative effort takes place to deliver features and perform maintenance activities. Due to the importance of software quality, automated regression testing has played a pivotal role in software development. New test code is developed to test the newly-added SUT code and is executed after code changes to ensure that the new changes do not introduce defects (Ali *et al.*, 2019).

To ensure the effectiveness of regression testing, developers need to maintain a set of high-quality test cases to validate software quality continuously. Unfortunately, similar to source code, test code may also contain defects and design issues that hinder the quality of the test code. For example, prior studies (Lam *et al.*, 2019; Luo *et al.*, 2014) have found that the results of some test cases may be unreliable (e.g., flaky tests) due to defects in test code. Thus far, researchers and practitioners have started to notice recurring design problems in the test code (Spadini *et al.*, 2018; Van Deursen *et al.*, 2001) and have coined the term *test smell*. Like code smells in source code, test smells indicate potential design problems in test code. Bavota *et al.* (2015) found that test smells are prevalent in software systems and may hinder test comprehension and maintenance.

Despite the findings achieved so far, there is limited empirical evidence on the awareness of test smells. One research found that developers are aware of test smells and their potential consequences (Peruma *et al.*, 2019), while others found that developers do not believe the benefit of removing test smells (Junior *et al.*, 2020a,b; Tufano *et al.*, 2016). Therefore, by studying why developers remove test smells from how test smells are addressed during software maintenance, we can improve test code quality and develop an effective test code refactoring recommendation tool. Studying the maintenance will help (1) expand future research on understanding what may prompt developers to maintain test code, and (2) provide evidence on the most paid attention test smells.

In this paper, we conduct an empirical study on the maintenance of test smell in 12 large-scale open-source systems. We study a total of 18 different types of test smells that were defined and studied in prior research (Garousi and Küçük, 2018; Junior *et al.*, 2020a,b; Peruma *et al.*, 2019; Qusef *et al.*, 2019; Spadini *et al.*, 2018). In particular, we seek to answer the three following research questions:

RQ1: How do test smells evolve overtime? We conduct a quantitative analysis to study how tests smell evolve over three years (from 2016 to the beginning of 2019) in the studied systems. Although we find that the total number of test smell instances increases over time, the test smell density remains relatively stable in the 12 studied systems after normalizing by the total number of test code lines.

RQ2: What is the motivation behind removing test smells? We conduct a qualitative analysis of a statistically significant sample of the commits that removed test smells. We find that in only 17% of the sampled commits, developers directly address the test smells. In particular, developers are more likely to address two test smells: *Exception Catch/Throw* and *Sleepy Test*.

However, in 83% of the studied commits, the test smells are removed due to the deletion of test code or are relocated to other test cases due to feature refactoring activities. In short, we find that developers often do not directly address the test smells when maintaining test code.

RQ3: What is the relationship between test smells and software quality? Similar to prior work (Chen *et al.*, 2017; de Pádua and Shang, 2018; Moser *et al.*, 2008; Munson and Khoshgoftaar, 1992), we build a logistic regression model to study the relationship between test smell and software quality. Some test smells (e.g., *Conditional Test Logic*, *Exception Catch/Throw*, and *Mystery Guest*) have an increasing relationship with a source code file's defect-proneness when controlling for confounding factors like the traditional product, process and coupling metrics. However, most types of test smells have minimal effect on the defect-proneness.

In summary, our findings show that, as a system evolves, developers may allocate resources on maintaining test code, but they may not be aware of the test smells. Moreover, some test smells have a minimal effect on defect-proneness, while only a few test smells have a positive impact on defect-proneness. Future studies on test smells may consider focusing on the types of test smells that may have a higher correlation with defect-proneness when helping developers with test code maintenance.

Paper Organization. The rest of the paper is organized as follows. Section 2 describes an overview of test smells. Section 3 presents our research questions and results. Section 4 discusses the implications of our findings. Section 5 discusses the threats to validity. Section 6 surveys related work. Finally, Section 7 summarizes the paper.

2 Background

2.1 A Brief Overview of Test Smells

Software testing is a vital component of modern software development. Testing identifies defects in source code early on before the defects could incur substantial impact (Spínola *et al.*, 2019). It is widely adopted to utilize unit testing frameworks such as JUnit or TestNG to enable test automation, i.e., test cases are written in test code, which can be executed. Like source code, test code may also have design problems or even defects that hinder testing effectiveness. Therefore, there has been an increasing interest to properly maintain and improve the design of test code (Levin and Yehudai, 2017; Pinto *et al.*, 2012; Shamshiri *et al.*, 2018). Therefore, researchers and practitioners have started to study the quality of test cases and identify various problems in test code (Bavota *et al.*, 2015; Spadini *et al.*, 2018; Van Deursen *et al.*, 2001). In particular, researchers have coined the term *test smell* to characterize the recurring test design problems that may impair test comprehension and maintainability.

Table 1: An overview of the studied test smells. The first set of test smells was first proposed by Deursen *et al.* (2001), and investigated further by other studies on its diffusion [**DF**] in software systems (Bleser *et al.*, 2019; Palomba *et al.*, 2016), impact on software test code comprehension [**CO**] (Bavota *et al.*, 2012; Tufano *et al.*, 2016), developers' awareness of test smells [**AW**] (Peruma *et al.*, 2019; Spadini *et al.*, 2020; Tufano *et al.*, 2016) and relation with software quality [**QT**] (Athanasiou *et al.*, 2014). The remaining test smells are recently studied test smells by Peruma *et al.* (2019) and investigated on their diffusions and developers' perceptions.

Test smell	Abbrev.	Description
Literature Test Smells (Deursen *et al.*, 2001)		
Assertion Roulette	AR	One test case may contain several assertions with no explanation. AR increases difficulties in comprehension [**DF/AW/QT**].
Eager Test	EGT	A test case may exercise several methods of the object under test, which may increase the difficulty in test maintenance [**DF/AW/QT/CO**].
General Fixture	GF	A test case's fixture is too general, and the test code only accesses a part of it. The test case may execute unnecessary code and increase runtime overhead [**DF/AW/CO**].
Lazy Test	LT	Occurs when multiple test cases invoke the same method of the source code object, which may increase the difficulty in test maintenance [**DF/AW**].
Mystery Guest	MG	Test code that uses external resources. Tests containing such a smell are dicult to comprehend and maintain, due to the lack of information to understand them [**DF/AW/QT/CO**].
Resource Optimism	RO	A test case that makes optimistic assumptions about the state/existence of external resources, which may cause flaky test results. [**DF/AW/QT**].
Sensitive Equality	SE	A test using the `toString` method for equality check in assert statements. The test case is sensitive to the implementation of `toString` [**DF/AW/QT/CO**].
Recently Proposed Test Smells (Peruma *et al.*, 2019)		
Conditional Test Logic	CTL	There exist conditions in a test case that may alter the behavior of the test and its expected output.
Constructor Initialization	CI	A test class may use a constructor instead of JUnit's `setUp()`. This may introduce side effects when the test class inherits another class, i.e., the parent class's constructor will still be invoked.
Empty Test	ET	Occurs when test code has no executable statements.
Exception Catch/Throw	ECT	Passing or failing a test case depends on custom exception handling code or exception throwing (instead of using Junit expected attribute), which may hide real problems and hamper debug.
Print Statement	PS	Print statements in unit tests are redundant as unit tests are executed as part of an automated script and do not affect the failing or passing of test cases. Furthermore, they can increase execution time if the developer calls a long-running method from within the print method (i.e., as a parameter).
Redundant Assertion	RA	A test case may contain assertion statements that are either always true or always false.
Sleepy Test	ST	Occurs when explicitly making a thread to sleep in test cases can cause flaky test results.
Duplicate Assert	DA	Occurs when a test case tests the same condition multiple times, which may increase test overhead.
Unknown Test	UT	A test method is written without an assertion statement.
IgnoredTest	IT	A test case that is disabled using JUnit's `@Ignore`.
Magic Number Test	MNT	A test method contains unexplained and undocumented numeric literals as parameters or identifiers, which increases maintenance difficulty.

Table 1 shows the 18 different types of test smells that we include in our study. These test smells are studied in prior work (Bavota *et al.*, 2012; Bavota *et al.*, 2015; Garousi and Küçük, 2018; Junior *et al.*, 2020a; Knuth, 1981; Peruma *et al.*, 2019). In particular, the current knowledge of test smells that we know from the literature was first proposed by Deursen *et al.* (2001), and these were expanded as a basis for further investigation in recent studies. For instance, some studies (Bavota *et al.*, 2015; Bleser *et al.*, 2019; Tufano *et al.*, 2016) found a high diffusion of test smells in software systems, and such test smells may not be removed as systems evolve. Other studies investigated the impact of test smell on code comprehension by measuring the time taken for understanding the test code in the presence/absence of test smells (Bavota *et al.*, 2012). Moreover, Athanasiou *et al.* (2014) studied the impact of test smell on software quality (correlation with post-release defect) to fill the missing gap from numerous prior studies that only underlines its effects on software maintainability.

Numerous researchers also surveyed software engineers to understand their awareness, perception, or identification. For instance, a recent study by Peruma *et al.* (2019) proposed a new set of test smells and investigated their diffusion and awareness. Their result suggests that developers are aware of test smells and their potential consequences. On the contrary, others give evidence that developers do not believe software systems could genuinely benefit from addressing test smells (Junior *et al.*, 2020a; Tufano *et al.*, 2016). Nevertheless, there is a lack of empirical evidence on what types of test smell developers pay attention to the most and thereby maintain software evolution. Similarly, there is also missing evidence on the common reasons and mechanisms in which test smells are addressed. Hence, in this paper, we study how test smells evolve and how developers manage test smells during software maintenance. Moreover, we also explore whether the existence and maintenance of test smells correlate with software quality. Therefore, our work uses the detection tool implemented by Peruma *et al.* (2019) which include the most comprehensive type of test smells up to date, encompassing both the test smells from the literature and their newly proposed test smells.

2.2 Identifying Test Smells

In this paper, we focus on studying the evolution and maintenance of test smells. To identify test smells, we adopt a test smell detection tool called *tsDetector* implemented by Peruma *et al.* (2019) to analyze the studied systems. We choose *tsDetector* because it can detect a comprehensive list of test smells (i.e., 18 test smells in total, as described in Table 1) and has an average F-score of 96.5% (Peruma *et al.*, 2019). We focus on these 18 test smells because they are related to unit testing practices in Java (Peruma *et al.*, 2020), advocated in xUnit guidelines (Meszaros, 2007), and extensively studied in prior researches in test code maintainability and developers' perception (Bavota *et al.*, 2012; Junior *et al.*, 2020a). Although Garousi and Küçük (2018) sum-

marized a catalog of 198 test smells, many are general code smells specific to TCN language, come from grey literature (i.e., blog posts), and difficult to generalize (e.g., complicated setup, long-running test, long test file). *tsDetector* uses JavaParser to detect test smell given the lists of the test files and the corresponding source code under test (i.e., *CUT*). The CUT files are required to detect specific types of test smells, such as *Eager Test* and *Lazy Test*, whose primary concerns are about testing multiple CUT files in one test case, which may negatively impact code comprehension.

To identify each test file's corresponding CUT files, we follow prior studies and utilize the naming convention (Chen *et al.*, 2017; Peruma *et al.*, 2019; Spadini *et al.*, 2018; Tufano *et al.*, 2016; Zaidman *et al.*, 2008). In particular, for each test file, we identify the corresponding CUT files by removing the prefix or the suffix of *"[Tt]est(s*)"* from the names of the test files. We manually verify the build configuration files (e.g., Maven or Gradle build file) of the studied systems to use the default heuristic specified by Maven/Gradle plugin to identify test files. The default heuristic matches with the prefix that we use to determine the test files. The test smell detector *tsDetector* takes the lists of test files, and their associated CUT files and reports any occurrences of the 18 types of test smells.

Although *tsDetector* outputs test smells at a file-level, most reported test smells are a line-level and method-level, which are aggregated per file. In the rest of our analysis, we study each test smell individually, therefore, at their respective line and method-level. Furthermore, we modify the *tsDetector* to output the raw count of test smells instead of the default boolean value. To encourage the replication of our results, we have made the dataset publicly available.[1]

3 Case Study Results

We first introduce our studied systems. We then discuss the results of our research questions. For each research question, we discuss its motivation, the approach we use to address the question, and the results.

3.1 Case Study Systems

Table 2 shows an overview of the studied systems. We conduct our study on several versions of the 12 open-source Java systems. In particular, we conduct our research in all official releases from the beginning of 2016 to the beginning of 2019. We chose the studied systems based on the following selection criteria. First, we selected the top 1,000 Java projects on GitHub ordered by popularity (i.e., stargazer count). We also made sure that the repositories are not forks. Second, we discarded projects that are below the 90th percentile in terms of size (i.e., lines of code), repository popularity (i.e., stars), and the number of

[1] https://github.com/SPEAR-SE/TestSmellEmpirical_Data

Table 2: An overview of the studied systems.

Systems	#Releases	LOC in Source Code (2016 - 2019)	LOC in Test Code (2016 - 2019)
Kafka	9	95K - 265K	18K - 101K
Groovy	9	338K - 393K	8K - 9K
Camel	8	586K - 1.0M	379K - 484K
Zookeeper	4	128K - 119K	25K - 36K
Cxf	9	696K - 753K	195K - 218K
Karaf	11	132K - 168K	14K - 17K
Flink	8	388K - 731K	100K - 234K
Accumulo	7	420K - 577K	49K - 47K
Hive	11	3.5M - 4.4M	162K - 221K
Bookkeeper	9	102K - 200K	32K - 85K
Wicket	8	264K - 257K	54K - 57K
Cassandra	6	315K - 184K	43K - 112K
Hadoop	3	637K - 1M	418K - 658K
Total	102	6.9M - 9.1M	1.1M - 1.6M

commits. We also remove systems that do not use issue report systems. In the end, we are left with these 12 systems. As shown in Table 3, there are 998 active contributors in total (ranges from 15 to over 200 contributors) in the studied systems with a wide range of experiences (i.e., in terms of number of commits). In some systems, such as Kafka and Flink, the contributors' median number of commits is relatively high (i.e., 306 and 278 commits, respectively), which shows that many contributors are actively contributing to the systems. In Karaf, on the other hand, the median number of commits is only two. The studied systems are widely used by practitioners, used in many commercial settings, and are large in scale, with the number of lines of code (LOC) in source code ranges from 6.9M to 9.1M, and the LOC in test code ranges from 1.1M to 1.6M. Moreover, the studied systems maintain a set of comprehensive test cases and adopt the continuous integration practice by running the test cases daily basis (Apache, 2020). The studied systems also cover different domains, from big data processing and data warehousing solutions to distributed databases and programming languages.

3.2 RQ1: How do test smells evolve overtime?

Motivation: Prior studies (Bavota *et al.*, 2015; Tufano *et al.*, 2016) reveal that test smells are prevalent in software systems, and their presence hinders the comprehension and maintenance of test code. In light of these findings, there is limited empirical evidence of how the pervasiveness of test smell changes over time and its relation to software maintenance. Namely, while many other software artifacts should inevitably grow as the system evolves, we believe that an interesting implication can be investigated by studying the pervasiveness of test smells from the aspect of whether developers resolve test smells dur-

Table 3: An overview of the developer experience and the number of contributors in the studied systems.

	# Contributed Commits						
Systems	Min	Q1	Median	Q3	Max	Mean #Commits	Contributor
Accumulo	1	5.75	27.00	486.25	2528	7.16	26
Bookkeeper	18	40.75	87.00	217.25	2545	4.37	35
camel	1	6.00	31.00	217.25	24339	6.61	204
Cassandra	2	23.50	101.50	189.50	1320	3.20	102
Cxf	1	3.00	10.00	82.75	8767	4.88	47
Flink	1	98.50	278.00	791.50	3456	3.25	175
Groovy	2	24.00	68.00	530.25	909	1.34	15
Hive	5	28.50	122.50	325.00	3041	2.80	112
Kafka	20	53.00	306.00	607.00	918	3.24	188
Karaf	1	1.00	2.00	61.00	949	1.91	38
Wicket	1	7.00	34.00	320.00	4219	1.39	20
Zookeeper	3	12.50	47.50	92.75	671	2.03	36
Hadoop	1	54.74	225.50	591	2368	2.47	373

ing software evolution. Hence, in this RQ, we quantitatively investigate the evolution of test smells.

Approach: To study the evolution of test smells, we follow the approach described in Section 2.2 to detect test smells in each studied software version. In particular, we apply a test smell detection tool called *tsDetector* (Peruma *et al.*, 2019) to analyze the studied systems based on six-month windows from 2016 to 2019. In total, we obtain seven snapshots per studied system. We consider a six-month window because studying the evolution of test code on a commit by commit basis is expensive and dilute the modifications of test smells. Moreover, since the studied systems have different sizes and test smells may co-evolve with the amount of added test code and the raw number of the test smell instances, we also report test smell density. We calculate test smell density by dividing the number of test smell instances by the total number of code lines. We use code lines as our normalization metrics because many test smells are detected at the line level. We also normalized using other metrics such as the number of methods in a file, and we found a similar trend in the result.

Results: Figure 1 shows the time series plots of the averaged test smell density in the studied systems from 2016 to 2019. We present the studied systems with a similar scale of test smell densities in one plot for visualization ease. Averaged test smell density is the normalized test smell metrics that is averaged over all studied systems. We averaged the test smell densities to show a generalized trend amongst the studied systems. Although not shown, we observe that test smell density either remains stable (i.e., Bookeeper and Groovy) or decreases (i.e., Kafka, Camel, Accumulo, Wicket, Hive, Cassandra, and CXF) in most of the studied systems. When observing individual metrics as in Figure 1, we find that most test smell densities also stay relatively stable. However, ignored test smells increased far greater over-time compared to other test

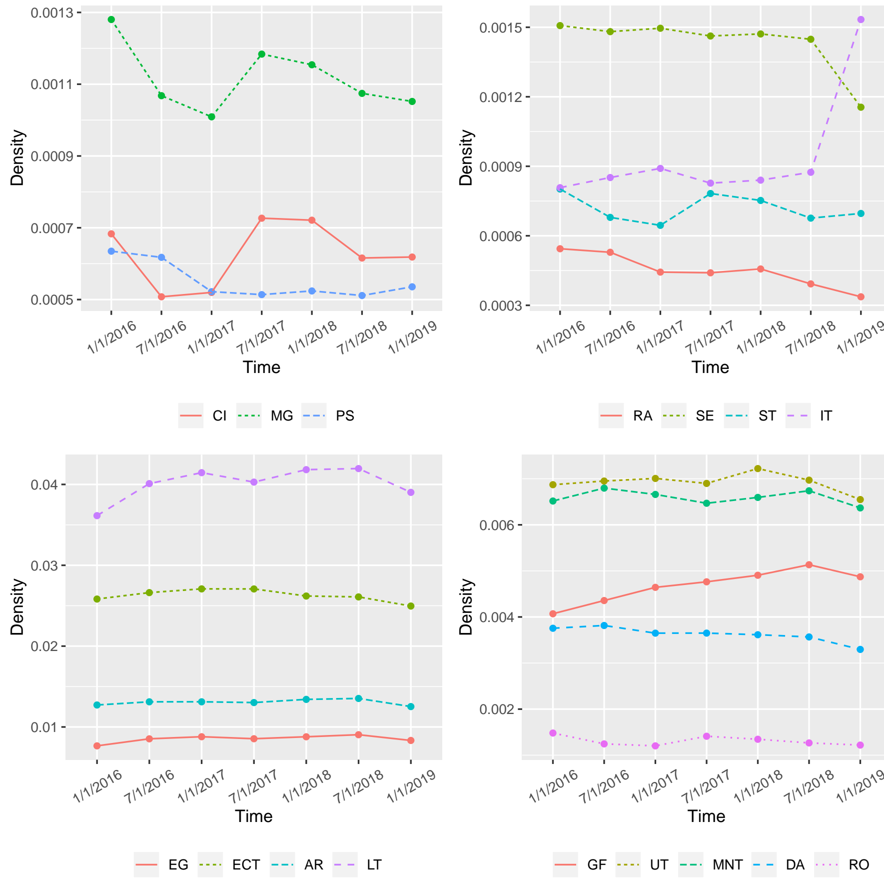

Fig. 1: Time series plots that show the evolution of the test smell density (normalized average) of the studied systems. The test smell densities are calculated based on seven snapshots that are taken every six months between 2016 and 2019.

smells. Figure 2 shows the evolution of the raw test smell metrics averaged over all of the studied systems. In general, we observe that all of the averaged raw test smell metrics increase over-time, but normalized test smell densities remain relatively stable.

We further investigate the change in the magnitude of test smell instances (i.e., raw counts) and test smell density between the two snapshots taken in 2016 and 2019. Table 4 shows the change in the magnitude of the test smell density, and similarly, Table 5 shows the change in the number of test smell instances (raw counts). As shown in Table 4, the test smell density decreases for most types of test smells. On the contrary, we find that the test smell instances' raw counts increase for most types of test smells (Table 5). Namely,

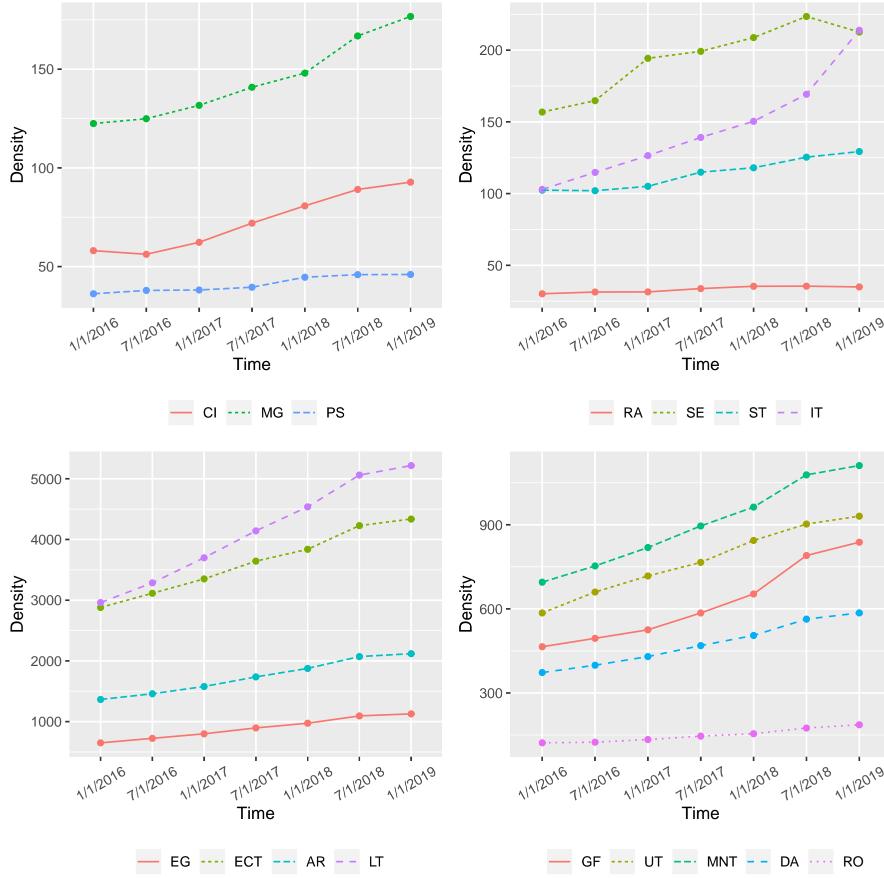

Fig. 2: Time series plots that show the evolution of the averaged raw test smell of the studied systems. The test smells are aggregated from on seven snapshots taken every six months between 2016 and 2019.

190 (81%) out of 234 (i.e., 18 test smell types times 12 studied systems) of the test smell types (across all studied systems) have an increase in the number detected test smell instances, which indicates that test smells are prevalent in the software and gradually grow over time. However, after normalized by the LOC of the test code, 121 out of 234 (51%) of the test smell types (across all studied systems) have a decreased test smell density. The findings may indicate that while the number of added test smell instances are higher as the systems evolve, test smell addition may be slower than that of test code addition. In other words, either developer may introduce fewer test smell instances when adding new test code or actively maintain test code, which results in the removal of test smell instances. We further study the reason for test smell removal in RQ2.

Table 4: The comparison of the test smell density (number of test smell instances per 1000 lines of test code) for each type of test smell in the studied systems from 2016 and 2019.

Test Smell	Accumulo			Bookkeeper			Camel			Cassandra			Cxf		
	2016	2019	%	2016	2019	%	2016	2019	%	2016	2019	%	2016	2019	%
AR	15.63	14.87	-5 % ↗	6.17	11.53	61 % ↘	12.58	13.61	8 % ↘	8.18	9.78	18 % ↘	14.67	14.69	-
CTL	5.39	5.45	1 % ↘	7.97	8.07	1 % ↘	3.47	3.78	9 % ↘	7.96	7.09	-12 % ↗	4.16	4.20	1 % ↘
CI	0.41	0.27	-40 % ↗	2.34	1.53	-42 % ↗	0.21	0.32	38 % ↘	0.06	0.17	101 % ↘	0.79	0.74	-6 % ↗
ET	0.05	0.02	-79 % ↗	0.00	0.00	-	0.19	0.14	-28 % ↗	0.00	0.00	- % ↘	0.06	0.05	-12 % ↗
ECT	19.67	15.88	-21 % ↗	19.46	24.50	23 % ↘	33.55	34.63	3 % ↘	21.85	20.75	-5 % ↗	29.11	28.33	-3 % ↗
GF	6.35	5.57	-13 % ↗	5.79	6.54	12 % ↘	2.05	2.37	14 % ↘	1.37	2.41	55 % ↘	4.26	3.93	-8 % ↗
MG	0.82	0.55	-40 % ↗	1.87	1.54	-19 % ↗	1.09	0.91	-17 % ↗	1.17	1.19	1 % ↘	1.40	1.27	-10 % ↗
PS	0.12	0.06	-63 % ↗	0.06	0.12	60 % ↘	0.08	0.06	-18 % ↗	0.35	0.27	-27 % ↗	0.08	0.10	24 % ↘
RA	0.92	0.34	-93 % ↗	0.06	0.32	134 % ↘	0.10	0.06	-52 % ↗	0.25	0.30	16 % ↘	0.12	0.12	-
SE	2.15	1.56	-32 % ↗	0.03	0.22	150 % ↘	0.85	0.77	-10 % ↗	0.28	0.51	59 % ↘	2.16	2.00	-7 % ↗
ST	0.56	0.55	-2 % ↗	3.01	1.47	-68 % ↗	1.32	0.94	-33 % ↗	0.87	0.76	-14 % ↗	0.47	0.53	12 % ↘
EG	14.11	15.22	8 % ↘	1.96	9.17	129 % ↘	3.11	3.51	12 % ↘	5.12	5.95	15 % ↘	6.71	6.49	-3 % ↗
LT	84.91	81.97	-4 % ↗	6.01	36.97	144 % ↘	11.92	15.00	23 % ↘	29.79	32.10	7 % ↘	24.74	24.40	-1 % ↗
DA	4.37	4.29	-2 % ↗	3.73	4.21	12 % ↘	2.04	2.47	19 % ↘	2.98	3.30	10 % ↘	3.03	3.01	-1 % ↗
UT	5.90	5.40	-9 % ↗	0.85	4.24	133 % ↘	3.46	4.28	21 % ↘	7.59	6.64	-13 % ↗	7.63	6.67	-13 % ↗
IT	0.46	0.61	28 % ↘	0.70	0.90	25 % ↘	0.86	1.10	25 % ↘	2.08	2.10	1 % ↘	0.94	0.78	-20 % ↗
RO	0.99	0.69	-35 % ↗	2.18	1.85	-17 % ↗	0.85	0.91	7 % ↘	1.22	1.24	2 % ↘	1.12	1.03	-9 % ↗
MNT	6.23	6.96	11 % ↘	3.04	6.62	74 % ↘	6.15	6.65	8 % ↘	5.45	5.75	5 % ↘	6.35	7.22	13 % ↘

	Flink			Groovy			Hive			Kafka		
	2016	2019	%	2016	2019	%	2016	2019	%	2016	2019	%
AR	12.24	12.92	5 % ↘	19.52	22.44	14 % ↘	13.17	11.46	-14 % ↗	16.67	16.83	1 % ↘
CTL	5.16	4.21	-20 % ↗	7.84	6.25	-23 % ↗	5.06	4.36	-15 % ↗	4.55	3.96	-14 % ↗
CI	0.73	0.70	-4 % ↗	0.77	0.74	-4 % ↗	0.64	0.76	17 % ↘	0.17	0.16	-9 % ↗
ET	0.04	0.10	83 % ↘	0.51	0.32	-47 % ↗	0.03	0.03	-12 % ↗	0.06	0.02	-98 % ↗
ECT	16.63	19.28	15 % ↘	64.35	58.01	-10 % ↗	23.08	25.94	12 % ↘	13.70	12.97	-5 % ↗
GF	0.57	2.85	133 % ↘	5.01	6.99	33 % ↘	5.68	6.08	7 % ↘	3.90	11.30	97 % ↘
MG	0.80	0.87	8 % ↘	1.03	0.85	-19 % ↗	0.43	0.50	16 % ↘	2.04	0.47	-125 % ↗
PS	0.10	0.03	-120 % ↗	3.21	1.91	-51 % ↗	0.50	0.37	-30 % ↗	0.00	0.07	200 % ↘
RA	0.04	0.03	-31 % ↗	3.08	1.59	-64 % ↗	0.93	0.75	-21 % ↗	0.06	0.13	76 % ↘
SE	0.26	0.45	52 % ↘	3.73	3.18	-16 % ↗	2.89	1.82	-46 % ↗	0.17	0.57	106 % ↘
ST	0.56	0.39	-36 % ↗	0.00	0.00	-	0.15	0.25	49 % ↘	0.12	0.11	-6 % ↗
EG	7.21	9.16	24 % ↘	9.38	15.24	48 % ↘	7.30	6.23	-16 % ↗	17.25	17.39	1 % ↘
LT	42.04	45.36	8 % ↘	43.42	75.80	54 % ↘	34.12	33.66	-1 % ↗	78.91	80.87	2 % ↘
DA	3.74	3.47	-8 % ↗	5.39	4.23	-24 % ↗	4.49	4.28	-5 % ↗	4.31	3.30	-27 % ↗
UT	3.31	5.32	46 % ↘	15.29	13.66	-11 % ↗	6.99	8.03	14 % ↘	4.43	5.12	14 % ↘
IT	0.54	1.23	78 % ↘	0.64	0.53	-19 % ↗	0.60	0.89	38 % ↘	0.29	0.38	26 % ↘
RO	0.89	0.93	4 % ↘	2.57	1.80	-35 % ↗	0.53	0.56	5 % ↘	2.10	0.59	-113 % ↗
MNT	4.90	5.44	10 % ↘	11.43	10.69	-7 % ↗	8.70	7.45	-15 % ↗	11.07	8.06	-32 % ↗

	Karaf			Wicket			Zookeeper			Hadoop		
	2016	2019	%	2016	2019	%	2016	2019	%	2016	2019	%
AR	10.96	11.84	8 % ↘	17.36	3.24	-137 % ↗	8.50	9.69	13 % ↘	9.70	9.92	2 % ↘
CTL	2.53	2.28	-10 % ↗	2.09	0.51	-121 % ↗	8.30	8.17	-2 % ↗	5.61	5.12	-9 % ↗
CI	0.49	0.37	-28 % ↗	1.29	1.20	-8 % ↗	0.56	0.66	17 % ↘	0.41	0.41	1 % ↘
ET	0.07	0.05	-28 % ↗	0.11	0.00	-200 % ↗	0.04	0.03	-36 % ↗	0.04	0.04	-11 % ↗
ECT	37.18	36.69	-1 % ↗	13.56	2.43	-139 % ↗	22.19	23.76	7 % ↘	21.41	21.31	-
GF	2.18	2.87	27 % ↘	5.31	0.12	-191 % ↗	4.27	5.85	31 % ↘	6.15	6.47	5 % ↘
MG	1.97	1.81	-9 % ↗	0.20	0.11	-63 % ↗	2.75	2.46	-11 % ↗	1.07	1.16	7 % ↘
PS	2.88	3.35	15 % ↘	0.13	0.02	-152 % ↗	0.16	0.14	-15 % ↗	0.58	0.48	-19 % ↗
RA	0.07	0.05	-28 % ↗	0.59	0.09	-148 % ↗	0.68	0.47	-36 % ↗	0.18	0.14	-22 % ↗
SE	0.56	0.90	46 % ↘	4.52	0.62	-152 % ↗	0.84	0.72	-15 % ↗	1.15	1.70	39 % ↘
ST	0.35	1.33	116 % ↘	0.00	0.04	200 % ↘	2.00	1.77	-12 % ↗	1.03	0.92	-11 % ↗
EG	5.83	7.49	25 % ↘	12.86	2.27	-140 % ↗	3.87	5.08	27 % ↘	4.93	5.12	4 % ↘
LT	21.44	28.04	27 % ↗	57.07	10.53	-138 % ↗	13.09	19.60	40 % ↘	22.38	22.99	3 % ↘
DA	1.83	1.65	-10 % ↗	5.07	0.81	-145 % ↗	4.23	4.20	-1 % ↗	3.58	3.64	1 % ↘
UT	17.36	15.03	-14 % ↘	5.41	0.65	-157 % ↗	6.62	5.74	-14 % ↗	4.48	4.36	-3 % ↗
IT	1.69	2.60	43 % ↘	0.17	7.32	191 % ↘	0.44	0.47	7 % ↘	1.09	1.03	-5 % ↗
RO	2.32	2.23	-4 % ↗	0.33	0.07	-130 % ↗	2.91	2.68	-8 % ↗	1.23	1.28	4 % ↘
MNT	5.69	6.27	10 % ↘	5.52	1.02	-138 % ↗	4.55	4.89	7 % ↘	5.62	5.78	3 % ↘

Discussion:

Since developers with higher experience may fix more test smells, we further study the correlation between developers' experience and test smell removal/addition. Following a prior study Rahman and Devanbu (2011), we use the number of previous commits as a proxy for developers' experience. From the beginning of 2016 to the beginning of 2019, we mined all the commits that modified the test file and calculated the test smell removal/addition for each unique contributor. Then we study the relationship between test smell removal/addition and developers' experience (i.e., in terms of the number of prior commits) by employing Spearman's rank correlation coefficient. We choose Spearman's rank correlation since it is a non-parametric correla-

Table 5: The comparison of the prevalence of test smells (i.e., the raw number of test smell instances) for the studied systems from 2016 to 2019.

Test Smell	Accumulo			Bookkeeper			Camel			Cassandra			Cxf		
	2016	2019	%	2016	2019	%	2016	2019	%	2016	2019	%	2016	2019	%
AR	647	707	9 % ↗	195	978	134 % ↗	4740	6573	32 % ↗	585	1317	77 % ↗	2845	3219	12 % ↗
CTL	223	259	15 % ↗	252	685	92 % ↗	1309	1828	33 % ↗	569	955	51 % ↗	807	921	13 % ↗
CI	17	13	-27 % ↘	74	130	55 % ↗	81	153	62 % ↗	4	23	141 % ↗	153	163	6 % ↗
ET	2	1	-67 % ↘	0	0	-	71	69	-3 % ↘	0	0	-	11	11	-
ECT	814	755	-8 % ↘	615	2079	109 % ↗	12641	16727	28 % ↗	1562	2795	57 % ↗	5644	6209	10 % ↗
GF	263	265	1 % ↗	183	555	101 % ↗	773	1144	39 % ↗	98	324	107 % ↗	826	861	4 % ↗
MG	34	26	-27 % ↘	59	131	76 % ↗	409	442	8 % ↗	84	160	62 % ↗	271	278	3 % ↗
PS	5	3	-50 % ↘	2	10	133 % ↗	29	31	7 % ↗	25	36	36 % ↗	16	23	36 % ↗
RA	38	16	-81 % ↘	2	27	172 % ↗	37	28	-28 % ↘	18	40	76 % ↗	23	26	12 % ↗
SE	89	74	-18 % ↘	1	19	180 % ↗	321	372	15 % ↗	20	69	110 % ↗	418	439	5 % ↗
ST	23	26	12 % ↗	95	125	27 % ↗	496	456	-8 % ↘	62	102	49 % ↗	91	116	24 % ↗
EG	584	724	21 % ↗	62	778	170 % ↗	1171	1698	37 % ↗	366	802	75 % ↗	1302	1422	9 % ↗
LT	3514	3898	10 % ↗	190	3137	177 % ↗	4490	7245	47 % ↗	2130	4324	68 % ↗	4797	5346	11 % ↗
DA	181	204	12 % ↗	118	357	101 % ↗	770	1194	43 % ↗	213	444	70 % ↗	588	659	11 % ↗
UT	244	257	5 % ↗	27	360	172 % ↗	1305	2067	45 % ↗	543	894	49 % ↗	1479	1462	-1 % ↘
IT	19	29	42 % ↗	22	76	110 % ↗	324	533	49 % ↗	149	283	62 % ↗	183	170	-7 % ↘
RO	41	33	-22 % ↘	69	157	78 % ↗	320	442	32 % ↗	87	167	63 % ↗	218	225	3 % ↗
MNT	258	331	25 % ↗	96	562	142 % ↗	2318	3211	32 % ↗	390	774	66 % ↗	1232	1583	25 % ↗

	Flink			Groovy			Hive			Kafka		
	2016	2019	%	2016	2019	%	2016	2019	%	2016	2019	%
AR	1204	3031	86 % ↗	152	212	33 % ↗	1721	2533	38 % ↗	286	1692	142 % ↗
CTL	508	988	64 % ↗	61	59	-3 % ↘	662	964	37 % ↗	78	398	134 % ↗
CI	72	165	78 % ↗	6	7	15 % ↗	84	168	67 % ↗	3	16	137 % ↗
ET	4	23	141 % ↗	4	3	-29 % ↘	4	6	40 % ↗	1	2	67 % ↗
ECT	1636	4521	94 % ↗	501	548	9 % ↗	3017	5734	62 % ↗	235	1304	139 % ↗
GF	56	669	169 % ↗	39	66	51 % ↗	743	1344	58 % ↗	67	1136	178 % ↗
MG	79	205	89 % ↗	8	8	-	56	111	66 % ↗	35	47	29 % ↗
PS	10	6	-50 % ↘	25	18	-33 % ↘	65	81	22 % ↗	0	7	200 % ↗
RA	4	7	55 % ↗	24	15	-46 % ↘	121	166	31 % ↗	1	13	171 % ↗
SE	26	106	121 % ↗	29	30	3 % ↗	378	402	6 % ↗	3	57	180 % ↗
ST	55	91	49 % ↗	0	0	-	20	56	95 % ↗	2	11	138 % ↗
EG	709	2149	101 % ↗	73	144	65 % ↗	954	1378	36 % ↗	296	1749	142 % ↗
LT	4135	10637	88 % ↗	338	716	72 % ↗	4460	7442	50 % ↗	1354	8131	143 % ↗
DA	368	814	75 % ↗	42	40	-5 % ↘	587	946	47 % ↗	74	332	127 % ↗
UT	326	1247	117 % ↗	119	129	8 % ↗	914	1775	64 % ↗	76	515	149 % ↗
IT	53	288	138 % ↗	5	5	-	79	196	85 % ↗	5	38	153 % ↗
RO	88	219	85 % ↗	20	17	-16 % ↘	69	123	56 % ↗	36	59	48 % ↗
MNT	482	1275	90 % ↗	89	101	13 % ↗	1137	1647	37 % ↗	190	810	124 % ↗

	Karaf			Wicket			Zookeeper			Hadoop		
	2016	2019	%	2016	2019	%	2016	2019	%	2016	2019	%
AR	156	223	35 % ↗	941	184	-135 % ↘	213	351	49 % ↗	4057	6527	47 % ↗
CTL	36	43	18 % ↗	113	29	-118 % ↘	208	296	35 % ↗	2344	3367	36 % ↗
CI	7	7	-	70	68	-3 % ↘	14	24	53 % ↗	170	269	45 % ↗
ET	1	1	-	6	0	-200 % ↘	1	1	-	17	24	34 % ↗
ECT	529	691	27 % ↗	735	138	-137 % ↘	556	861	43 % ↗	8953	14014	44 % ↗
GF	31	54	54 % ↗	288	7	-191 % ↘	107	212	66 % ↗	2571	4254	49 % ↗
MG	28	34	19 % ↗	11	6	-59 % ↘	69	89	25 % ↗	449	760	51 % ↗
PS	41	63	42 % ↗	7	1	-150 % ↘	4	5	22 % ↗	242	314	26 % ↗
RA	1	1	-	32	5	-146 % ↘	17	17	-	74	93	23 % ↗
SE	8	17	72 % ↗	245	35	-150 % ↘	21	26	21 % ↗	480	1118	80 % ↗
ST	5	25	133 % ↗	0	2	200 % ↗	50	64	25 % ↗	432	607	34 % ↗
EG	83	141	52 % ↗	697	129	-138 % ↘	97	184	62 % ↗	2062	3365	48 % ↗
LT	305	528	54 % ↗	3093	598	-135 % ↘	328	710	74 % ↗	9358	15116	47 % ↗
DA	26	31	18 % ↗	275	46	-143 % ↘	106	152	36 % ↗	1499	2393	46 % ↗
UT	247	283	14 % ↗	293	37	-155 % ↘	166	208	22 % ↗	1872	2865	42 % ↗
IT	24	49	68 % ↗	9	416	192 % ↗	11	17	43 % ↗	455	680	40 % ↗
RO	33	42	24 % ↗	18	4	-127 % ↘	73	97	28 % ↗	516	841	48 % ↗
MNT	81	118	37 % ↗	299	58	-135 % ↘	114	177	43 % ↗	2351	3798	47 % ↗

tion test that does not assume the underlying data distribution. We found a positive correlation (i.e., 0.53) between test smell addition and developers' experience and found a negative correlation (i.e., -0.57) between test smell removal and developers' experience. The correlation analysis suggests a non-negligible correlation that experienced developers are more likely to add more test smells and remove fewer test smells. One potential explanation for our result is that even highly experienced developers often do not refactor test smells due to lack of awareness or benefits, which aligns with a prior survey (Peruma *et al.*, 2019). Another reason may be that most experienced developers work on the most often exercised and most complex part of the system (Zeller, 2009). To corroborate our result, we study the relationship between the size

of code changes (i.e., lines of code and deleted) and developers' experience. We employ quantile-based correlation, where we split developers into four different quantiles based on experiences and studied their correlation with code size. Our result shows an increase in the correlation between experience and code size (i.e., 0.26, 0.50, 0.67, and 0.70). Therefore, we observe that the high expertise team may not necessarily remove more test smells because they may be responsible for larger and more complex changes.

Although the number of test smells increases as the systems evolve, after normalization against $Test_{LOC}$, the test smell density generally remains stable. We also find that some types of test smell, such as *Eager Test*, *Ignored Test*, *Unknown Test*, *Lazy Test*, and *Sleepy Test*, have one of the largest increase in terms of test smell density in most studied systems. In contrast, *Exception Catch/Throw*, *Redundant Assertion*, and *Print Statement* have the largest decrease in terms of test smell density in most studied systems.

3.3 RQ2: What is the motivation behind removing test smells?

Motivation: A recent study (Garousi and Küçük, 2018) claims that developers perceive test smell as harmful in software systems. In contrast, other studies reveal that developers are unaware of test smells and do not acknowledge the benefits of refactoring them (Junior *et al.*, 2020a; Tufano *et al.*, 2016). Nevertheless, there is limited empirical support on test smell removals, whether a test smell vanishes as a side-effect of code evolution or is a deliberate refactoring target. Such evidence is necessary to reveal the current perception developers may have on test smells. Hence, we perform a qualitative study on the test smell removing commits to identify reasons that prompt developers to fix the test code with test smells and the mechanisms employed to address test smells.

Approach: We conduct a qualitative study on commits removing test smells. We leverage Git to extract all the commits except for the merge commits between 2016 and 2019. We only keep the commits that include test file modifications and discard the other commits from further analysis. A commit modifies a test file if the involved files have the extension *".java"* and have a prefix or a suffix of *"[Tt]est(s*)"*. For each of the commits, we run *tsDetector* on the two versions of the software (i.e., every two consecutive commits) and calculate the raw test smell differences.

To understand why developers remove test smells, we look at a combination of bug reports, commit messages, test code, and the commit history of relevant test code. In the studied samples, 241 commits (80%) includes issue ID from the Jira bug report, and 25 commits (8%) use GitHub's pull request/issue tracking. The remaining 34 commits (11%) only contained commit messages, which contained sufficient information to understand the reason behind test

Table 6: A summary of our manual analysis on the commits of test smell removal, i.e., 304 sampled commits minus 12 commits that are incorrectly flagged by the test smell detection tool. Our analysis focuses on the context of each commit and how the test smell is addressed in the commit. In particular, we show the association between test code changes and the corresponding maintenance activities that developers apply.

	Maintenance Activities					
Code change	Refactoring Test Code	Feature Improvement	Bug Fixing	Feature Addition	Others	Total #
TEST SMELL AWARE REFACTORING						
Exception Catch/Throw	9	4	3	5	-	21
Sleepy Test	9	4	1	1	-	15
Unknown Test	-	3	1	1	2	7
Assertion Roulette	1	-	1	-	-	2
Sensitive Equality	-	-	2	-	-	2
Magic Number	1	-	-	1	-	2
Conditional Test Logic	-	1	-	-	-	1
Total	20	12	8	8	2	50
TEST SMELL UNAWARE REFACTORING						
Persistence	10	22	12	13	1	58
By-Product Removal	20	18	22	10	5	75
Total	30	40	34	23	6	133
OTHER CODE CHANGES						
Test Code Deletion	13	33	30	20	1	97
Add Comment/@Ignore	2	1	1	3	-	7
Revert a Commit	-	1	3	1	-	5
Total	15	35	34	24	1	109
#Total	65	87	76	55	9	292

code changes. Such commit messages may include keywords like *"Refactor"* or *"Fix test speed."* Using the artifacts above, we answer two types of questions: 1) What kind of maintenance activity initially prompted developers to address test smells (i.e., the main purpose of the commit)? 2) How is the test smell addressed (e.g., deliberate refactoring or by-product of other maintenance activities)? We used the two preliminary inquiries to gain further insight into developers' awareness of the most common reason for removing test smell.

In the analysis, we take a statistically significant sample of the commits removing test smells. In particular, we apply stratified random sampling on these commits with a 95% confidence level and a 5% confidence interval. We adopted stratified sampling to sample the test smells in each studied system independently, which can be advantageous to reduce sampling error when a subpopulation within the overall population varies (Zhao *et al.*, 2019). The first two authors examine the sample independently. Any disagreement is discussed until reaching a consensus. To assist our qualitative study, we leveraged a tool called Refactoring Aware Commit Review (Tsantalis *et al.*, 2018), which is a code diff visualization tool for showing the refactoring activities applied between two commits.

Results: In total, we manually analyzed 304 commits from a total of 1,452 commits (achieving a 95% confidence level with a 5% confidence interval). Table 6 shows a two-dimensional summary illustrating the association between the maintenance activities that initially prompted developers to maintain test code (horizontal dimension) and the type of specific test code changes (vertical dimension) that developers applied when removing test smells. We also found 12 incorrectly detected test smell instances by *tsDetector*, i.e., a 4% false-positive rate, and excluded them in Table 6.

For the maintenance activities (horizontal), we uncover five categories that prompted developers to maintain test code. Four of the five categories are: refactoring test code (65 commits), feature improvement (87 commits), bug fixing (76 commits), and adding new functionality (55 commits). The remaining nine commits (i.e., the fifth category - "others") consist of the ones that we cannot identify clear motives due to insufficient documentation (e.g., low-quality commit messages and bug reports). As an example, in CXF (*bcb6385a*), the developer addresses a test smell (i.e., *Ignored Test*) by completing the test implementation. However, the test case was ignored when it was first introduced to the codebase years ago and did not reference any bug report. Thus, it is difficult to label the correct maintenance motives.

We classify the type of test code changes (vertical) into three categories: *Test Smell Aware Refactoring* (50 commits), *Test Smell Unaware Refactoring* (133 commits), and *Other Code Changes* (109 commits). We classify a commit in the category of *Test Smell Aware Refactoring* if developers directly addressed the test smell. We classify a commit as a *Test Smell Unaware Refactoring* if the test smells are removed as a side-effect of other activities. *Test Smell Unaware Refactoring* consists of two subcategories: *Test Smell Persistence* and *By-product Removal*. The *Test Smell Persistence* (58 commits) shows instances of applying standard code refactoring, such as extracting common test code, where test smells are transiently relocated to another test class. The *By-product Removal* (75 commits) represents the cases when the test smell is removed due to refactoring and maintenance of other tasks (e.g., removing duplicate source code). We group the remaining commits that remove test smells but are not test smell specific refactoring into the *"Other Code Changes"* (109 commits). These commits made changes such as deleting test code, disabling tests (commenting or ignoring code), and reverting a commit.

In the following subsections, we discuss our qualitative analysis results of the three high-level test smell removal categories.

Test Smell Aware Refactoring

Although less frequent, we find that developers directly refactor specific test smells in 50 out of 292 (17%) studied commits. As shown in Table 6, these commits are related to removing *Exception Catch/Throw* (21 commits), *Sleepy Test* (15 commits), *Unknown Test* (7 commits), *Assertion Roulette* (2 commits), *Sensitive Equality* (2 commits), *Magic Number* (2 commits), and *Conditional Test Logic* (2 commits). By looking into the vertical dimension (i.e., the context of the commits), we find that *Test Smell Aware Refactoring* hap-

pen more frequently during test refactoring commits (20/50), but developers also deliberately address test smells during other maintenance activities (e.g., feature improvement and bug fixing).

However, not all test smell removing commits are deliberately fixed by developers. Therefore, we also report the proportion of test smell *"fixing"* commits (i.e., deliberately fixed by developers) over the number of test smell removing commits in our stratified samples (i.e., all commits that remove the specific test smell), which shows the true proportion of the fixed test smells. In this case, even though *Exception Catch/Throw* has the highest number of test smell removing commits, only 31% were fixed test smell instances. *Sleepy Test* has the highest number of intentional fixes (60%) compared to other test smells. 21% of *Unknown Test*, 12.5% of *Sensitive Equality*, 5.1% of *Magic Number*, 2.9% of *Assertion Roulette*, and 1.9% of *Conditional Test Logic* are deliberately fixed by developers. Our findings show that, in the studied systems, developers are more likely to fix *Sleepy Test* and *Exception Catch/Throw* due to the existence of test smell instead of other maintenance activities.

Refactor Sleepy Test. 15 out of the 292 (5%) commits are related to refactoring the *Sleepy Test* test smell. This finding shows that developers deliberately fixed 60% of removed *Sleepy Test*. This test smell occurs when developers explicitly cause a thread to sleep, leading to unexpected results as the processing time for a task can differ on different devices (Meszaros, 2007). We find that developers often remove *Sleepy Test* due to unexpected test behavior and increased test time. For example, in Kafka (*7b7c4a7*), a developer mentions that:

"The timeouts are often large (e.g., 10 seconds) and still occasionally they trigger prematurely. They need to be replaced by waitUntilTrue and some logic that checks when processing in streams is complete".

In another example, a developer in Camel (*722e590c*) mentions that *"[u]se awaitility for testing where we otherwise use thread sleep which can be speeded up."*. We find developers often apply two approaches to address *Sleepy Test*. One is to use `waitFor()` condition in the Java *Awaitility* library and the other is to refactor the test smell using Java's *Future* library. These two approaches allow the test case to run asynchronously without blocking.

As presented in Table 4 (RQ1), while *Sleepy Test* accounts for one of the most prevalent test smells, we find in our manual study that developers also allocate some efforts to refactor such smells. The awareness for *Sleepy Test* may be a result of the increase in attention for the unreliability in test code qualities (e.g., flaky tests) (Eck *et al.*, 2019; Lam *et al.*, 2019; Shi *et al.*, 2019). Moreover, we find that developers may also be concerned with an increased test execution time caused by calling `thread.sleep()`. Future research is needed to understand further developers' awareness and opinion on the consequences of *Sleepy Test*.

Refactor Exception Catch/Throw. We find that 21 out of the 292 (7.2%) commits are related to refactoring the *Exception Catch/Throw* test smell. This finding shows that developers deliberately fixed 31% of the removed *Exception*

Catch/Throw. As discussed in a previous paper (Peruma *et al.*, 2019), this test smell occurs when the passing or failing of the test is dependent on custom exception handling code or exception throwing instead of using JUnit's expected attribute. In this category, developers deliberately refactored the test smell in 10 commits, and the remaining were refactored during other maintenance activities: feature improvement (3 commits), bug fixing (3 commits), and feature addition (5 commits). As shown in Listing 1, developers remove the code logic in the catch block that determines the passing and failing of the test case. The developers mentioned in the bug report that using fail in the catch block is a bad programming style and masks the details of the stack trace[2]. After removing the test smell, a new test smell (i.e., unknown test) is introduced. We have also seen other similar cases in our study, where a new test smell is introduced after developers resolved the current test smell. Our finding shows that developers are sometimes unaware of the test smells; thus, they are likely to introduce new test smells when addressing existing ones. In short, our manual study finds that developers are more likely to refactor the *Exception Catch/Throw* test smell during various maintenance tasks. We also find that these test smells removal is often not associated with test failures but improves future maintainability.

Listing 1: Developers removed the dependency of test outcome on exception handling code (Flink - *e83217bd*).

```
1  @Test
2  public void testZeroSizeHeapSegment() {
3  -    try{
4           MemorySegment segment = new HeapMemorySegment(new byte[0]);
5           testZeroSizeBuffer(segment);
6           testSegmentWithSizeLargerZero(segment);
7  -    }
8  -    catch (Exception e) {
9  -        e.printStackTrace();
10 -        fail(e.getMessage());
11 -    }
12 }
```

Unknown Test. 7 out of 292 (<3%) commits are removal of test smell called *Unknown Test*. This finding shows that developers deliberately fixed 21% of the removed *Unknown Test*. This test smell occurs when test cases do not contain any logic or assertions statements. Thus, it is challenging to comprehend what the role of the test case is. As an example, in Kafka (*7d6ca52a*), the test class called *JmxReporterTest.java* is added three years ago. However, three years after its creation, the developers noticed missing test code while working on other tasks and immediately addressed it. Similarly, as mentioned in CXF (*bcb6385*), developers completed the missing test implementation two years ago. Thus, our finding suggests that there might be other instances of

[2] https://github.com/apache/flink/pull/4446

the *Unknown Test*, where developers may only notice them while maintaining other tasks. One potential reason for adding *Unknown Test* code may be that in feature additions, the *Unknown Test* gets added to prepare for the future implementation. This is illustrated in CXF (*2705f4d*), *[CXF-7525] Completing the system test*, where developers initially only provided empty test cases when implementing the feature and later complete the test case. While adding *Unknown Tests* may serve as code documentation to describe what test cases should be implemented in the future; developers may forget to complete the test case and become technical debt (Pham and Yang, 2020; Spínola *et al.*, 2019).

Refactor Sensitive Equality. We find that developers refactor the *Sensitive Equality* test smell in 2 out of 292 (<1%) commits. The finding shows that developers deliberately fixed 12.5% of the removed *Sensitive Equality.* This test smell occurs when the test method verifies objects by invoking the toString() method. The potential consequence of the test smell is that the change in the implementation of toString() might result in test failure (Meszaros, 2007). We find a similar discussion in Flink (*390d3613*), "*rerollercoasting through abstraction layer; we don't really know what the implementation is by calling tostring?*". Thus, our findings show that developers may be aware of the inherent issues associated with using a default method with unknown implementation.

In the commits, while there were a total of 16 samples of the removed *Sensitive Equality*, only 2 out of 16 commits (12%) reflected an awareness of the test smell (in other cases, developers delete the entire test method or move the whole test code to another test case). It may be because the use of the default toString() is intuitive from both its purpose and naming convention, and thus, developers may not have an immediate incentive to address the test smell until the test fails.

Refactor Magic Number. We find that developers refactor *Magic Number* in 2 out of 292 (<1%) commits. This finding shows that developers deliberately fixed 5.1% of the removed *Magic number.* This test smell occurs when assert statements in a test method contain numeric literals (i.e., Magic Numbers) as parameters. *Magic Number* does not indicate the meaning/purpose of the number. Hence, they should be replaced with constants or variables, thereby providing a descriptive name for the input (Meszaros, 2007). As an example in Kafka (*7ebc5da6*), the test smell was refactored after a feature addition, which involved explicitly replacing the *Magic Number* with a variable with a more meaningful variable name to improve code comprehension.

Refactor Assertion Roulette. We find that 2 out of 292 (<1%) commits are from refactoring *Assertion Roulette* (AR). The finding shows that developers deliberately fixed 2.9% of the removed *Assertion Roulette.* This test smell occurs when the test method has several assertion statements making it challenging to determine which assertion had failed (Meszaros, 2007). Although prior work (Deursen *et al.*, 2001; Meszaros, 2007) proposes using an assertion explanation to refactor the test smell, we find that developers may also remove the test smell using another assertion statement. Figure 3 shows an

```
36 -        Conditional conditional =
feature.getConditional().get(0);
37 -        assertNotNull(conditional.getCondition());

38 -        assertEquals(1,conditional.getCondition().size());
39 -        String dependency = conditional.getCondition().get(0);
40 -        assertNotNull(dependency);

41 -        assertEquals("http", dependency);


42 -        assertNotNull(conditional.getBundles());
43 -        assertEquals(1,
feature.getConditional().get(0).getBundles().size());
44 -
45 -        conditional = feature.getConditional().get(1);
46 -        assertNotNull(conditional.getCondition());
47 -        assertEquals(1,conditional.getCondition().size());
48 -        dependency = conditional.getCondition().get(0);
49 -        assertNotNull(dependency);
50 -        assertEquals("req:osgi.ee;filter:=\"(&(osgi.ee=JavaSE)
(!(version>=1.7)))\"", dependency);
```

```
38 +        Conditional conditional1 =
feature.getConditional().get(0);
39 +        assertThat(conditional1.getCondition(),
contains("http"));
40 +        assertEquals(1, conditional1.getBundles().size());
41 +
42 +        Conditional conditional2 =
feature.getConditional().get(1);
43 +        assertThat(conditional2.getCondition(),
contains("req:osgi.ee;filter:=\"(&(osgi.ee=JavaSE)(!
(version>=1.7)))\""));
```

Fig. 3: Assertion refactoring, removing duplicate assertion and *Assertion Roulette* test smells. Located in the commit 3b42fb5 from Apache Karaf.

example where assertContain is used to remove both *Assertion Roulette* and duplicate assertion test smell. In this case, developers attempt to mitigate the test code's verboseness by refactoring with assertions, which helps to remove the test smells.

Conditional Test Logic. We find that 1 out of 292 (<1%) commits refactor *Conditional Test Logic*. The finding shows that developers deliberately fixed 1.9% of the removed *Unknown Test*. This test smell occurs when the test case's success or failure depends on the assertion method within the control flow blocks and thus not predictable (Meszaros, 2007). A prior survey (Garousi and Küçük, 2018) noted that developers prefer to consider it is smelly or not on a "case-by-case basis". Our study also finds that developers typically do not refactor *Conditional Test Logic*. For the only case that we found, developers refactored the test smell when there are nested conditional statements. In Kafka (*7ebc5da6*), shown in the code snippet below, the developer simplifies the test smell's verbosity with assertion statements. Namely, assertThat() & is() is used to improve the readability of the test logic.

```
1  @Test
2  public void checkTypeInformation() {
3     .....
4   - if(tupleType.isTupleType()) {
5   - if(!((TupleTypeInfo<?>)tupleType).equals(testTupleType)) {
6   -       fail("Tuple type information was not set correctly!");
7   - }
8   - } else {
9   -   fail("Type information was not set to tuple type information!");
10  - }
11  + assertThat(tupleType.isTupleType(), is(true));
12  + assertThat(tupleType, is(equalTo(expectedType)));
13    .....
14 }
```

Although less frequent, we find that developers deliberately refactor specific test smells in 50 out of 292 (17%) studied commits. In particular, *Exception Catch/Throw* and *Sleepy Test* are the two most commonly refactored test smells. Based on the discussion in bug reports and commit messages, we find that developers are often aware of the sub-optimal practice of using Thread.sleep and spent efforts on improving the design of exception handling mechanisms. Even though the number is less, we also find some refactoring of other test smells, such as the *Unknown Test*, *Magic Number*, *Sensitive Equality*, *Conditional Test Logic*, and *Assertion Roulette*.

Test Smell Unaware Refactoring

Most test smells (133/292, 45%) are not removed by developers but are either relocated or deleted as consequences of other refactoring activities. We classify such commits as *Test Smell Unaware Refactoring* since developers were unaware of the test smells and removed them as a by-product of other maintenance tasks, such as refactoring, feature improvement, feature addition, or bug fixing. In our manual analysis, we find that *Test Smell Unaware Refactoring* may affect the test smells in two ways: 1) The test smells are relocated to another codebase, i.e., test smell persistence. 2) The test smells are removed unintentionally due to cascading results of other source code refactoring activities. Below we discuss the two categories in detail.

Test Smell Persistence. We find that for 58 out of 292 (20%) commits, test smells were relocated to other codebase locations. In this case, test code may undergo various refactoring activities such as introducing inheritance (19 commits), extracting method (25 commits), extracting class (10 commits), replacing method with the existing helper (2 commits), and moving method/class (2 commits). For example, in CXF (*31a4a55*), developers applied two refactorings (i.e., extract superclass and pull up method) to the test case *JCacheOAuthDataProviderTest* to extract common test code. However, four existing test smells (i.e., *Assertion Roulette*, *Conditional Test Logic*, *Duplicate Assertion*, and *Magic Number*) in the test code are relocated to the new test case as a result of the refactoring. Namely, developers did not remove the test smells during test code refactoring. In some cases, relocation may magnify test smell's effect. For example, in Hive (*14e92703*), while the developer fixes a bug associated with test failure, the developer extracts a reusable method that explicitly causes a thread to sleep and relocates code to a method in a test utility file. The method was then used in three other test cases, thus magnifying the test smell's impact.

By-Product Removal. We find that for 75 out of 292 commits (25%), test smells are removed by developers as a cascading effect of source code changes. Such maintenance activities include feature improvement, adding features, and bug fixing (i.e., the horizontal view in Table 6). For example, a commit in Accumulo (*9dadca0f*) implements a new feature and refactors the source code using a builder design pattern. As a result of the source code changes, one test smell instance of *eager test* (i.e., calling multiple source code methods in a test

case) is removed since the test case now calls the builder method instead of invoking four distinct methods.

In summary, we find that developers may refactor test code while performing other maintenance activities. Developers may refactor for future maintainability or as a necessary precursor for change in feature requirement. For example, to support new features in the source code, developers may refactor test code to accommodate common logic in test code and apply code reuse. However, in most cases, test smells were unintentionally removed by test code relocation or diffusion as a side-effect of these maintainability tasks. Our findings show that developers are unaware of test smells and may not actively remove test smells as systems evolve.

We find that developers often refactor test code, but they may not directly remove test smells. Many manually studied test smells (133/292, 45%) are relocated or removed unintentionally by developers while refactoring test code.

Other Maintenance Activities

We find that test code may be deleted as a system evolves. The majority of test smell removals are related to test code deletion. 109 out of 292 (37%) commits belong to the category *Other Code Changes*. We classify a commit into this category when the removed test smell results from deleting test code, disabling test (ignoring/commenting out), or reverting a commit.

Test Code Deletion. In our study, we find that for a non-trivial number of commits (97/292 commits, 33%), the test smells are removed because the test code is deleted. Developers may delete a test case when it becomes redundant or obsolete. For example, in Flink (*e671f34*), while porting source code to another file, developers discuss the removal of test cases since the other file already has similar test cases. Developers sometimes also delete test cases when they become obsolete or hard to maintain as the system evolves. For example, in Accumulo (*c265ea5b*), the test code becomes irrelevant since the corresponding features under test are unstable and removed. Therefore, the test smells in the test code are also removed.

Add Comment/@Ignore. 7 out of 292 (2%) commits are related to commenting out or ignoring the test code. This category represents removing test smells as a result of temporarily disabling the test code. In general, we find that developers may comment out the entire test case to bypass test failure. For example, in Kafka (*ca1f18e*), developers commented out the test code to temporarily make the test pass since the test would only work after developers migrate to Java 9. As another example, in Camel (*9ad68066*), the test case is ignored due to test failure caused by a recent upgrade to jetty 9.3. Although the test smell is removed due to commenting or ignoring the test code, the test smell is not addressed. Lastly, we also see cases where the commented out test case was only brought back a few months later. Future studies should

also investigate if such commented out test cases are re-enabled or become a technical debt in the system (Pham and Yang, 2020).
Revert a Commit. 5 out of 292 (<2%) commits are related to reverting the test code. This category represents removing test smells as a result of temporarily reverting software to the previous versions. For example, in Wicket (*266c90037*), the system was reverted due to a defect caused by adding new features. Thus, the newly introduced test smell was also reverted.

> We find that as the system evolves, test code and its associated test smells may be deleted due to the obsoleteness and maintenance difficulty of the source/test code. Developers may also temporarily comment out test cases to bypass test failures caused by recent code changes. However, we see instances where developers only bring the commented out test code back after several months or years.

Summary & Implication. Our manual study shows that, in most cases, developers may not be aware of the test smells. We find that 82.9% of the studied test smells are removed, relocated, or disabled (e.g., commented out) as a by-product of other maintenance activities. During these refactoring activities, developers may relocate the test smell to another test case, and the test smell remains unchanged. In some cases, as discussed, the impact of test smell may become larger, as the test code that contains test smells is extracted to become a utility method. Nevertheless, we still find that developers deliberately removed test smells in 16% of the studied commits. In particular, we find that developers are more likely to remove *Exception Catch/Throw* and *Sleepy Test*. Our finding suggests that, although developers may refactor test code, they often do not deliberately remove test smells. In the next RQ, we further investigate the relationship between test smells and software quality.

3.4 RQ3: What is the relationship between test smells and software quality?

Motivation: Although researchers have made a necessary step towards understanding the maintainability aspects of test smells (Bavota *et al.*, 2015; Bleser *et al.*, 2019; Junior *et al.*, 2020a; Peruma *et al.*, 2019), it is still not clear whether removing test smells has an effect on the quality of software. In the previous RQ, we find that in addition to deliberately resolving test smells, test smells are commonly removed as by-products of other maintenance activities, such as deleting the test code entirely. Regardless of how the test smells are removed or relocated (RQ2), the removed test smells are no longer impacting the source code files. However, it remains unknown whether test smells have any relationship with the code quality. Hence, in this RQ, we aim to understand further the relationship between test smells and software quality, particularly the post-release defect. Our finding may help identify the types of test smells that correlate with a post-release defect and inspire future research that helps developers efficiently address more test smells.

Approach: Our goal is not to predict defects but to study the additive effect of test smell metrics on post-release defects over-controlled metrics using logistic regression models.[3] Logistic regression models are commonly used in prior research to study the effect of various software metrics on post-release defect (Bird *et al.*, 2011; Chen *et al.*, 2012; de Pádua and Shang, 2018). Below, we define the metrics that we use and the model building process. The metrics are extracted from 197 total official releases.

Studied Metrics & Data Collections

- ***Post-Release Defects.*** The post-release defect is our response metric in the regression model. The post-release defect is defined as the defects reported within a fixed time frame after a certain version of a software is released (Moser *et al.*, 2008; Munson and Khoshgoftaar, 1992; Piotrowski and Madeyski, 2020). For each source code file, we label it as defect prone if the file is modified at least once in bug fixing commits within six months after the release of the software system (Zimmermann *et al.*, 2007). Developers in the studied systems are required to enter the issue ID in commit messages. Thus, we first query the issue tracker to obtain a list of bug reporting issues within six-month of each release date. We then find all the bug-fixing commits based on whether the commit messages contain one of the obtained issue IDs. At the end of the step, we obtain the list of source code files that contain post-release defects (e.g., TRUE or FALSE). Finally, if a test file tests a source code file that contains a post-release defect, we label the test file as defect-prone.
- ***Traditional Product and Process Metric.*** Similar to prior studies, we control for traditional product and process metrics in our regression model. Previous studies found that traditional product metrics (e.g., lines of code) and process metrics (e.g., code churn and pre-release defect) are good explainers for post-release defects (Moser *et al.*, 2008; Nagappan and Ball, 2005; Nagappan *et al.*, 2006) and are commonly used as baseline metrics (Bird *et al.*, 2011; Chen *et al.*, 2012; D'Ambros *et al.*, 2010). We collect these metrics at the test-file level and use them as a baseline to build a BASE model. We later add test smell metrics to the BASE model and study whether the test smell metrics may further explain a source code file's defect-proneness. Although our metrics may not represent all of the metrics, they are shown to have a high correlation with other complexity metrics and used for benchmarking in prior proposals of new metrics (Biyani and Santhanam, 1998; Chen *et al.*, 2017; D'Ambros *et al.*, 2010). For the traditional product metric, we used *CLOC* (AlDanial, 2019) to extract the LOC metric in the test file. For traditional process metrics, we use commands "git follows" and "git diff" to extract three different code churn metrics: file churn, code churn, and deleted lines of code. File churn is the number of commits that modified the file. Code churn is the total number of code lines, such as code deletion, addition, and modification. Finally,

[3] Logistic Regression from Lrm R package.

code deletion is the total lines of code deleted. For the other process metric, namely the pre-release defect metric, we follow a similar approach to extracting post-release defects using a six-month time window before one software release.

- ***Coupling Metric.*** In addition to the traditional product and process metrics, we also add two coupling metrics (namely COUPLING) as our controlled metrics to the BASE model to reduce the effects of confounding variables. We measure two coupling metrics, *ts_coupling* (i.e., a test case to source code) and *tt_coupling* (i.e., test cases to test cases), in test cases, which are used in prior studies to assess the quality of test code design (Child *et al.*, 2019). We exclude the dependencies with external frameworks or libraries when calculating the coupling metrics.
- ***Test Smell Product and Process Metrics.*** We consider both test smell product and process metrics. Test smell product metrics (TEST_PRODUCT) are the number of detected test smells present in the system's current release. Test smell process metrics (TEST_PROCESS) are the number of test smells added and removed six months before releasing the system. Note that we extract the two metrics for each type of test smells (18 types in total). Calculating test smell process metrics can be challenging due to file deletion and rename. To address these challenges, we use the "git follow" to keep track of package change and file renaming. Since test smells are detected at different granularities, such as line, method, and class level, we aggregated the test smells at the file-level.

Model Construction

We use a logistic regression model to model post-release defect because it is easier to interpret and is widely used in prior studies (Chen *et al.*, 2017; Harrell Jr, 2015; Kuhn and Johnson, 2013; Nagappan and Ball, 2005). Logistic regression can better isolate (with a predominantly additive effect) the effects of the test smell metrics on explaining post-release defect over the BASE model (i.e., an improvement on the model fitness) (Harrell Jr, 2015). In particular, we build an initial model using the baseline metrics (i.e., traditional process and product metrics and coupling metrics). Then, we build a series of new models to add TEST_PRODUCT and TEST_PROCESS over the BASE model. By studying the explanatory power of a series of models and their additive effects of test smell metrics, we explore whether test smell contributes to a better explanation of post-release defect. We build three models for each studied system:

BASE (LOC+CHURN+PRE+COUPLING): The baseline model uses the traditional product, process, and coupling metrics.
BASE+TEST_PRODUCT: We add TEST_PRODUCT to the BASE model and measure the improvement in the explanatory power over the BASE model.
BASE+TEST_PRODUCT+TEST_PROCESS: The third model measures the combined effect of TEST_PRODUCT and TEST_PROCESS metrics over the BASE model.

For each model that we construct, we first apply data transformation to reduce the data skewness. We follow prior studies using log-transformation on the metrics to normalize the data (Chen *et al.*, 2012; de Pádua and Shang, 2018). Second, we remove the metrics with a zero variance because these metrics do not contribute to the model (i.e., the values are constant). Third, we apply redundancy analysis to drop predictors that can be predicted based on a model composed of all other predictors with an adjusted R2 of higher than 0.9.[4] Since some metrics may be correlated and cause the problem of multicollinearity and overfitting (Harrell Jr, 2015; Jiarpakdee *et al.*, 2018; Wang *et al.*, 2018), we use Variance Inflation Factors (VIFs) to detect the collinearity among the metrics (Kuhn and Johnson, 2013). A high VIF value reflects an increase in the variance due to collinearity in the data. If a metric has a VIF value larger than 10, we remove the metric from the model (Kuhn and Johnson, 2013).[5]

Model Assessment Process

Our goal is not to predict post-release defect, but rather to study the explanatory power of the test smell metrics. Thus, we adopt three different model assessment techniques (probability of defect-proneness, Wald χ^2 test, and area under the curve; AUC) to understand the relationship between test smell and post-release defect.

First, we study the contribution of individual TEST_PRODUCT and TEST_PROCESS metric by looking at proportions of χ^2 for each metric relative to the total χ^2 of the model. χ^2 is a likelihood ratio test to identify how much a metric contributes to the model's fitness. The higher χ^2 indicates a higher explainability of the metric (i.e., more important) in the model (de Pádua and Shang, 2018; Harrell Jr, 2015). Finally, we use AUC, the area under the Receiver Operating Characteristics (ROC), to compare nested logistic regression to capture the relationship between the explanatory metrics and the source code's defect-proneness file (Harrell Jr, 2015). AUC measures the fitness of the model. An increase in AUC when new metrics are added to the model indicates that the new model has a higher ability to capture the relationship and a better fitness (i.e., there is a correlation between the added metrics and defect-proneness, after controlling for the baseline metrics).

Second, we study the effect size of test smell metrics on the probability of defect-proneness (Moser *et al.*, 2008; Shang *et al.*, 2015). To quantify the effect, we set all of the model's metric values to their mean value and record the probability of defect-proneness. Then, we increase the value of the metrics in which we want to measure the effect (i.e., test smell metrics). For each subject metric, we increase the value by 125% and 150% of its mean value and recalculate the probability of defect-proneness after the increase and report the percentage difference. A positive value indicates that increasing the metric's value increases the probability of the post-release defect. A negative value

[4] Redundancy analysis from the Hmisc R package.
[5] VIF analysis from RegClass R package.

indicates that increasing the value of the metric decreases the likelihood of the post-release defect. The intuition behind the analysis is to understand which metric contributes more to the explainability of the software defects while controlling for other metrics (de Pádua and Shang, 2018).

Results: **The explanatory power of test smell metrics in regression models.** *Adding test smell metrics increase the AUC of the model by an average of 8.25% over the BASE model.* Table 9 – 13 show the details of the regression models, where we show the additive effects of test smell features over the baseline metrics. We move the tables to the appendix to make the paper more concise. We show the proportion of χ^2 to understand the importance of including the metric on the model fitness. We show the AUC to understand whether our test smell metric contributes to a higher ability to capture the relationship on the post-release defect over baseline metrics. We find that in all of the models, the AUC increases by 5.1% over the baseline when adding TEST_PRODUCT metrics; and there is around 8.25% increase in AUC over the baseline when adding *both of the* TEST_PRODUCT and TEST_PROCESS metrics. Although the increase is small, we see a consistent result in all the studied systems except Wicket (as discussed in RQ1, Wicket experienced major refactoring in 2019, which may affect the modeling results). Our results also show that adding TEST_PROCESS metrics have only a small increase in the model's explainability after considering the baseline metrics and TEST_PRODUCT. The potential reason may be that, as shown in RQ2, developers may remove or add a test smell as a by-product of other refactoring activities. Therefore, there may be noises in the TEST_PROCESS metrics. Lastly, for the proportion of χ^2, we find that the explainability of test smell metrics varies from system to system. However, we see that test smell metrics such as *Conditional Test Logic*, *Constructor Initialization*, *Exception Catch/Throw*, *Mystery Guest*, *Resource Optimism*, *Assertion Roulette*, *Eager Test*, and *Lazy Test* have higher explainability across the studied systems. In short, these test smells may have a higher correlation with the defect-proneness of source code files.

The effect size of test smell metrics on defect-proneness. *Most test smell metrics have minimal effect on defect-proneness.* The analysis mentioned above shows the explainability of the metrics but not the effect. Hence, we further study the effect of each test smell metric. Table 14– 15 show the effect size of the TEST_PRODUCT and TEST_PROCESS metrics on post-release defects for the studied systems. As discussed in the approach section, we measure the effect size by increasing individual test smell metrics while keeping all other metrics at the mean value. We find that the effect size and direction (i.e., positive or negative) of the effect vary from system to system. However, the effects of most test metrics on the defect-proneness are minimal (i.e., less than 1% increase in the probability of defect-proneness when 150% increases the value of the test smell metric). Compared to TEST_PROCESS metrics, TEST_PRODUCT metrics, in general, have a slightly larger positive relationship with source code defect-proneness. Among all test smell metrics,

we find that *Exception Catch/Throw* and *Conditional Test Logic* show the highest positive relationship in the majority of the studied systems. The findings imply that more *Exception Catch/Throw* and *Conditional Test Logic* in a test case may lead to a higher probability of having a post-release defect in its corresponding source code file. The analysis result on *Exception Catch/Throw* also echoes our finding in RQ2. We found that developers are more likely to refactor *Exception Catch/Throw* when maintaining test code. On the other hand, in RQ2, we only observed one commit that addressed *Conditional Test Logic*. Prior research (Peruma *et al.*, 2019) found that developers do not naturally think of *Conditional Test Logic* as a problem. Hence, future studies are needed to evaluate further the effect of this test smell on software quality. Finally, one possible reason for the high variability in effect size of the TEST_PROCESS metric compared to the TEST_PRODUCT metric could be that many of the test smells were removed as a by-product in the effort to improve test code maintainability (as found in RQ2). Thus, future research on test smell should consider those by-product removals and relocation when designing the study.

Table 7: The comparison of the area under (a ROC) curve for the studied systems. The model is trained using the system in the first column, and AUC is calculated using the system depicted in the remaining columns.

	Accumulo	Bookkeeper	Camel	Cassandra	Cxf	Flink	Groovy	Hadoop	Hive	Kafka	Karaf	Wicket	Zookeeper
Accumulo	0.73	0.61	0.52	0.68	0.62	0.63	0.59	0.60	0.53	0.66	0.59	0.59	0.60
Bookkeeper	0.62	0.87	0.62	0.64	0.58	0.64	0.71	0.54	0.58	0.61	0.61	0.58	0.51
Camel	0.60	0.65	0.75	0.63	0.68	0.64	0.51	0.61	0.52	0.66	0.49	0.69	0.62
Cassandra	0.67	0.66	0.57	0.78	0.69	0.64	0.55	0.65	0.55	0.59	0.53	0.67	0.61
Cxf	0.62	0.50	0.69	0.63	0.78	0.62	0.58	0.71	0.52	0.62	0.58	0.68	0.55
Flink	0.66	0.80	0.65	0.72	0.67	0.77	0.83	0.65	0.59	0.79	0.61	0.73	0.70
Groovy	0.60	0.63	0.53	0.66	0.63	0.64	0.97	0.58	0.51	0.65	0.60	0.57	0.63
Hadoop	0.60	0.63	0.53	0.66	0.63	0.64	0.97	0.58	0.51	0.65	0.60	0.57	0.63
Hive	0.61	0.62	0.67	0.62	0.62	0.64	0.81	0.61	0.67	0.61	0.64	0.67	0.56
Kafka	0.64	0.77	0.55	0.71	0.67	0.73	0.64	0.61	0.59	0.82	0.53	0.64	0.67
Karaf	0.54	0.59	0.54	0.59	0.53	0.57	0.49	0.63	0.54	0.58	0.82	0.60	0.65
Wicket	0.53	0.56	0.49	0.53	0.52	0.51	0.58	0.55	0.52	0.60	0.58	0.82	0.59
Zookeeper	0.51	0.58	0.57	0.57	0.58	0.56	0.73	0.71	0.53	0.52	0.61	0.55	0.83

The comparison of area under (a ROC) curve for the studied systems. *The cross-system AUC is lower than within-system AUC.* We further investigate whether different systems share a similar relationship between test smell and defect proneness (i.e., whether the models are applicable cross-systems). Table 7 shows the results of our cross-system AUC using combined (i.e., product and process) test smell features. In general, we find that the results of cross-system AUC are lower than the within-system AUC. In particular, some models trained on one system (e.g., Accumulo) perform worse when applied on some systems (e.g., AUC is 0.53 when the model is applied on

Camel) but are relatively better when applied on other systems (e.g., AUC is 0.67 when applied on Cassandra and Kafka). Our results show that while different systems have different development characteristics, some systems may have a more similar relationship between test smells and defect-proneness. Future studies are needed to further study effect of test smells across systems from different domains.

The studied test smell metrics increase the AUC of the model by an average of 8.25% over the BASE model. The test smell product metrics such as *Conditional Test Logic*, *Constructor Initialization*, *Exception Catch/Throw*, *Mystery Guest*, *Resource Optimism*, *Assertion Roulette*, *Eager Test*, and *Lazy Test* may have a higher correlation with the defect-proneness, while process metrics have little or no improvements to the model fitness. Contrarily, the effect size of test smell metrics have minimal effect on defect-proneness, and different test smell has a different effect on the defect-proneness of source code files across the studied systems.

4 Implications of our Findings

Table 8 summarizes the results and implications for each research question.

Table 8: Summary of our findings and their implications.

Findings about how test smells evolve overtime	**Implications**
F.1 Although the total number of test smell increases over time, after normalizing by the total number of lines of test code, the test smell density remains relatively stable in most of the 12 studied systems.	**I.1** As software system evolves, test smell will likely co-evolve with amount of added test code. However, our results suggest that developers may allocate some resources in maintaining test code that results in removal of test smells.
Findings about test smell awareness refactorings	**Implications**
F.1 *Sleepy Test* & *Exception Catch/Throw* are the two test smells that developers directly address..	**I.1** Our results may help future research and tool builders to focus on these two test smells for a better recommendation support on addressing test smells.
F.2 Developers sometimes refactor test smell to remove verbose statements. In particular, developers may use better assertion style to remove test smells.	**I.2** There are numerous testing frameworks that offer distinct assertion syntax. However, due to lack of experience and knowledge, developers may sometimes resort to verbose assertions. Future research should investigate refactoring recommendation using better assertion statements.
Findings about test smell unawareness refactorings	**Implications**
F.1 58 out of 292 (20%) commits relocated test smells to another test case after some refactoring and maintenance activities. In such cases, developers pay attention to test code reusability and duplication instead of addressing test smells. Subsequently, we find that in some cases relocation can diffuse the impact of test smells (e.g., relocated to a utility file).	**I.1** Although the maintenance test code has become a prominent task in recent years, for the most part, test smell is not the reason for refactoring, and developers may not pay attention to addressing test smells.
F.2 70 out of 292 (24%) commits remove test smells while working on other maintenance tasks.	**I.2** Test smells are inherent problems, which may hinder test design and comprehension. However, our result shows that developers may not be aware of the test smells. Many test smells are indirectly removed when developers deal with bug fixing or feature enhancement.
Findings about other maintenance activities that removed test smells	**Implications**
F.1 Most test smells are removed due to test code deletion (33%). We find that as test code evolves, there may be substantial instances of ad-hoc manual test code deletions caused by redundant or obsolete test code, which remove test smells as a side-effect.	**I.1** Our result suggests that developers often manually maintain test code by deleting duplicate or obsolete test code, which may be time consuming. Future studies should support the detection of refactoring opportunities or even conduct automated refactoring to reduce maintainability efforts.
F.2 Developer tends to disable (i.e., commenting out or ignoring) test case (2%) to make a test pass.	**I.2** Future studies should further investigate the causes for disabling test cases, and whether disabled test cases become technical debts in the systems (e.g., forget to re-enable) (Pham and Yang, 2020; Spínola *et al.*, 2019).
Findings about relationship between test smell and software quality	**Implications**
F.1 Test smell metrics complement traditional metrics in explaining post-release defects, even though the addition is small (an average of 5.8% increase in AUC). **F.2** The test smells such as *Conditional Test Logic*, *Constructor Initialization*, *Exception Catch/Throw*, *Mystery Guest*, *Resource Optimism*, *Assertion Roulette*, *Eager Test*, and *Lazy Test* have a higher defect explanatory power in the model. **F.3** The effect sizes of most test smell metrics on defect-proneness are small. Among all test smells, *Conditional Test Logic* and *Exception Catch/Throw* have the largest effect on a post-release defect.	**I.1** Our result suggests that test smell metrics have a certain correlation with post-release defect, even though the correlation (i.e., improvement in model fitness) is not large. **I.2** Future studies on test smells may focus more on the above-mentioned test smell due to their higher explainability in post-release defect in the model. **I.3** Future studies should further investigate the effect of *Conditional Test Logic* and *Exception Catch/Throw* on software quality and help practitioners prioritize their effort on addressing test smells.

5 Threats to Validity

External Validity: The studied systems are all open source implemented in Java, so the results may not be generalizable to all systems. To minimize the threat, we study systems that are large in scale, cover various domains, frequently used in commercial settings, and diversify the pool of test code under analysis based on the expertise of the developer. Even though our results are consistent among the studied systems, other developers/systems might exhibit a different awareness level about the test smells. Therefore, future studies must evaluate the results on additional systems and systems implemented in different programming languages.
Internal Validity: There may be confounding metrics that may affect the result of our logistic regression model. To mitigate this, we include baseline metrics, such as lines of test code, code churn, and two coupling metrics (i.e., source code to test code dependencies and test code to test code dependencies) in the model. Moreover, our model does not indicate a causal relationship, but rather that there is a possibility of a relationship that may be further investigated in future research. Furthermore, our study aims to understand the relationship between test smell metrics to software post-release defects by studying the effect of test smells on post-release defects. Therefore, we build a logistic regression model to study the relationship between test smell and post-release defects, because logistic regression models provide better interpretability compared to more advanced machine learning model. Future studies investigating the effect of test smell on prediction performance should study just-in-time prediction and cross-system prediction.
Construct Validity: There may be false positives in the tool, *tsDetector*, that we used for identifying test smells. However, we found that false positive rate to be low in our manual study. We found 12 false positives (4% false positive rate), which is consistent with the number reported in the prior study (Peruma *et al.*, 2019). Moreover, there may be biases in our manual study on characterizing the commits that remove test smells. To minimize the biases, two authors independently inspect every commit and then merge the results. Furthermore, we examine all available software artifacts, such as commit messages, code changes, and bug reports. As for the reasons for removing test smells, many non-technical factors may play a role, such as a lack of knowledge and lack of time. However, in the bug report we analyzed, we did not find that developers mention such non-technical aspects that challenge test code's maintainability. Future studies should further dedicate studies on such non-technical factors. A recent paper Spadini *et al.* (2020) reported a severity threshold for *tsDetector* to make a recommendation when test smells are prevalent. Such results have a low impact on our results because we study test smells removal at a more general level, not only when there are too many tests smells. Finally, in our time series plot (RQ1), we plot the averaged test smell metrics of all systems. To ensure that one system does not overestimate the average, leading to false trends, we verified that the individual systems' time series has the same trend as the average.

6 Related Work

There is ongoing research into the discovery, classification of test smells, and their impact on software quality and maintainability (Athanasiou *et al.*, 2014; Bavota *et al.*, 2015; Garousi and Küçük, 2018; Junior *et al.*, 2020a; Levin and Yehudai, 2017; Spadini *et al.*, 2018; Tufano *et al.*, 2016). In this section, we describe related prior research in two areas: (1) awareness and maintenance of test smells, and (2) software defect modelling.

Awareness and Maintenance of Test Smells. Akiyama (1971) discuss the importance of having a well-designed test code. He argued that well-designed test cases are easier to comprehend and maintain. He proposed that refactoring production code is different from refactoring test code and suggested different types of test smell refactoring operations, such as removing dependencies and making resources unique. Motivated by this work, Deursen *et al.* (2001) introduced 11 catalogs of test smells, which are patterns of poor design decisions associated with test code. Since the proposal of test smells, a study by Tufano *et al.* (2016) surveyed developers' awareness of test smells. The result shows that most developers do not recognize design problems in test code and do not perceive test smells as actual problems. Furthermore, to understand what kind of tool support is required, the study also conducted quantitative research to observe when test smells are introduced and fixed. The result shows that test smells have long survivability (i.e., 100 days) and are mostly introduced the first time the test code is written. Similar to our work, Tufano *et al.* (2017) studied the reasons behind code smells in production code. They also find that code smell is no removed as a by-product of code deletion or comment. Another survey by Junior *et al.* (2020a) suggests that developers' professional experience cannot be considered a root cause for the insertion of test smells in test code. Similarly, we believe that not all test smells may result in problems, and perhaps there may be specific types of test smells that may require more attention as a result of other factors. We also study the reasons that prompt developers to remove test smells. Peruma *et al.* (2019) propose a new catalog of test smells and a detection tool, elucidate a lack of investigation of test smells in Android applications. They concluded that test smells are widely distributed and are similar in both mobile and non-mobile application domains. Subsequently, they surveyed developers' awareness of these detected test smells. They found that developers are often aware of the negative consequences of test smells in the software system. Similarly, our study uses the tool devised by Peruma *et al.* (2019) to mine test smell removing commits from the 12 large-scale software systems. In particular, our study attempts to uncover different types of test smells that developers may pay attention to the most in software development and what may be the reasons for removing test smells. We uncover these by studying the code review artifacts (i.e., bug report, commit message, pull request) and code context (manual code analysis). Consequently, we believe this is a necessary step towards understanding the impact of test smells and building a better test smell refactoring tool that reflects developers' needs. Spadini *et al.* (2020) uses tsDetector to propose a

severity threshold for detection rules. The new thresholds have been determined after investigating developers' perception of test smell severity. Unlike their work, we study the general trend on how developers remove test smells. Using such a threshold may under-estimate situations where developers may remove test smells. Due to the increasing importance of test code maintenance, Garousi and Küçük (2018) conducted a large-scale systematic study to summarize a catalog of 196 test smell instances. However, most of the studied test smells are related to general code smells (e.g., long parameter list, god class, no comments, and bad naming), code smells specific to TCN language, and code smells from grey literature (i.e., blog posts) or difficult to generalize (e.g., complicated setup, long-running test, long test file). Different from their research, we focus on 18 other test smells because they are related to unit testing practices in Java (Peruma *et al.*, 2020) and advocated in xUnit guidelines Meszaros (2007). These 18 test smells have also been highlighted as problematic and extensively studied in prior research in test code maintainability and developers' awareness about test smells Bavota *et al.* (2012); Junior *et al.* (2020a).

Yu *et al.* (2019) is the first work to investigate the process involved in comprehending test code. They surveyed developers' time spent reading and extending the test code at various test case design steps. Although their research place a necessary step towards understanding factors that influence test code comprehension, it is difficult to gain actionable insights for tool support. The study does not provide empirical evidence on the complex characteristics of test code evolution. Motivated by limitations of current repair techniques to design test cases accurately, the study by Pinto *et al.* (2012) analyzes test code evolution in terms of their modification, addition and deletion to elucidate complicated evolution of test cases to suggest better repair techniques in the future. Similarly, we attempt to fill the missing empirical evidence in how developers may remove test smells in practice. We also analyze the relationship between test smell metrics and software quality. Our modeling analysis identifies types of test smells that have a higher relationship with defect-proneness.

Software Defect Modelling. Software defect modeling has been proposed to ensure high quality by understanding the relationship between various software metrics (e.g., lines of code, McCabe's Cyclomatic complexity) and the software defects (Moser *et al.*, 2008; Munson and Khoshgoftaar, 1992). So far, there are two main approaches in software defect modeling. The first is to use quality metrics to predict where the defect may occur, allocating maintenance efforts in the specific code artifacts (Moser *et al.*, 2008; Piotrowski and Madeyski, 2020). Another approach is by studying the relationship between the studied metrics and the probability of a post-release defect (de Pádua and Shang, 2018; Shang *et al.*, 2015; Shihab *et al.*, 2010). The two share a different defect labeling process in the model. SZZ is used in the formal approach to predict defect introducing code changes (Kamei *et al.*, 2016; Rodríguez-Pérez *et al.*, 2020). Since defect modeling at the commit-level may miss relevant bugs (i.e., bugs introduced in one version but not found until much later), we use a more

interpretable process like the post-release defect to label our defect data. The post-release defects provide a different indication of the software quality.

Motivated by their work, numerous researchers have begun to use anti-pattern or code smells as a quality metric for defect modeling. For instance, the work by de Pádua and Shang (2018) uses exception handling anti-pattern to characterize post-release defect. Palomba *et al.* (2014) found that there is a correlation between code smell removal and defect-proneness. Since test smell is becoming an important and ongoing research interest, researchers have also started to investigate test smell metrics in the defect modelling (Qusef *et al.*, 2019; Spadini *et al.*, 2018). In particular, prior researches study the relationship by looking only at the test smell product metric, which is the presence of test smells in software systems (Spadini *et al.*, 2018). However, this may not be an accurate assessment as software evolution involve complex code changes. To fill this gap, Palomba *et al.* (2019) proposed a defect prediction model based on both process and product metrics. They claimed that their combined metrics improve the performance of prediction accuracy. Similarly, we study test process metrics to gain further insights into how the test smell addition and removal in software evolution may provide additional explanatory power to the probability of a post-release defect. We also control for various baseline metrics (e.g., traditional process, product, and coupling metrics) in our model. We found that test smell process metrics provide less explanatory power than test smell product metrics. We also found that some test smells, such as *Conditional Test Logic* and *Exception Catch/Throw*, have a larger correlation with software defect-proneness.

7 Conclusion

First and foremost, the primary value of our research work comes from recognizing the importance of understanding why developers remove test smells and the mechanisms in which they are addressed. We believe this is a necessary corequisite to validate current perception of test smells towards developing a more useful refactoring recommendation tool. Without such knowledge, future studies may progress to propose new test smells and detection tools with minor applicability in the wild and may even hamper software maintenance effort. To that end, we attempt to tackle this problem in three folds. First, we find that developers may allocate resources in the maintenance of test code. The test smell density decreases over time, even though the total number of test smell increases in the software systems. Second, we find that developers are more likely to address a subset of test smells (i.e., *Exception Catch/Throw* and *Sleepy Test*) and the rest were usually removed indirectly as a side-effect of accomplishing other non-trivial maintenance tasks related to fixing bugs or change in feature requirements. Similarly, we identify other code changes besides refactoring that relocate, diffuse, delete, disable and revert test code that caused the removal of test smells. Finally, we apply regression models to understand the relationship between test smell metrics and post-release defect.

After controlling for baseline metrics (i.e., LOC, code churn, pre-release defect, and coupling in test code), we find that test smell metrics provide additional defect explanatory power, although the increase is small. Our model also finds that test smells such as *Exception Catch/Throw* and *Conditional Test Logic* have a larger effect on post-release defect. In summary, our study highlights that developers may allocate resources on maintaining test code, but they often do not address test smells. However, we find that some test smells do have some relationship between post-release defect. Future studies are needed to better assist developers with prioritizing the resources to address test smells and refactoring test code.

References

Akiyama, F. (1971). An example of software system debugging. In C. V. Freiman, J. E. Griffith, and J. L. Rosenfeld, editors, *Information Processing, Proceedings of IFIP, 1971*, pages 353–359. North-Holland.

AlDanial (2019). Count lines of code. `https://github.com/AlDanial/cloc`.

Ali, N. B., Engström, E., Taromirad, M., Mousavi, M. R., Minhas, N. M., Helgesson, D., Kunze, S., and Varshosaz, M. (2019). On the search for industry-relevant regression testing research. *Empirical Software Engineering*, **24**(4), 2020–2055.

Apache (2020). Apache jenkins. `https://builds.apache.org/`. Last accessed April 3 2020.

Athanasiou, D., Nugroho, A., Visser, J., and Zaidman, A. (2014). Test code quality and its relation to issue handling performance. *IEEE Transactions on Software Engineering*, **40**(11), 1100–1125.

Bavota, G., Qusef, A., Oliveto, R., Lucia, A. D., and Binkley, D. W. (2012). An empirical analysis of the distribution of unit test smells and their impact on software maintenance. In *28th IEEE International Conference on Software Maintenance, ICSM*, pages 56–65. IEEE Computer Society.

Bavota, G., Qusef, A., Oliveto, R., De Lucia, A., and Binkley, D. (2012). 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)*, pages 56–65.

Bavota, G., Qusef, A., Oliveto, R., De Lucia, A., and Binkley, D. (2015). Are test smells really harmful? an empirical study. *Empirical Software Engineering*, **20**(4), 1052–1094.

Bird, C., Nagappan, N., Murphy, B., Gall, H., and Devanbu, P. (2011). Don't touch my code!: Examining the effects of ownership on software quality. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, SIGSOFT/FSE '11, pages 4–14.

Biyani, S. and Santhanam, P. (1998). Exploring defect data from development and customer usage on software modules over multiple releases. In *Ninth*

International Symposium on Software Reliability Engineering, ISSRE, pages 316–320. IEEE Computer Society.

Bleser, J. D., Nucci, D. D., and Roover, C. D. (2019). Assessing diffusion and perception of test smells in scala projects. In M. D. Storey, B. Adams, and S. Haiduc, editors, *Proceedings of the 16th International Conference on Mining Software Repositories, MSR*, pages 457–467. IEEE / ACM.

Chen, T., Thomas, S. W., Hemmati, H., Nagappan, M., and Hassan, A. E. (2017). An empirical study on the effect of testing on code quality using topic models: A case study on software development systems. *IEEE Transactions on Reliability*, **66**(3), 806–824.

Chen, T., Shang, W., Nagappan, M., Hassan, A. E., and Thomas, S. W. (2017). Topic-based software defect explanation. *J. Syst. Softw.*, **129**, 79–106.

Chen, T.-H., Thomas, S. W., Nagappan, M., and Hassan, A. (2012). Explaining software defects using topic models. In *Proceedings of the 9th Working Conference on Mining Software Repositories*, MSR '12.

Child, M., Rosner, P., and Counsell, S. (2019). A comparison and evaluation of variants in the coupling between objects metric. *J. Syst. Softw.*, **151**, 120–132.

D'Ambros, M., Lanza, M., and Robbes, R. (2010). An extensive comparison of bug prediction approaches. In J. Whitehead and T. Zimmermann, editors, *Proceedings of the 7th International Working Conference on Mining Software Repositories, MSR 2010 (Co-located with ICSE), Cape Town, South Africa, May 2-3, 2010, Proceedings*, pages 31–41. IEEE Computer Society.

de Pádua, G. B. and Shang, W. (2018). Studying the relationship between exception handling practices and post-release defects. In *Proceedings of the 15th International Conference on Mining Software Repositories, MSR*, pages 564–575.

Deursen, A., Moonen, L. M., Bergh, A., and Kok, G. (2001). Refactoring test code. Technical report, Amsterdam, The Netherlands, The Netherlands.

Eck, M., Palomba, F., Castelluccio, M., and Bacchelli, A. (2019). Understanding flaky tests: the developer's perspective. In M. Dumas, D. Pfahl, S. Apel, and A. Russo, editors, *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE*, pages 830–840. ACM.

Garousi, V. and Küçük, B. (2018). Smells in software test code: A survey of knowledge in industry and academia. *Journal of Systems and Software*, **138**, 52–81.

Harrell Jr, F. E. (2015). *Regression modeling strategies: with applications to linear models, logistic and ordinal regression, and survival analysis*. Springer.

Jiarpakdee, J., Tantithamthavorn, C., and Hassan, A. E. (2018). The impact of correlated metrics on defect models. *CoRR*, **abs/1801.10271**.

Junior, N. S., Soares, L. R., Martins, L. A., and Machado, I. (2020a). A survey on test practitioners' awareness of test smells. *CoRR*, **abs/2003.05613**.

Junior, N. S., Soares, L. R., Martins, L. A., and Machado, I. (2020b). A survey on test practitioners' awareness of test smells. *CoRR*, **abs/2003.05613**.

Kamei, Y., Fukushima, T., McIntosh, S., Yamashita, K., Ubayashi, N., and Hassan, A. E. (2016). Studying just-in-time defect prediction using cross-project models. *Empir. Softw. Eng.*, **21**(5), 2072–2106.

Knuth, D. E. (1981). *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming*. Addison-Wesley, Reading, MA, 2nd edition.

Kuhn, M. and Johnson, K. (2013). *Applied predictive modeling*, volume 26. Springer.

Lam, W., Godefroid, P., Nath, S., Santhiar, A., and Thummalapenta, S. (2019). Root causing flaky tests in a large-scale industrial setting. In D. Zhang and A. Møller, editors, *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA*, pages 101–111. ACM.

Levin, S. and Yehudai, A. (2017). The co-evolution of test maintenance and code maintenance through the lens of fine-grained semantic changes. In *2017 IEEE International Conference on Software Maintenance and Evolution, ICSME*, pages 35–46. IEEE Computer Society.

Luo, Q., Hariri, F., Eloussi, L., and Marinov, D. (2014). An empirical analysis of flaky tests. In S. Cheung, A. Orso, and M. D. Storey, editors, *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22)*, pages 643–653. ACM.

Meszaros, G. (2007). *xUnit test patterns: Refactoring test code*. Pearson Education.

Moser, R., Pedrycz, W., and Succi, G. (2008). A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In W. Schäfer, M. B. Dwyer, and V. Gruhn, editors, *30th International Conference on Software Engineering (ICSE),*, pages 181–190. ACM.

Munson, J. C. and Khoshgoftaar, T. M. (1992). The detection of fault-prone programs. *IEEE Trans. Software Eng.*, **18**(5), 423–433.

Nagappan, N. and Ball, T. (2005). Use of relative code churn measures to predict system defect density. In *Proceedings of the 27th international conference on Software engineering*, pages 284–292.

Nagappan, N., Ball, T., and Zeller, A. (2006). Mining metrics to predict component failures. In L. J. Osterweil, H. D. Rombach, and M. L. Soffa, editors, *28th International Conference on Software Engineering (ICSE),*, pages 452–461. ACM.

Palomba, F., Bavota, G., Penta, M. D., Oliveto, R., and Lucia, A. D. (2014). Do they really smell bad? A study on developers' perception of bad code smells. In *30th IEEE International Conference on Software Maintenance and Evolution*, pages 101–110. IEEE Computer Society.

Palomba, F., Nucci, D. D., Panichella, A., Oliveto, R., and Lucia, A. D. (2016). On the diffusion of test smells in automatically generated test code: an empirical study. In *Proceedings of the 9th International Workshop on Search-Based Software Testing, SBST@ICSE*, pages 5–14. ACM.

Palomba, F., Zanoni, M., Fontana, F. A., Lucia, A. D., and Oliveto, R. (2019). Toward a smell-aware bug prediction model. *IEEE Trans. Software Eng.*, **45**(2), 194–218.

Peruma, A., Almalki, K., Newman, C. D., Mkaouer, M. W., Ouni, A., and Palomba, F. (2019). 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, pages 193–202.

Peruma, A., Almalki, K., Newman, C. D., Mkaouer, M. W., Ouni, A., and Palomba, F. (2020). tsdetect: An open source test smells detection tool. In *Proceedings of the 2020 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, New York, NY, USA. Association for Computing Machinery.

Pham, T. M.-T. and Yang, J. (2020). The secret life of commented-out source code. In *28th IEEE/ACM International Conference on Program Comprehension, ICSE*.

Pinto, L. S., Sinha, S., and Orso, A. (2012). Understanding myths and realities of test-suite evolution. In W. Tracz, M. P. Robillard, and T. Bultan, editors, *20th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-20)*, page 33. ACM.

Piotrowski, P. and Madeyski, L. (2020). Software defect prediction using bad code smells: A systematic literature review. In *Data-Centric Business and Applications*, pages 77–99.

Qusef, A., Elish, M. O., and Binkley, D. W. (2019). An exploratory study of the relationship between software test smells and fault-proneness. *IEEE Access*, **7**, 139526–139536.

Rahman, F. and Devanbu, P. T. (2011). Ownership, experience and defects: a fine-grained study of authorship. In R. N. Taylor, H. C. Gall, and N. Medvidovic, editors, *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*, pages 491–500. ACM.

Rodríguez-Pérez, G., Robles, G., Serebrenik, A., Zaidman, A., Germán, D. M., and González-Barahona, J. M. (2020). How bugs are born: a model to identify how bugs are introduced in software components. *Empir. Softw. Eng.*, **25**(2), 1294–1340.

Shamshiri, S., Rojas, J. M., Galeotti, J. P., Walkinshaw, N., and Fraser, G. (2018). How do automatically generated unit tests influence software maintenance? In *11th IEEE International Conference on Software Testing, Verification and Validation, ICST*, pages 250–261. IEEE Computer Society.

Shang, W., Nagappan, M., and Hassan, A. E. (2015). Studying the relationship between logging characteristics and the code quality of platform software. *Empirical Software Engineering*, **20**(1), 1–27.

Shi, A., Bell, J., and Marinov, D. (2019). Mitigating the effects of flaky tests on mutation testing. In D. Zhang and A. Møller, editors, *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA*, pages 112–122. ACM.

Shihab, E., Jiang, Z. M., Ibrahim, W. M., Adams, B., and Hassan, A. E. (2010). Understanding the impact of code and process metrics on post-

release defects: A case study on the eclipse project. In *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, page 4. ACM.

Spadini, D., Palomba, F., Zaidman, A., Bruntink, M., and Bacchelli, A. (2018). On the relation of test smells to software code quality. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 1–12.

Spadini, D., Schvarcbacher, M., Oprescu, A., Bruntink, M., and Bacchelli, A. (2020). Investigating severity thresholds for test smells. In S. Kim, G. Gousios, S. Nadi, and J. Hejderup, editors, *MSR '20: 17th International Conference on Mining Software Repositories, Seoul, Republic of Korea, 29-30 June, 2020*, pages 311–321. ACM.

Spínola, R. O., Zazworka, N., Vetro, A., Shull, F., and Seaman, C. B. (2019). Understanding automated and human-based technical debt identification approaches-a two-phase study. *J. Braz. Comp. Soc.*, **25**(1), 5:1–5:21.

Tsantalis, N., Mansouri, M., Eshkevari, L. M., Mazinanian, D., and Dig, D. (2018). Accurate and efficient refactoring detection in commit history. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, pages 483–494, New York, NY, USA. ACM.

Tufano, M., Palomba, F., Bavota, G., Penta, M. D., Oliveto, R., Lucia, A. D., and Poshyvanyk, D. (2016). An empirical investigation into the nature of test smells. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 4–15.

Tufano, M., Palomba, F., Bavota, G., Oliveto, R., Penta, M. D., Lucia, A. D., and Poshyvanyk, D. (2017). When and why your code starts to smell bad (and whether the smells go away). *IEEE Trans. Software Eng.*, **43**(11), 1063–1088.

Van Deursen, A., Moonen, L., Van Den Bergh, A., and Kok, G. (2001). Refactoring test code. In *Proceedings of the 2nd international conference on extreme programming and flexible processes in software engineering (XP2001)*, pages 92–95.

Wang, S., Chen, T.-H., and Hassan, A. E. (2018). Understanding the factors for fast answers in technical q&a websites. *Empirical Software Engineering*, **23**(3), 1552–1593.

Yu, C. S., Treude, C., and Aniche, M. F. (2019). Comprehending test code: An empirical study. *CoRR*, **abs/1907.13365**.

Zaidman, A., Rompaey, B. V., Demeyer, S., and v. Deursen, A. (2008). Mining software repositories to study co-evolution of production test code. In *2008 1st International Conference on Software Testing, Verification, and Validation*, pages 220–229.

Zeller, A. (2009). *Why Programs Fail - A Guide to Systematic Debugging, 2nd Edition*. Academic Press.

Zhao, X., Liang, J., and Dang, C. (2019). A stratified sampling based clustering algorithm for large-scale data. *Knowl. Based Syst.*, **163**, 416–428.

Zimmermann, T., Premraj, R., and Zeller, A. (2007). Predicting defects for eclipse. In *Proceedings of the Third International Workshop on Predictor*

Models in Software Engineering, PROMISE 07, page 9.

8 Appendix

Table 9: The statistics of the regression models showing additive defect explainability of PD(TEST_PRODUCT) + PR(TEST_PROCESS) metrics over the BASE(LOC+CHURNS+PRE+COUPLING). Tot. χ^2 shows the total explanatory power of the studied model. We also show the proportion of χ^2 contributed by each metric.

project	modelType	Metrics	χ^2	Tot. χ^2	AUC (%)
Camel	BASE	LOC	76.33%	276.76	0.74
		pre_bug	4.76%		
		codeChurn	7.79%		
		ts_coupling	11.12%		
	BASE+PD	ts_coupling	34.6%	88.98	0.75 (1.3%)
		Assertion.Roulette	15.11%		
		Conditional.Test.Logic	6.02%		
		General.Fixture	8.2%		
		Mystery.Guest	14.6%		
		Redundant.Assertion	8.01%		
		Duplicate.Assert	7.82%		
		Resource.Optimism	5.64%		
	BASE+PD+PR	ts_coupling	31.38%	104.29	0.75 (1.6%)
		Assertion.Roulette	12.57%		
		Conditional.Test.Logic	5.59%		
		General.Fixture	7.23%		
		Mystery.Guest	12.42%		
		Redundant.Assertion	5.82%		
		Duplicate.Assert	5.81%		
		Resource.Optimism	4.86%		
		Print.Statement.Added	4.47%		
		Print.Statement.Removed	4.86%		
		Eager.Test.Removed	4.99%		
Cassandra	BASE	LOC	43.38%	228.50	0.75
		pre_bug	33.02%		
		codeChurn	6.37%		
		fileChurn	8.95%		
		ts_coupling	5.72%		
		tt_coupling	2.57%		
	BASE+PD	ts_coupling	69.01%	18.93	0.76 (1.1%)
		tt_coupling	30.99%		
	BASE+PD+PR	ts_coupling	30.73%	42.50	0.78 (3.3%)
		tt_coupling	13.8%		
		IgnoredTest.Added	11.15%		
		Conditional.Test.Logic.Removed	33.06%		
		Exception.Catching.Throwing.Removed	11.25%		
Flink	BASE	LOC	46.04%	1396.14	0.76
		pre_bug	35.08%		
		codeChurn	6.61%		
		deletedLine	0.29%		
		fileChurn	10.41%		
		ts_coupling	1.56%		
	BASE+PD	ts_coupling	19.61%	111.28	0.77 (1.0%)
		Conditional.Test.Logic	9.81%		
		Exception.Catching.Throwing	4.96%		
		General.Fixture	12.99%		
		Mystery.Guest	22.05%		
		Lazy.Test	3.62%		
		Unknown.Test	20.69%		
		Magic.Number.Test	6.27%		
	BASE+PD+PR	ts_coupling	16.27%	134.14	0.77 (1.5%)
		Conditional.Test.Logic	8.14%		
		Exception.Catching.Throwing	4.12%		
		General.Fixture	10.77%		
		Mystery.Guest	18.29%		
		Lazy.Test	3.0%		
		Unknown.Test	17.16%		
		Magic.Number.Test	5.2%		
		EmptyTest.Added	4.28%		
		Mystery.Guest.Added	3.67%		
		Duplicate.Assert.Removed	3.08%		
		Magic.Number.Test.Removed	6.01%		

Table 10: The statistics of the regression models showing additive defect explainability of PD(TEST_PRODUCT) + PR(TEST_PROCESS) metrics over the BASE(LOC+CHURNS+PRE+COUPLING). Tot. χ^2 shows the total explanatory power of the studied model. We also show the proportion of χ^2 contributed by each metric.

project	modelType	Metrics	χ^2	Tot. χ^2	AUC (%)
Accumulo	BASE	LOC	33.05%	188.85	0.68
		pre_bug	20.8%		
		deletedLine	21.54%		
		fileChurn	8.52%		
		ts_coupling	13.57%		
		tt_coupling	2.52%		
	BASE+PD	ts_coupling	40.5%	63.25	0.71 (3.9%)
		tt_coupling	7.52%		
		Conditional.Test.Logic	6.11%		
		Constructor.Initialization	16.76%		
		Redundant.Assertion	6.75%		
		Eager.Test	6.17%		
		Duplicate.Assert	7.07%		
	BASE+PD+PR	ts_coupling	31.71%	80.79	0.73 (6.4%)
		tt_coupling	5.89%		
		Conditional.Test.Logic	4.78%		
		Constructor.Initialization	13.12%		
		Redundant.Assertion	5.28%		
		Eager.Test	4.83%		
		Duplicate.Assert	5.54%		
		Duplicate.Assert.Added	4.95%		
		Unknown.Test.Added	10.78%		
		Eager.Test.Removed	5.98%		
Bookkeeper	BASE	LOC	56.96%	177.81	0.8
		pre_bug	5.12%		
		codeChurn	22.63%		
		fileChurn	4.8%		
		ts_coupling	10.48%		
	BASE+PD	ts_coupling	25.04%	74.40	0.84 (5.3%)
		Assertion.Roulette	9.16%		
		Constructor.Initialization	20.52%		
		Lazy.Test	5.54%		
		Unknown.Test	23.86%		
		Resource.Optimism	15.89%		
	BASE+PD+PR	ts_coupling	16.61%	112.12	0.87 (8.8%)
		Assertion.Roulette	6.08%		
		Constructor.Initialization	13.62%		
		Lazy.Test	3.67%		
		Unknown.Test	15.83%		
		Resource.Optimism	10.54%		
		Exception.Catching.Throwing.Added	4.4%		
		Lazy.Test.Added	3.75%		
		Unknown.Test.Added	7.85%		
		Magic.Number.Test.Added	5.82%		
		General.Fixture.Removed	3.85%		
		Sleepy.Test.Removed	7.97%		

Table 11: The statistics of the regression models showing additive defect explainability of PD(TEST_PRODUCT) + PR(TEST_PROCESS) metrics over the BASE(LOC+CHURNS+PRE+COUPLING). Tot. χ^2 shows the total explanatory power of the studied model. We also show the proportion of χ^2 contributed by each metric.

project	modelType	Metrics	χ^2	Tot χ^2	AUC (%)
Hive	BASE	LOC	72.11%	500.82	0.64
		pre_bug	1.76%		
		codeChurn	6.96%		
		ts_coupling	5.67%		
		tt_coupling	13.5%		
	BASE+PD	ts_coupling	14.86%	191.14	0.66 (1.6%)
		tt_coupling	35.38%		
		Conditional.Test.Logic	2.58%		
		EmptyTest	3.62%		
		General.Fixture	9.66%		
		Lazy.Test	12.53%		
		Duplicate.Assert	4.16%		
		Unknown.Test	9.21%		
		IgnoredTest	2.09%		
		Resource.Optimism	5.91%		
	BASE+PD+PR	ts_coupling	10.94%	259.52	0.67 (3.3%)
		tt_coupling	26.05%		
		Conditional.Test.Logic	1.9%		
		EmptyTest	2.67%		
		General.Fixture	7.12%		
		Lazy.Test	9.23%		
		Duplicate.Assert	3.07%		
		Unknown.Test	6.78%		
		IgnoredTest	1.54%		
		Resource.Optimism	4.35%		
		Conditional.Test.Logic.Added	1.56%		
		Mystery.Guest.Added	3.57%		
		Duplicate.Assert.Added	5.94%		
		IgnoredTest.Added	1.84%		
		Redundant.Assertion.Removed	2.27%		
		Sensitive.Equality.Removed	2.84%		
		Eager.Test.Removed	2.08%		
		Duplicate.Assert.Removed	1.62%		
		Unknown.Test.Removed	3.15%		
		Magic.Number.Test.Removed	1.48%		
Wicket	BASE	LOC	89.35%	103.02	0.78
		fileChurn	6.56%		
		tt_coupling	4.09%		
	BASE+PD	tt_coupling	14.08%	29.95	0.8 (2.9%)
		Conditional.Test.Logic	48.84%		
		Eager.Test	23.95%		
		Unknown.Test	13.13%		
	BASE+PD+PR	Conditional.Test.Logic	22.75%	48.82	0.82 (5.6%)
		Eager.Test	12.09%		
		Unknown.Test	7.92%		
		Assertion.Roulette.Added	30.48%		
		Conditional.Test.Logic.Added	17.51%		
		Exception.Catching.Throwing.Added	9.26%		
Zookeeper	BASE	LOC	50.8%	79.69	0.79
		pre_bug	36.45%		
		codeChurn	12.75%		
	BASE+PD	Resource.Optimism	100.0%	4.90	0.8 (1.7%)
	BASE+PD+PR	Duplicate.Assert.Removed	25.57%	16.76	0.83 (5.4%)
		Unknown.Test.Removed	74.43%		

Table 12: The statistics of the regression models showing additive defect explainability of PD(TEST_PRODUCT) + PR(TEST_PROCESS) metrics over the BASE(LOC+CHURNS+PRE+COUPLING). Tot. χ^2 shows the total explanatory power of the studied model. We also show the proportion of χ^2 contributed by each metric.

project	modelType	Metrics	χ^2	Tot χ^2	AUC (%)
Kafka	BASE	LOC	58.84%	848.39	0.8
		pre_bug	28.83%		
		codeChurn	2.0%		
		fileChurn	4.69%		
		ts_coupling	4.52%		
		tt_coupling	1.13%		
	BASE+PD	ts_coupling	31.14%	123.05	0.81 (1.6%)
		tt_coupling	7.76%		
		Exception.Catching.Throwing	19.13%		
		Mystery.Guest	10.74%		
		Redundant.Assertion	4.45%		
		Lazy.Test	9.81%		
		Duplicate.Assert	6.8%		
		Unknown.Test	4.14%		
		Magic.Number.Test	6.02%		
	BASE+PD+PR	ts_coupling	28.13%	136.23	0.82 (2.6%)
		tt_coupling	7.01%		
		Exception.Catching.Throwing	17.28%		
		Mystery.Guest	9.7%		
		Redundant.Assertion	4.02%		
		Lazy.Test	8.86%		
		Duplicate.Assert	6.14%		
		Unknown.Test	3.74%		
		Magic.Number.Test	5.44%		
		Exception.Catching.Throwing.Added	3.57%		
		Exception.Catching.Throwing.Removed	2.82%		
		Print.Statement.Removed	3.28%		
Karaf	BASE	LOC	59.15%	10.26	0.64
		tt_coupling	40.85%		
	BASE+PD	tt_coupling	9.8%	42.78	0.75 (14.4%)
		General.Fixture	14.16%		
		Print.Statement	16.85%		
		Sleepy.Test	47.66%		
		Unknown.Test	11.53%		
	BASE+PD+PR	tt_coupling	6.16%	72.25	0.82 (28.4%)
		General.Fixture	8.53%		
		Print.Statement	9.76%		
		Sleepy.Test	30.24%		
		Unknown.Test	6.33%		
		Eager.Test.Added	5.87%		
		IgnoredTest.Added	9.52%		
		Magic.Number.Test.Added	9.87%		
		General.Fixture.Removed	13.73%		
Hadoop	BASE	LOC	100.0%	6.72	0.82
	BASE+PD	Duplicate.Assert	100.0%	7.21	0.96 (14.1%)
	BASE+PD+PR	Duplicate.Assert	50.29%	14.74	0.97 (17.7%)
		Duplicate.Assert.Added	49.71%		

Table 13: The statistics of the regression models showing additive defect explainability of PD(TEST_PRODUCT) + PR(TEST_PROCESS) metrics over the BASE(LOC+CHURNS+PRE+COUPLING). Tot. χ^2 shows the total explanatory power of the studied model. We also show the proportion of χ^2 contributed by each metric.

project	modelType	Metrics	χ^2	Tot χ^2	AUC (%)
Cxf	BASE	LOC	70.8%	217.57	0.74
		pre_bug	17.85%		
		deletedLine	3.51%		
		fileChurn	3.72%		
		ts_coupling	1.84%		
		tt_coupling	2.27%		
	BASE+PD	ts_coupling	9.09%	44.10	0.77 (3.4%)
		tt_coupling	11.19%		
		Assertion.Roulette	47.17%		
		Constructor.Initialization	12.22%		
		Exception.Catching.Throwing	20.34%		
	BASE+PD+PR	ts_coupling	5.97%	67.19	0.78 (5.0%)
		tt_coupling	7.34%		
		Assertion.Roulette	30.96%		
		Constructor.Initialization	8.02%		
		Exception.Catching.Throwing	13.35%		
		Sensitive.Equality.Added	10.22%		
		Exception.Catching.Throwing.Removed	8.44%		
		Eager.Test.Removed	9.34%		
		IgnoredTest.Removed	6.36%		
Groovy	BASE	LOC	100.0%	6.72	0.82
	BASE+PD	Duplicate.Assert	100.0%	7.21	0.96 (14.1%)
	BASE+PD+PR	Duplicate.Assert	50.29%	14.74	0.97 (17.7%)
		Duplicate.Assert.Added	49.71%		

Table 14: The effect size of the test smell metrics on post-release defects. Effect is measured by setting the subject metric to 110 % and 150 % of it mean value, while other metrics are kept at their mean values. Bolded numbers indicate a positive increase in effect.

project	Metric Group	Individual Metric	110 % ↑	150 % ↑
Accumulo	TEST_PRODUCT	Conditional.Test.Logic	**0.12 %**	**0.62 %**
		Constructor.Initialization	-0.07 %	-0.34 %
		Redundant.Assertion	-0.04 %	-0.18 %
		Eager.Test	-0.4 %	-1.97 %
		Duplicate.Assert	-0.18 %	-0.91 %
	TEST_PROCESS	Eager.Test.Removed	-0.01 %	-0.04 %
		Duplicate.Assert.Added	**0.01 %**	**0.03 %**
		Unknown.Test.Added	-0.01 %	-0.03 %
Hive	TEST_PRODUCT	Conditional.Test.Logic	**0.08 %**	**0.41 %**
		EmptyTest	**0.01 %**	**0.03 %**
		General.Fixture	**0.05 %**	**0.27 %**
		Lazy.Test	-0.25 %	-1.22 %
		Duplicate.Assert	**0.05 %**	**0.26 %**
		Unknown.Test	**0.12 %**	**0.58 %**
		IgnoredTest	**0.01 %**	**0.06 %**
		Resource.Optimism	**0.07 %**	**0.33 %**
	TEST_PROCESS	Redundant.Assertion.Removed	-0.01 %	-0.01 %
		Sensitive.Equality.Removed	**0.01 %**	**0.04 %**
		Eager.Test.Removed	**0.01 %**	**0.01 %**
		Duplicate.Assert.Removed	**0.01 %**	**0.01 %**
		Unknown.Test.Removed	**0.06 %**	**0.29 %**
		Magic.Number.Test.Removed	-0.01 %	-0.06 %
		Conditional.Test.Logic.Added	-0.01 %	-0.01 %
		Mystery.Guest.Added	**0.01 %**	**0.03 %**
		Duplicate.Assert.Added	**0.01 %**	**0.04 %**
		IgnoredTest.Added	**0.01 %**	**0.03 %**
Wicket	TEST_PRODUCT	Conditional.Test.Logic	**0.01 %**	**0.01 %**
		Eager.Test	**0.01 %**	**0.03 %**
		Unknown.Test	**0.01 %**	**0.01 %**
	TEST_PROCESS	Assertion.Roulette.Added	-0.01 %	-0.01 %
		Conditional.Test.Logic.Added	-0.01 %	-0.01 %
		Exception.Catching.Throwing.Added	**0.01 %**	**0.01 %**
Cassandra	TEST_PROCESS	Conditional.Test.Logic.Removed	-0.01 %	-0.03 %
		Exception.Catching.Throwing.Removed	**0.01 %**	**0.07 %**
		IgnoredTest.Added	-0.01 %	-0.01 %
Bookkeeper	TEST_PRODUCT	Assertion.Roulette	-0.17 %	-0.82 %
		Constructor.Initialization	**0.33 %**	**1.81 %**
		Lazy.Test	-0.22 %	-1.01 %
		Unknown.Test	-0.09 %	-0.42 %
		Resource.Optimism	**0.07 %**	**0.36 %**
	TEST_PROCESS	General.Fixture.Removed	-0.01 %	-0.02 %
		Sleepy.Test.Removed	**0.01 %**	**0.02 %**
		Exception.Catching.Throwing.Added	**0.01 %**	**0.01 %**
		Lazy.Test.Added	-0.03 %	-0.14 %
		Unknown.Test.Added	-0.03 %	-0.17 %
		Magic.Number.Test.Added	-0.01 %	-0.01 %
Camel	TEST_PRODUCT	Assertion.Roulette	**0.01 %**	**0.01 %**
		Conditional.Test.Logic	-0.01 %	-0.01 %
		General.Fixture	-0.01 %	-0.01 %
		Mystery.Guest	**0.01 %**	**0.01 %**
		Redundant.Assertion	**0.01 %**	**0.01 %**
		Duplicate.Assert	**0.01 %**	**0.01 %**
		Resource.Optimism	**0.01 %**	**0.01 %**
	TEST_PROCESS	Print.Statement.Removed	**0.01 %**	**0.01 %**
		Eager.Test.Removed	-0.01 %	-0.01 %
		Print.Statement.Added	-0.01 %	-0.01 %
Groovy	TEST_PRODUCT	Duplicate.Assert	-0.01 %	-0.01 %
	TEST_PROCESS	Duplicate.Assert.Added	**0.01 %**	**0.01 %**
Karaf	TEST_PRODUCT	General.Fixture	**0.01 %**	**0.01 %**
		Print.Statement	**0.01 %**	**0.01 %**
		Sleepy.Test	**0.01 %**	**0.01 %**
		Unknown.Test	-0.01 %	-0.02 %
	TEST_PROCESS	General.Fixture.Removed	-0.01 %	-0.01 %
		Eager.Test.Added	**0.01 %**	**0.01 %**
		IgnoredTest.Added	-0.01 %	-0.01 %
		Magic.Number.Test.Added	**0.01 %**	**0.01 %**
Hadoop	TEST_PRODUCT	Duplicate.Assert	-0.01 %	-0.01 %
	TEST_PROCESS	Duplicate.Assert.Added	**0.01 %**	**0.01 %**

Table 15: The effect size of the test smell metrics on post-release defects. Effect is measured by setting the subject metric to 110 % and 150 % of it mean value, while other metrics are kept at their mean values. Bolded numbers indicate a positive increase in effect.

project	Metric Group	Individual Metric	110 % ↑	150 % ↑
Cxf	TEST_PRODUCT	Assertion.Roulette	**0.01 %**	**0.08 %**
		Constructor.Initialization	-0.01 %	-0.03 %
		Exception.Catching.Throwing	**0.02 %**	**0.11 %**
	TEST_PROCESS	Exception.Catching.Throwing.Removed	**0.01 %**	**0.01 %**
		Eager.Test.Removed	-0.01 %	-0.02 %
		IgnoredTest.Removed	**0.01 %**	**0.01 %**
		Sensitive.Equality.Added	-0.01 %	-0.01 %
Flink	TEST_PRODUCT	Conditional.Test.Logic	**0.04 %**	**0.22 %**
		Exception.Catching.Throwing	-0.12 %	-0.57 %
		General.Fixture	**0.01 %**	**0.07 %**
		Mystery.Guest	**0.02 %**	**0.11 %**
		Lazy.Test	-0.07 %	-0.34 %
		Unknown.Test	-0.06 %	-0.31 %
		Magic.Number.Test	-0.05 %	-0.27 %
	TEST_PROCESS	Duplicate.Assert.Removed	-0.01 %	-0.02 %
		Magic.Number.Test.Removed	-0.01 %	-0.01 %
		EmptyTest.Added	-0.01 %	-0.02 %
		Mystery.Guest.Added	**0.01 %**	**0.01 %**
Zookeeper	TEST_PROCESS	Duplicate.Assert.Removed	-0.01 %	-0.01 %
		Unknown.Test.Removed	-0.01 %	-0.02 %
Kafka	TEST_PRODUCT	Exception.Catching.Throwing	**0.36 %**	**1.81 %**
		Mystery.Guest	**0.06 %**	**0.31 %**
		Redundant.Assertion	-0.04 %	-0.2 %
		Lazy.Test	-0.9 %	-4.32 %
		Duplicate.Assert	**0.17 %**	**0.84 %**
		Unknown.Test	-0.06 %	-0.28 %
		Magic.Number.Test	**0.33 %**	**1.66 %**
	TEST_PROCESS	Exception.Catching.Throwing.Removed	-0.01 %	-0.05 %
		Print.Statement.Removed	-0.03 %	-0.16 %
		Exception.Catching.Throwing.Added	**0.03 %**	**0.14 %**