Revisiting Test Smells in Automatically Generated Tests: Limitations, Pitfalls, and Opportunities

Panichella, A.; Panichella, Sebastiano; Fraser, Gordon; Sawant, Anand Ashok; Hellendoorn, Vincent J.

DOI
10.1109/ICSME46990.2020.00056

Publication date
2020

Document Version
Accepted author manuscript

Published in
Proceedings - 2020 IEEE International Conference on Software Maintenance and Evolution, ICSME 2020

Revisiting Test Smells in Automatically Generated Tests: Limitations, Pitfalls, and Opportunities

Annibale Panichella
Delft University of Technology
Delft, The Netherlands
A.Panichella@tudelft.nl

Sebastiano Panichella
Zurich University of Applied Science
Zurich, Switzerland
sebastiano.panichella@zhaw.ch

Gordon Fraser
University of Passau
Passau, Germany
Gordon.Fraser@uni-passau.de

Anand Ashok Sawant
University of California Davis
Davis, USA
asawant@ucdavis.edu

Vincent J. Hellendoorn
University of California Davis
Davis, USA
vhellendoorn@ucdavis.edu

***Abstract*—Test smells attempt to capture design issues in test code that reduce their maintainability. Previous work found such smells to be highly common in automatically generated test-cases, but based this result on specific static detection rules; although these are based on the original definition of "test smells", a recent empirical study showed that developers perceive these as overly strict and non-representative of the maintainability and quality of test suites. This leads us to investigate how effective such test smell detection tools are on automatically generated test suites. In this paper, we build a dataset of 2,340 test cases automatically generated by EvoSuite for 100 Java classes. We performed a multi-stage, cross-validated manual analysis to identify six types of test smells and label their instances. We benchmark the performance of two test smell detection tools: one widely used in prior work, and one recently introduced with the express goal to match developer perceptions of test smells. Our results show that these test smell detection strategies poorly characterized the issues in automatically generated test suites; the older tool's detection strategies, especially, misclassified over 70% of test smells, both missing real instances (false negatives) and marking many smell-free tests as smelly (false positives). We identify common patterns in these tests that can be used to improve the tools, refine and update the definition of certain test smells, and highlight as of yet uncharacterized issues. Our findings suggest the need for (i) more appropriate metrics to match development practice; and (ii) more accurate detection strategies, to be evaluated primarily in industrial contexts.**

***Index Terms*—Test Generation, Test Smells, Software Quality**

I. Introduction

An ambitious goal of software engineering research is to automatically assess the quality and maintainability of test code. Automatically identifying this is challenging [1], [2], as the notion of quality is both context-sensitive and subject to change over time. A common approach is to focus on those parts of a program that clearly violate certain well-established rules informed by practice; *e.g.*, that a single test method should not test multiple requirements [3], [4]. Ideally, developers can be alerted to such issues automatically and they can be resolved through meaning-preserving refactorings – which is the goal of "test smell" detection [1].

Contemporary test smell detection relies on a well-defined and empirically grounded set of "smells" that reflect improper testing practice. The canonical catalog of these smells was proposed by van Deursen *et al.* [1], who categorize 11 smell types with the corresponding refactorings to resolve them. Prior work has created tools that automatically mark these smells in test code [5]. This is no small chore: many of these require complex inference about the code, *e.g.*, detecting "indirect testing" [1] requires knowing whether (unmocked) calls to other classes constitute "testing" those classes.

Heuristic (static) rule-based detection of test smells is widely used, but over-reliance on such rules risks missing and mischaracterizing issues as testing practices change over time. Recent work showed that test smells are especially prevalent in automatically generated test-cases [6]; however, the heuristics defined typically apply to the goal of detecting smells in manually written test cases, which may not be directly transferrable to automatically generated test suites, as those tests are often markedly different in structure and semantics from their human-written counterparts. Furthermore, while these rules have been designed based on the original definition of "test smells", a recent empirical study showed that developers do not perceive these as reflective of test suite quality and maintainability, finding them overly strict [2].

Considering this challenge of automatically detecting test smells, the non-trivial gap to assessing automatically generated test suites, and the changes in testing practice since the original categorization, we argue that a critical reassessment of test smells is due, specifically focussing on automatically generated test cases. In this work, we manually annotated test smells in 100 test suites spanning 2,340 test cases. This paper provides the following contributions:

Internal Validity: *How widespread are test smells in automatically generated test cases?* We find that EvoSuite [7], a representative test generator, effectively abstracts away many resources (*e.g.*, database and file access), and applies mocking where necessary. However, it still introduces a substantial number of indirect tests and often generates tests that check

a wide range of behaviors simultaneously. We suggest that a notion of semantic objective is needed to refine its focus.

Tool Validity: *How accurate are automated tools in detecting test smells in automatically generated tests?* Comparing our manual annotations against two popular detection tools, we find a substantial rate of misclassifications. Many of these stem from discrepancies between the heuristics and the peculiarities of EVOSUITE's tests. Notably, prior work did not detect many of these issues in their own manual validation, raising concerns about self-evaluations of the validity of heuristics as opposed to industry-based evaluations – a common practice in our field.

External Validity: *How well do test smells reflect real problems in generated test suites?* We highlight how several smells rarely indicated issues, either because of advances in testing frameworks (*esp.* "Assertion Roulette") or because they encode a matter of preference (*e.g.*, "Eager Test"). In turn, we highlight several issues not yet described by any smells, particularly reflecting atypical patterns found in the tests produced automatically.

II. BACKGROUND

A. Test Smells

The notion of code smells goes back to Fowler's book [8]. Code that smells is not necessarily faulty, but may contain quality issues that inhibit maintenance or lead to a bug at some point [9]. The notion of code smells was later extended to test code [1]. To help developers avoid such issues, several tools have been introduced to automatically flag smelly test code [2], [3]. This has in turn enabled researchers to empirically study the prevalence of test smells and their effects, confirming that test smells are not only common in open source and industrial software, but also have a strong negative impact on program comprehension and maintenance [2], [5], [10].

B. Test Case Generation

Writing test code can be tedious and methodical. To alleviate this burden, a longstanding goal of researchers is to produce tests automatically. Automated test generation methods have been developed for many specific testing problems, most popularly generating unit tests using either random sampling or search-based techniques. In random testing, sequences of calls to constructors and methods are randomly assembled, and objects created in these calls are used as parameters for successive calls. A primary application of random testing is to find undeclared exceptions [11] or violations of general object contracts [12], but the generated tests can also be used as automated regression tests. The effectiveness of random test generators can be increased by integrating heuristics [13], [14]. In search-based testing, evolutionary search algorithms are popular, which gradually improve random initial sequences of calls to maximize code coverage [15]–[18].

Automatically generated tests do tend to be harder to read and interpret than manually crafted ones, which negatively impacts their maintainability [19]. Research has focused on improving this maintainability primarily by addressing the amount of code generated: RANDOOP [12] removes redundant tests after-the-fact, while EVOSUITE [17] uses test suite size as a secondary objective during its search and further post-processes individual tests to be "1-minimal" with respect to coverage; *i.e.*, leaving only statements that cannot be removed without reducing the achieved coverage.

Various other post-processing steps are common for automated test generators; for example, most tools integrate some form of regression oracle generation [20] such that the tests contain assertions based on the current behavior of the code under test. To further improve maintainability, these assertions can be minimized for their fault-finding potential [21]. Further optimizations for maintainability include applying specific heuristics from industrial application [22], semantic simplification [23], prioritizing using literals taken from the source code [24], generation of meaningful names for tests [25], measuring readability using prediction models [26], and adding textual summaries [27].

C. Limitations of Prior Work

Grano *et al.* [6] investigated test smells in automatically generated test cases and found them widespread: the vast majority (81%) of test suites generated by EVOSUITE contain at least one test smell, and even more so among other test case generation tools. That rate is remarkably high considering how widely EVOSUITE is used, which seems unlikely if those smells are indeed so abundant and indicative of quality and maintainability issues. A careful manual analysis of its tests and their purported smells thus seems in order. We focus on three systematic concerns with the established numbers:

1) Grano *et al.* used the warnings raised by an automated test smell detection tool to build their *gold standard*. However, the tool is known to overestimate the number of test smell instances [6], [28]. In this paper, we address this issue by building a *gold standard* with manually annotated classes, to have a more clear view of test smells occurrences in automatically generated tests.
2) The authors used test case generation tools with the default parameter values. However, tools like EVOSUITE are equipped with tuneable parameters, including the timeouts used for the minimization process, which are critical for producing maintainable tests. Test cases are not minimized after this timeout, which may leave many unnecessary statements and assertions, making them more likely to contain certain smells (*e.g.*, Assertion Roulette, discussed later). Thus an incorrect setup, rather than a fundamental issue in test case generation, could contribute to the higher numbers of smells observed in Grano *et al.* [6]. We address this by tuning EVOSUITE's parameters to ensure high quality test suite generation.
3) Modern test case generation tools use mocks to reduce the dependencies between generated tests and external resources (such as files). However, this aspect was not considered nor discussed by Grano *et al.*. Since some test smells are related to using external resources, we investigate in-depth how the relation between mocking

and certain test smells manifests in both their annotations and the generated tests.

Finally, the general explanation given by Grano *et al.* for the results is unsatisfactory; the argument is that generated tests are "*scented since the beginning since crossover and mutation operations performed though their evolution do not change the structure of the tests*." However, tools like EVOSUITE and JTEXPERT do evolve test suites and their test cases. Indeed, the mutation operator can add, remove, or insert statements in the test cases [29], [30], thus, altering the test structure. If the majority of the generated tests are indeed smelly, the root cause is to be found somewhere else.

III. METHODOLOGY

This section details the empirical evaluation we conduct to assess the performance of test smell detection tools when applied to automatically generated tests. A complete replication package is available on Zenodo [31].

A. Research Questions

The following research questions steer our empirical study:

RQ1 *How widespread are test smells in automatically generated test cases?*

RQ2 *How accurate are automated tools in detecting test smells in automatically generated tests?*

RQ3 *How well do test smells reflect real problems in test suites?*

We first determine the spread and distribution of test smells based on a manual analysis. Previous work [6] answered a similar research question, proposing a test smell detection tool as the *gold standard* with the assumption that this tool is 100% precise for four types of test smells and over 65% precise for three other smells [6]. In this study, we answer **RQ1** by building a curated dataset of automatically generated test cases (further described in Section III-B) and manually identifying the presence of one or more of six types of test smells (see Section III-D). Our approach addresses an important *threat to construct validity* to previous work, as our gold standard does not depend on detection tools. We next compare the detected extent of test smells with that predicted by two automatic smell detection tools in **RQ2** to establish their accuracy. This aims to (in)validate the findings of previous work which suggest that automatically generated test suites are highly prone to test smells. Finally, **RQ3** reflects on test smells, asking whether, and how often, they relate to actual problems in the test suites, which is illustrated through a series of examples.

B. Test case generation

To build our benchmark, we consider the same 100 Java classes used by Grano *et al.* [6]. We extract these classes from the SF110 dataset [29], which contains 110 projects from `SourceForge.net`. The selected classes are non-trivial as identified using a well-established *triviality test* [30], which filters out classes whose methods have a McCabe's Cyclomatic complexity lower than three. This helps ignore classes that can be fully covered by simple method invocations. We then generate JUnit test cases for the selected classes using EVOSUITE [7] version 1.0.7. EVOSUITE is a state-of-the-art unit-test generator for Java programs that has won several editions of the SBST tool competition [32]–[34] against competing random and search-based unit test generation tools; it produces test suites with high code coverage [29], has documented fault detection capability [35], [36], and is publicly available on `GitHub`.[1] EVOSUITE also applies several post-processing optimizations to reduce the size of the test cases as well as the number of generated assertions (see Section II-B).

Grano *et al.* used the default values for EVOSUITE parameters with the motivation that "*the use of default values does not impact the performance of automated test case generation tools*" [6]. However, there are several problems with this assumption: first, the default values are optimized with respect to code coverage [37], but other test suite properties like length or number of assertions may well be affected by changes in parameters. Second, the claim only holds for hyper-parameters of the metaheuristic search parameters for a given algorithm over multiple classes, but not necessarily for general parameters of EVOSUITE, such as post-processing options, search budget, or the choice of algorithm. Indeed, a large body of research has shown that changing the evolutionary algorithm can have a dramatic impact on the overall performance [30], [38]–[40]. Similarly, the search budget also has a substantial impact on the overall performance, as shown in the SBST tool competitions (e.g., [34]).

Parameter Settings. The default settings of EVOSUITE uses the *whole-suite approach* and optimizes eight different coverage criteria simultaneously. For the post-processing steps, the EVOSUITE default settings use 60 seconds for test case minimization, and a further 60 seconds for assertions generation/minimization. These parameter values can be derived from the EVOSUITE code and from the replication package by Grano *et al.*[2] If the test case minimization and the assertion generation reach their respective time-outs, EVOSUITE terminates the post-processing and returns the original, *non-minimized*, test suites. To ensure that this phase completes successfully, we thus increase the timeouts from 60 seconds (used in [6]) to ten minutes; this also leads to higher coverage.

In this work, we use more suitable settings for the considered task: we use DynaMOSA as the core evolutionary algorithm following the recommendations from more recent work [39]. DynaMOSA uses a many-objective genetic algorithm to evolve a population of 50 test cases. Test cases are evolved using *single-point* crossover with probability $p_c = 0.75$, *uniform mutation* with probability $p_m = 1/n$ (n being the length of the test cases), and *tournament selection*. DynaMOSA is also configured to optimize for eight coverage criteria, namely *branch*, *line*, *method*, *exception*, *input*, and *output* coverage plus *weak mutation score*. We select DynaMOSA over the *whole-suite approach* as the former can achieve higher coverage with shorter tests.

[1] https://github.com/EvoSuite/evosuite

[2] https://zenodo.org/record/3337892#.XswWby-w3yU

TABLE I: Comparison between the test suites generated with our setting vs. Grano *et al.* [6] setting. We report the median values, the the interquartile range (IQR), and confidence intervals (CI) using bootstrapping at 95% significance level.

	Settings by Grano *et al.* [6]			Our Settings		
Criterion	M	IQR	CI	M	IQR	CI
Branch Coverage	0.69	0.71	[0.66, 0.72]	**0.74**	0.70	[0.71, 0.76]
Overall Coverage	0.67	0.66	[0.65, 0.70]	**0.74**	0.65	[0.71, 0.76]
# of Test Cases	14	23	[13.86, 14.13]	15	26	[14.83, 15.17]
Total Test Length	50	135	[47.25, 52.60]	**46**	110	[43.63, 48.24]

Coverage Results. Table I lists summary statistics of the test suites generated using EVOSUITE with our settings, and compares them to the test suites obtained using the settings from the prior work. The reported numbers are the averages (median values plus interquartile ranges) over 50 independent runs. We run EVOSUITE multiple times on each class to address the randomized nature of evolutionary algorithms used to synthesize test suites. As a consequence, our test suites have slightly higher coverage (5%, significant in 45 out of 100 classes, as confirmed by Wilcoxon test $p < 0.05$) and are a bit shorter than in prior work [6].

C. Detection Tool Selection

We select two test smell detection tools. The first tool proposed by Bavota *et al.* [28] applies static detection rules. It has been widely used in previous work [28], [41], [42] to study the distribution of test smells in (manually-written tests of) open-source projects. Grano *et al.* [6] use this same tool to detect smells in automatically generated test suites. Bavota *et al.* report that this tool achieves 100% recall and 88% precision when detecting test smells in manually-written tests. For automatically generated tests, Grano *et al.* [6] reported an average precision of 75%, with 100% precision on four test smell types: *Assertion Roulette*, *Mystery Guest*, *Sensitive Equality*, and *For Testers Only*.

The second detection tool we study is TSDETECT [10], which is publicly available on `GitHub`.[3] Recently, Spadini *et al.* [2] calibrated the detection rules in TSDETECT based on developers' perception and classification of test smell *severity*, resulting in thresholds that are better aligned with what developers consider actual bad test design choices.

While TSDETECT can detect 21 test smell types, the tool used by Grano *et al.* [6] detects just seven; in our analysis, we focus on these seven, five of which are shared by both tools: *Assertion Roulette*, *Eager Tests*, *Sensitive Equality*, *Mystery Guest*, *Resource Optimism*. The *Indirect Testing* test smell is only supported by the tool used by Grano *et al.*; we discard the *For Tester Only* smell, since it is not applicable to automatically generated test suites that, by design, are linked only to the class they were generated to test. Comparing the results from two automated detection tools reduces the risk of drawing conclusions specific to one tool alone, helping us focus on identifying common limitations instead. An overview of the shortlisted smells can be found in Table II.

[3]The tool can be found at: https://github.com/TestSmells/TestSmellDetector

D. Manual validation

To create a golden set of test smells in automatically generated test suites, we manually evaluate all 100 generated test suites. For each test suite, we analyze whether any of the tests in the test suite suffer from one of the six smells specified in Table II. Out of these six smells, we do not explicitly annotate *Mystery Guest* and *Resource Optimism*, as these do not apply to tests generated by EVOSUITE and are thus trivially false for all its tests. We conduct this manual analysis in a multi-stage cross-validated manner:

Step 1. We divide the 100 test suites into two sets of 50 each. The first two authors of this paper independently analyze the first set, and the last two authors independently analyze the other set. For each test suite, the analysis is done across four dimensions, with each dimension corresponding to one of the selected test smells. Since we only look for the presence of a smell, for each dimension we use a binary marker. For our analysis, as a guideline each author adheres to the detection rules listed in Table II.

Step 2. For each set of 50 test suites, the two authors responsible for the analysis discuss their findings and determine the level of agreement. For the first set, the authors disagree on 12% of the 200 (four per test suite for 50 test suites) cases and for the second set, the authors disagree on 25% of the cases. Both sets of authors then discuss the disputed cases to come to a final resolution. We note that this discussion was used to slightly refine guidelines for corner-cases, as test smells manifest in many complex ways. At the end of the discussion phase, there are just four cases from the first set that have consensus and similarly three from the second set.

Step 3. For the seven disputed cases, all four authors responsible for the manual analysis discuss the cases to come to a final agreement. Furthermore, during this phase test cases that are not smelly but still interesting anomalies are also discussed. At the end of this phase the classification of the 100 test suites is set.

IV. EMPIRICAL RESULTS

A. RQ1: Distribution of Test Smells

To evaluate automated test smell annotation tools (RQ2) as well as to study the gaps in test smell coverage (RQ3), we need to understand in what ways our tests are characterized by the presently used test smells. This necessarily requires manual annotation, which we conducted as described in Section III-D. To be consistent with prior work [6], we annotated each smell at the level of the entire test suite; if any one test in said suite contained that smell, the entire suite is annotated accordingly.

Table III shows the resulting general incidence rate of each smell across the 100 test suites. Overall, test smells are commonly present in a small, but non-trivial portion of automatically generated test suites. The respective incidence rates are all quite similar, but their distributions vary; in about half of the test suites no smells were present at all,

TABLE II: Test smells considered in this paper.

Test smell	Definition by van Deursen *et al.* [1]	Rules for interpretation
Mystery Guest	Test case that accesses external resources such as files and databases, so that it is no longer self-contained.	Discarded, since the EvoSuite test suite runner by definition mocks out all accesses to external resources.
Eager Test	A test that checks multiple different functionalities in one case, which makes it hard to read or understand.	(1) The test must have more than one assertion and (2) at least one assertions is not on the result of a `get` method.
Assertion Roulette	A test that has multiple assertion statements that do not provide any description of why they failed	A test must have two or more assertions and neither has any explanatory message accompanying them
Indirect Testing	Tests the class under test using methods from other classes.	The presence of any assert that uses a method that is not part of the class under test.
Sensitive Equality	When an test checks for equality through the use of the `toString` method.	Any assert that checks the exact value of a String that is returned through a `toString` call is said to be sensitive
Resource Optimism	A test that makes optimistic assumptions about the state/existence of external resources	Discarded as the EvoSuite test suite runner by definition mocks out all accesses to external resources

TABLE III: Distribution statistics of the selected test smells in 100 test suites spanning all collected systems.

Smell	Rate
Eager Test	21%
Assertion Roulette	17%
Indirect Testing	32%
Sensitive Equality	19%
Mystery Guest*	0%
Resource Optimism*	0%

* EvoSuite never generates tests requiring external resources.

```
@Test(timeout = 4000)
public void test5() throws Throwable {
    ActionRegistry a0 = new ActionRegistry();
    ErrorPage errorPage0 = new ErrorPage();
    Label label0 = new Label(errorPage0, errorPage0);
    a0.addEntry("+}OIdF3!uYBcSb=", "+}OIdF3!uYBcSb=", false);
    boolean boolean0 = a0.isActionMethod(label0, (String) null);
    assertFalse(boolean0);
}
```

Fig. 1: Example of a false positive for the tool used by Grano *et al.* for Assertion Roulette.

another five contained every possible smell, and the remainder involved some pairings more frequently occurring than others. In particular, eager tests and assertion roulette often co-occurred (appearing together in 12 out of their respective 20 & 16 occurrences); these both describe tests that involve "too much" testing, either in terms of methods tested or in terms of (non-trivial) properties asserted.

We noticed that individual tests in a test suite were often very similar to each other, with typically just a few archetypes repeated with slightly different setup and conclusion. As a consequence, suites that we marked with a test smell also tended to contain it in many of its test methods. Conversely, ca. half of the suites contained no test smells at all for the same reason, although we caution that a few of these were empty (as in, no tests were generated). This suggests that generating diverse tests for a given class is still an open challenge, at least with respect to common smell-related pitfalls.

Finding 1. Test smells are commonly present in a small, but non-trivial portion of automatically generated test suites. However, their rate of occurrence is smaller than previously reported.

B. RQ2: Accuracy of Automated Test Smell Detection

Table IV reports the false-positive rates (FPR), false-negative rates (FNR), precision, recall, and F-measure that each tool achieves for each test smell. We were not able to compute all performance metrics for certain smells because either (1) our *gold standard* does not include instances of those smells (as in the case of Mystery guest and Resource optimism), or (2) the detection tool was unable to detect any instances of them.

Assertion roulette. The tool used by Grano *et al.* largely overestimates the number of instances for assertion roulette. The tool raises warnings for 76% of the test suites generated by EvoSuite, which is in line with the percentage (73.4%) reported in their work [6]. However, our analysis reveals a high rate of false-positives; although the tool achieves 100% recall, its low precision results in an F-measure of just 0.36.

To better understand this high FPR, we manually inspected the warnings raised by the tool. Most saliently, we observed that test methods with only a single assertion are common amongst the false positives; Figure 1 shows such an example, in which the test contains just one assertion. Such cases by definition *cannot* be classified as an instance of assertion roulette, as there is no cause for confusion in case of a failure.

Table IV suggests that TsDetect works fairly well on this test smell, reaching a higher precision (67%) but a lower recall compared to the tool used by Grano *et al.*, giving it a 21% higher F-score. This is mainly due to the higher threshold value (three assertions) used by Spadini *et al.*. However, this simple heuristics also causes the tool to miss some instances with fewer assertions; an example is shown in Figure 5, in which a test case checks (asserts) two different properties: object equality and the value of its attributes. At the same time, we acknowledge that tests with few assertions are questionable instances of assertion roulette. As pointed out by Spadini *et al.* [42], "*the test method name accurately reflects the reason for the test to fail*" even when no further comments

TABLE IV: Detection performance of different automated test smell detection tools for test cases generated by EvoSuite. FPR denotes the False Positive Rate and FNR is the False Negative Rate. The best values are highlighted in grey colour.

Test smell	Tool used by Grano *et al.* [6]					tsDetect calibrated by Spadini *et al.* [2]				
	FPR	FNR	Precision	Recall	F-measure	FPR	FNR	Precision	Recall	F-measure
Assertion Roulette	0.72	0.00	0.22	1.00	0.36	0.05	0.50	0.67	0.5	0.57
Eager Test	0.53	0.05	0.33	0.95	0.49	0.05	0.45	0.73	0.55	0.63
Mystery Guest	0.12	—	—	—	—	0.03	—	—	—	—
Sensitive Equality	0.00	0.67	1.00	0.33	0.50	0.00	0.67	1.00	0.33	0.50
Resource Optimism	0.02	—	—	—	—	0.02	—	—	—	—
Indirect Testing	0.00	1.00	—	0.00	—	—	—	—	—	—

```
@Test(timeout = 4000)
public void test07() throws Throwable {
   ScriptOrFnScope s0 = new ScriptOrFnScope((-806),
                        (ScriptOrFnScope) null);
   ScriptOrFnScope s1 = new ScriptOrFnScope((-330), s0);
   s1.preventMunging();
   s1.munge();
   assertNotSame(s0, s1);
}
```

Fig. 2: Example of false positive for the tool used by Grano *et al.* for Eager Test

```
@Test(timeout = 4000)
public void test00() throws Throwable {
    Show show0 = new Show();
    File file0 = MockFile.createTempFile("...");
    MockFileOutputStream m0 = new MockFileOutputStream(file0, false);
    MockPrintStream mP0 = new MockPrintStream(m0);
    show0.printHelpExtra(mP0, (List) null);
    assertEquals(797L, file0.length());
}
```

Fig. 3: Example of false positive for Mystery Guest

are provided in the tests.

Eager Test. The tool used by Grano *et al.* achieves high recall (95%) but low precision. The tool raises warnings for 62% of the test suites generated by EvoSuite, which corresponds to a false-positive rate of 53%; *i.e.*, the majority of warnings raised are not actual test smell instances. This tool uses the number of methods calls (not including constructors) in a test method to determine whether it is eager or not, but eagerness is properly concerned with *functionality* – whether more than one requirement is tested. Figure 2 depicts an example of a false positive where two methods on the object `ScriptOrFnScore1` are invoked. The first method sets the private attribute `markedForMunging` of the class to `false` (it is `true` by default). The method `munge` manipulates symbols in the global scope of the class if and only if the attribute `markedForMunging` is set to `true`. Testing this scenario requires both method invocations, otherwise one of the branches inside the method `munge` cannot be tested.

tsDetect achieves a higher precision (73%), again by using a higher threshold [42] for the same metric (the number of method invocations). This also causes it to again miss some instances, resulting in a lower recall (55%). For example, tsDetect correctly annotates the test in Figure 2 as non-smelly, but misses the case in Figure 5. The resulting F-score is again higher for tsDetect, reinforcing that research with developers and human participants is critical to calibrating test smell detection tools properly.

Mystery Guest and Resource Optimism. For these two types of smells, both detection tools raise several warnings. However, they are all false positives by definition, as our *gold standard* does not contain any instances of such smells. The detection tools both annotate test methods that contain specific strings or objects, such as: "`File`", "`FileOutputStream`" "`DB`", "`HttpClient`" as smelly; however, EvoSuite separates the test code from environmental dependencies (*e.g.*, external files) in a fully automated fashion through bytecode instrumentation [43]. In particular, it uses two mechanisms: (1) *mocking*, and (2) customized *test runners*. For one, classes that access the filesystem (*e.g.*, `java.io.File`) have all their methods (and constructor) mocked [43]. EvoSuite also replaces general calls to the Java Virtual Machine (*e.g.*, `System.currentTimeMillis`) with mock classes/methods with deterministic behavior. Finally, the test runner used by EvoSuite replaces occurrences of *console inputs* (*e.g.*, `java.io.InputStream`) in all instrumented classes with a customized console. Notice that EvoSuite resets all mock objects before every test execution. The application of static rules based on string patterns is thus insufficient to identify instances of these smells. Grano *et al.*'s tool, especially, does not identify mocks, thus raising a warning every time a test contains the string "`File`". Figure 3 shows an example of such a false positive. While tsDetect avoids misclassification of mocked file access by checking for the string "Mock", it does not inspect whether a customized test runner is used, which helps it achieve a lower FPR.

Indirect testing. 32% of the test suites in our *gold standard* contains test cases affected by indirect testing. This makes it the most widespread smell in automatically generated tests. However, the tool used in prior work [6] fails to detect any instances of this smell. Furthermore, tsDetect does not detect indirect testing. Therefore, further research is needed to capture indirect testing with automated tools effectively.

Sensitive equality. Based on its definition, this smell is particularly easy to detect with static rules, as it just requires checking whether the `toString` method is used for (equality-related) assertions. Surprisingly, both test smell detection tools detect only a small portion of this test smell's instances. Through manual analysis, we discovered that these tools successfully detect sensitive equality if and only if the method `toString` directly appears within an assertion. However, both detection tools can be easily fooled by using first storing the result of `toString` in a local variable and then asserting

```
@Test(timeout = 4000)
public void test56() throws Throwable {
    SubstringLabeler substringLabeler0 = new SubstringLabeler();
    substringLabeler0.connectionNotification("testSet", "testSet");
    InstanceEvent instanceEvent0 = substringLabeler0.m_ie;
    substringLabeler0.acceptInstance(instanceEvent0);
    assertEquals("SubstringLabeler",
                 substringLabeler0.getCustomName());
    assertFalse(substringLabeler0.isBusy());
    assertEquals("Match",
                 substringLabeler0.getMatchAttributeName());
}
```

Fig. 4: Example of eager test.

```
@Test(timeout = 4000)
public void test58() throws Throwable {
    OrganizationImpl organizationImpl0 = new OrganizationImpl();
    boolean boolean0 = organizationImpl0.equals(organizationImpl0);
    assertTrue(boolean0);
    assertEquals(0L, organizationImpl0.getPrimaryKey());
}
```

Fig. 5: Example of assertion roulette.

its value against the target.

Finding 2. Test smell detection tools overestimate the occurrence of all evaluated test smells, sometimes by large margins. TSDETECT is more precise due to calibrated thresholds, but sacrifices recall. Involving human participants is critical to improving the accuracy of these tools.

C. RQ3: Relation to Real Issues

The goal of test smells is to reflect real, rectifiable issues in test cases. It is thus important to ascertain that detected smells are actually indicative of problems in automatically generated test cases. Our manual validation was based on the definition and interpretation of a test smell provided by van Deursen *et al.* to ensure a fair comparison with previous work on test smell detection, but these smells have not been reassessed for generated test suites. In this section, we do so for the four test smells that EVOSUITE test suites can plausibly contain.

Eager test. We avoided mislabeling tests as eager when they checked an object's state using multiple getter calls after some action (which is rarely avoidable). Even so, we find that automatically generated tests are often eager in that they test (entirely) unrelated functionalities. One such example can be seen in Figure 4, where, the entity under test is `SubstringLabeler`. We observe that two of the asserts are checking the result of a getter on the entity, whereas the other checks whether the object is busy. Cases such as these were quite common, and clearly reflect a lack of singular purpose in test cases, which indeed risks maintainability issues.

Assertion roulette. Assertions in the test code can add a text-based explanation that is shown if it fails, which can help identify specifically which assert first triggered an error in case there are multiple. In our analysis, we thus do not consider tests with just one assert (with no accompanying message) as smelly, since it is trivial to trace failures for these; but, EVOSUITE tends to generate many test cases with multiple, and often very many, assertions. This is largely because it is prone to testing for multiple results of a series of method calls,

```
@Test(timeout = 4000)
public void test21() throws Throwable {
    Home home0 = new Home();
    SwingViewFactory swingViewFactory0 = new SwingViewFactory();
    PhotoController photoController0 = new PhotoController(home0,
           (UserPreferences) null, (View) null, swingViewFactory0,
           (ContentManager) null);
    Camera.Lens camera_Lens0 = Camera.Lens.FISHEYE;
    Camera camera0 = new Camera(2026, 3700L, 3700L, 2026, 3700L,
           2026, 3700L, camera_Lens0);
    home0.setCamera(camera0);
    assertEquals(3700L, camera0.getTime());
}
```

(a) Indirect test of `Camera` instead of `PhotoController`.

```
@Test(timeout = 4000)
public void test05()  throws Throwable  {
    LinkedHashMap<String, Object> linkedHashMap0 = new
                    LinkedHashMap<String,Object>();
    TeamFinderImpl teamFinderImpl0 = new TeamFinderImpl();
    linkedHashMap0.put("com.liferay.portal.service.persistence." +
                    TeamFinder.findByG_N_D",teamFinderImpl0);
    teamFinderImpl0.setJoin((QueryPos) null, linkedHashMap0);
    assertFalse(linkedHashMap0.isEmpty());
}
```

(b) Indirect test of `LinkedHashMap` instead of `TeamFinderImpl`.

Fig. 6: Examples of the indirect testing smell.

without a clear understanding of whether those results are related to a single "behavior" (*i.e.*, a single semantic action). We tend to find that when a test case has this smell, it is often also classified as an eager test case. One example of this can be found in Figure 5, which contains an `assertTrue` on the result of a (tautological) equality test and an unrelated `assertEquals` on an attribute of the same object.

Indirect testing. In our manually analyzed dataset, we found 30 test suites with cases of indirect testing, in which the actual tested behavior (*e.g.*, the final assert statement) relied on an unmocked call to a method of some other class to confirm correct behavior. This clearly violates the containment expected in unit testing. We specifically observed two kinds of indirect testing: (1) those where the entity under test has nothing to do with the test case at all, and (2) those where the test case asserts a property of a class that is related to the entity under test after the entity has interacted with it.

An example of the former can be seen in Figure 6a, where the class under test is `PhotoController`, but the time set on the `Camera` class is being asserted. In this test, the call to `home0.setCamera` leads to coverage on the class under test (the `PhotoController` is an observer of `home0`) such that the statement survives EVOSUITE's minimization. When EVOSUITE's regular mutation-based assertion minimization does not succeed in retaining any relevant assertions, as a last resort EVOSUITE adds an assertion on the last return value produced in the test case. In this case, however, the time value set on the `Camera` has nothing to do with the `PhotoController`. Support for more advanced assertions could have avoided this problem.

A more clear-cut case of indirect testing can be seen in Figure 6b. Here the class under test is `TeamFinderImpl`, but the ultimate assert checks a `LinkedHashMap` for emptiness to confirm some aspect of the behavior of `setJoin`

```
@Test(timeout = 4000)
public void test62() throws Throwable {
    SubstringLabeler.Match substringLabeler_Match0 = new
                                SubstringLabeler.Match();
    String string0 = substringLabeler_Match0.toString();
    assertEquals("Substring:     [Atts: ]", string0);
}
```

Fig. 7: Example of sensitive equality.

(to which the `LinkedHashMap` is passed). Although we marked this as smelly, in accordance with the pre-established definition, it is debatable whether this is actually an issue: there may not be a direct way to test this map's value through `TeamFinderImpl` (*e.g.*, through a getter), so that the tester is faced with the choice of either incurring this smell or not testing this property. This is not endemic to automatically generated test suites either; questions regarding testing of hidden (or 'private') properties are abundant on *e.g.*, StackOverflow and no consensus exists on what is appropriate.

Sensitive equality. Asserting the configuration of an object using its representation, as returned by a `toString` method, is non-robust: that representation is prone to changing in trivial ways, like adding/removing punctuation, which would cause a spurious test failure. We find that automated test cases do generate some tests that rely on the value returned by `toString` methods. Oddly enough, the invocation of `toString` is rarely done directly in the assert; rather, its result is often stored in a local variable which is then compared to the expected value in the assert (as seen in Figure 7). Whether these uses of `toString` constitute a real problem is debatable; for any such test, EvoSuite also generated many test cases that explicitly check for equality (to equivalent objects) and/or the values returned by all 'getter' methods. Tests such as this seemed to genuinely test the current implementation of the `toString` method – we very rarely found cases where the string representation was used specifically to confirm program state after some call, or to test equality to another object.

Mystery guest and Resource optimism. Mocking and bytecode instrumentation are the core techniques used by EvoSuite to handle environmental dependencies [43]. Originally, these techniques were introduced to solve other challenges, such as removing non-determinism (the primary cause of flaky tests), avoiding the creation/deletion/modification of external files, and ultimately to increase code coverage. Our analysis reveals that these strategies positively impact the maintainability of generated tests by preventing these smells.

Finding 3. While EvoSuite generates eager tests and ones with multiple assertions, their severity is debatable. Mocks and the bytecode instrumentation used in EvoSuite effectively prevent mystery guests and resource optimism.

V. Qualitative reflection

In the previous section, we presented quantitative results grounded in a thorough investigation of test smell prevalence. In the process of this annotation effort, one cannot help but observe many recurring patterns, both in the way test smells manifest, are (mis-)detected and miss other issues entirely. This section discusses such observations qualitatively, with examples from our dataset, discussing each smell separately.

A. On Rule-Based Detection of Test Smells

Automatically detecting test smells requires explicitly encoding their most salient, reliable characteristics. The previous section discussed the challenge of this problem in relation to established definitions [1], but these definitions are not exact; both Grano *et al.* and Spadini *et al.* quote these definitions but interpret them differently in subtle ways. We discuss issues with these definitions and their interpretation here.

Eager test. These tests evaluate the behavior of multiple methods in a single test method. Both tools considered rely on the number of production method invocations to detect this, but Spadini *et al.* [2] set a higher threshold than Grano *et al.* [6], who consider any more than one invocation to be smelly. This definition does not necessarily capture real "eagerness", however; some tests necessarily invoke multiple methods to test more complex behavior (*e.g.*, a pair of encrypt and decrypt methods); as long as a separate test case exists for its intermediate stages, this should not be a concern. It is highly non-trivial to detect for this automatically and it is especially fault-prone to assume a threshold of just one invocation. In our manual analysis, we excluded many common occurrences of this pattern, such as multiple invocations of getters of the same class, which simply test various aspects of its state after a single operation, or two equality checks that ascertain bidirectional equality. Note that refactoring those would result in substantial code bloat (as also alluded to by Van Deursen *et al.* [1]). As such, detecting this test smell requires much more semantic awareness than is currently present.

Assertion roulette. When a test case has multiple asserts without explanations, pinpointing why it failed was historically complicated: JUnit 2 was widely used at the time of this smell's definition, which had no traceability for the cause of failing test cases with multiple asserts. Both Spadini *et al.* and Grano *et al.* annotate this smell when an assert statement has no string message to explain a potential failure [2], [6], though Spadini *et al.* require at least two such asserts. Currently, EvoSuite only documents cases where an exception is expected (using JUnit's `fail` method) – automatically generating failure-related messages is out of the scope of current tools. This results in automated tools marking many of their tests as smelly, in many cases incorrectly so. For one, test cases with just a single assert, even if not explained, should never involve this confusion. Furthermore, it is debatable whether *e.g.*, `assertNull` (in general) needs an explanatory message as the expected behavior is encoded in its name reason. More generally, advances in the JUnit framework have removed the traceability confound entirely. We still annotated some cases with this smell based on a strict adherence to its definition, but suggest that this smell has become obsolete, which is further reinforced by its high degree of overlap with Eager Test.

Indirect testing. Testing classes other than the specific entity under test is considered indirect testing. Grano *et al.* interpret this as using any methods of another class [6]; but, we found many such invocations that were necessary for setup, which were often either Mocked, or not used in any assertions (*i.e.*,, only needed for setting up a scenario). Even discarding such trivial distractors, we found indirect testing to be a widespread issue with automated generated test suites in our manual analysis. Strangely, although we adhere to a stricter definition than Grano *et al.*, we still find 30 cases of test suites with this smell. This is significantly more than Grano *et al.*, whose detection approach did not even identify a single instance.

Sensitive equality. When a test asserts that an object has a given value (or checks its equality) using the result of its `toString` method, it is considered "sensitive". Grano *et al.* interpret this as the presence of a `toString` call specifically in an assert statement [6]. However, we found that EvoSuite often stores the result of a `toString` in a local variable before checking its value, so this detection rule has many false negatives. This pattern suggests a disconnect between human-written and automatically generated test suites; the proposed rule may work well on regular tests, but falls short on those automatically generated by EvoSuite.

Mystery guest and resource optimism. Mocking and byte-code instrumentation introduce more challenges for test smell detection tools based on static rules. TSDetect successfully reduces the false positive rate by checking for mocked objects. However, static rules fall short for strategies that work at the instrumentation level. These strategies can be fully detected via dynamic analysis (*e.g.*, identifying which objects are in memory) or using watchdogs to check whether the tests modify external files. Therefore, we foresee more sophisticated rules to detect mystery guests and resource optimism in automatically generated tests effectively.

B. On Issues not Included in Test Smells

During our manual analysis, we also uncovered issues that are not captured by the existing test smells definitions. This is due in part to the unique nature of (EvoSuite's) automatically generated tests, but also reminiscent of more general problems with detecting only a closed vocabulary of "issues".

Absence of assertions. We find that many test cases contain (sometimes elaborate) setup and invocations to the entity under test, but then do not assert the results of these method invocations in any way. One example is shown in Figure 8a, where the assert is commented out by EvoSuite due to instability concerns, which results in this test case having no asserts. In Figure 8b, EvoSuite generates a test case with just a constructor invocation but does nothing with it at all. Such invocations will show up as providing code coverage for the methods being inspected; however, with no assertions taking place, it tests nearly nothing of semantic importance[4]. This reflects a disconnect between the optimization metric of

[4]Though, one might argue, that an invocation which does not trigger an exception is still a form of a test.

```
@Test(timeout = 4000)
public void test3() throws Throwable {
    XML xML0 = new XML();
    MockPrintStream mockPrintStream0 = new
                                MockPrintStream(",qmf=");
    ConsoleInput consoleInput0 = new ConsoleInput((AzureusCore)
                                null, mockPrintStream0);
    xML0.execute((String) null, consoleInput0,
                                consoleInput0.torrents);
    //Unstable assertion: assertFalse(consoleInput0.isDaemon());
}
```

(a) Example of test with unstable assertion.

```
@Test(timeout = 4000)
public void test7() throws Throwable {
    ClientIDManagerImpl impl0 = new ClientIDManagerImpl();
}
```

(b) Example of test with no assertion.

Fig. 8: Example of tests with no assertions.

```
@Test(timeout = 4000)
public void test0() throws Throwable {
    SessionProperties sp0 = mock(SessionProperties.class, new
                                ViolatedAssumptionAnswer());
    SessionProperties sp1 = mock(SessionProperties.class, new
                                ViolatedAssumptionAnswer());
    doReturn("The 'data' array must have length ==2.")
                .when(sp1).getObjectFilterExclude();
    doReturn("7}3c]d+XG]mJk6La")
                .when(sp1).getObjectFilterInclude();
    ISession i0 = mock(ISession.class, new
                                ViolatedAssumptionAnswer());
    doReturn((IApplication) null).when(i0).getApplication();
    doReturn(sp0, sp1, sp1).when(i0).getProperties();
    ObjectTreeCellRenderer o0 = null;
    try {
      o0 = new ObjectTreeCellRenderer((ObjectTreeModel) null, i0);
      fail("Expecting exception: NullPointerException");

    } catch(NullPointerException e) {
      verifyException("net.sourceforge.squirrel_sql.client.session
                .mainpanel.objecttree.ObjectTreeCellRenderer", e);
    }
}
```

Fig. 9: Example of a test case with failed setup.

"coverage" and real-world validity of test cases; addressing this could lead to more useful support for developers.

Too many assertions. A substantial number of test cases contained many asserts, often at least five, but sometimes dozens – one peculiar suite had multiple test cases with nearly 80 assertions. This reflects an incredibly high assertion density. Although this certainly overlaps with the definition of established smells such as Assertion Roulette and Eager Test, the scope of this problem is vastly different, and thus likely requires differently targeted solutions than what might plausibly occur in regular, developer-generated tests.

Failed setup. We found many tests that involved a substantial amount of setup, often including entities set up via mock objects, but nevertheless resulting in exceptions that suggest the setup was not successful. Figure 9 shows an example of such a test: all the test code related to mocking `ISession i0` helps to cover the elaborate initialization code of the class `ObjectTreeCellRenderer`; yet eventually the constructor throws a `NullPointerException`. This is again indicative of a mismatch between coverage of code vs. actual

requirements: while EvoSuite succeeded in achieving high coverage through this setup, the resulting test is unlikely to be helpful for finding faults, besides being hard to maintain.

VI. Threats to Validity

The narrow focus of this work implies that the main threats to its validity are external.

Threats to external validity. EvoSuite is commonly used and has been continually developed, but is far from the only automatic test suite generation tool. Other examples include Randoop [12] and JTExpert [14], which were assessed in some prior work (*e.g.*, [6]); conclusions from our manual analysis may not extend to these tools. Nonetheless, many of our observations concern problems caused by the discrepancy between generated tests and those a human might write; even if specific issues may not recur across tools, the presence of a discrepancy almost certainly will. The broader result of this paper, which showed that current test smells are often inappropriate indicators of issues with such test suites, is thus likely equally applicable to such work, and follow-up studies assessing the degree and characteristics of such issues for other tools would be appropriate. Similarly, our selection of test smells is smaller than some prior work, but comparable to other. This does not invalidate our specific findings for these smells, but does imply that further studies are needed to confirm whether similar conclusions apply to other test smells.

Threats to construct validity. The main challenge in conducting this study was the interpretation of the definitions of the various test smells, which were never defined precisely and have been adopted in subtly different ways. We aimed to interpret them using a small set of simple, but semantically reasonable rules, which we detailed carefully. Choosing alternative interpretations may be appropriate for some purposes (*e.g.*, when using an older version of JUnit), and can certainly change a number of annotations. But, our experience while annotating was that no variation would eliminate a smell completely, or make it abundant. Furthermore, two raters independently annotated each example and discussed any discrepancies with respect to the established rules, adding clauses agreed on by all annotators in case of any lingering ambiguity. As such, we are confident that our annotations are internally consistent and highly traceable to our rules, which we believe are common-sense interpretations of these smells.

VII. The Path Forward

In contrast to previous work, which found that the test suites generated by EvoSuite are riddled with smells, we found the majority of generated test suites to be smell free. Having analyzed the main causes for these tools' false positives, it stands out that these are all static in nature and use rather simple heuristics (*e.g.*, relying on specific numerical thresholds for the number of asserts that are not able to clearly capture semantic aspects of smells) that require enhancements. To that end, we propose the following review of test smell detection:

1) Smells such as Assertion Roulette (which has become generally obsolete), Resource Optimism, For Testers Only and, Mystery Guest no longer apply to (well-calibrated) automatic test generators. A root and branch review of test smells and their detection tools/strategies is warranted to ensure that developers and future work do not rely on definitions that need substantial adaptation to this context.
2) The definition and interpretation of certain smells, such as Indirect Testing, appeared to be unpriced and possibly incomplete. Currently, tools interpret it as any method invocation to a class that is not the one under test, which hardly matches its intent. There is thus a need for a definition that is not only precise but captures the notion of a *semantic objective* for a test, under which it tests a specific, realistic behavior. This objective need not be self-contained and could span multiple methods, or even classes, as long as it has a well-defined goal. Defining this, and automatically detecting it, is an ambitious, yet pressing open challenge for this line of research.
3) Our study highlights important internal validity of prior work. Current tools are benchmarked on a false-positive prone "golden set" of manually validated data, which resulted in obvious errors being left uncaught. This highlights the need for a global, thoroughly verified dataset that can be used across studies as a reference benchmark.

VIII. Conclusions

This paper investigates test smell occurrence in automatically generated test-cases and the extent to which contemporary test smell detection tools are able to identify them. We built a dataset of 2,340 test cases automatically generated by EvoSuite for 100 Java classes and conducted a multi-stage, manual cross-validation to identify six types of test smells across these. Our results show that test smells are commonly present in a small, but non-trivial portion of automatically generated test suites. However, they occurr far less often then reported by the tool (and analysis) of Grano *et al.* [6], while tsDetect achieves somewhat better results [42]. In particular, although EvoSuite does generate eager tests and tests with multiple assertions, many heuristically detected cases are not problematic, while the severity of others is debatable (as also argued in recent work [42]). In turn, mocks and bytecode instrumentation techniques used in EvoSuite effectively address the problem of avoiding mystery guests and resource optimism. Our findings suggests that involvement of human participants (preferably in industrial contexts) is critical to ensuring that the design test smell detection tools better reflects developer practice. Additionally, the discrepancies between tool assessment and practical accuracy exposed by our work suggests the need for further studies of other test case generation tools and test smells.

Acknowledgements

The authors thank Davide Spadini for providing the implementation of tsDetect with calibrated thresholds. This work is supported by EPSRC project EP/N023978/2.

REFERENCES

[1] A. Deursen, L. Moonen, A. Bergh, and G. Kok, "Refactoring test code," in *Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes (XP2001)*, 2001, pp. 92–95.

[2] D. Spadini, M. Schvarcbacher, A.-M. Oprescu, M. Bruntink, and A. Bacchelli, "Investigating severity thresholds for test smells," 2020.

[3] G. Bavota, A. Qusef, R. Oliveto, A. D. Lucia, and D. W. Binkley, "An empirical analysis of the distribution of unit test smells and their impact on software maintenance," in *ICSM*, 2012, pp. 56–65.

[4] N. Tsantalis, T. Chaikalis, and A. Chatzigeorgiou, "Ten years of jdeodorant: Lessons learned from the hunt for smells," in *25th International Conference on Software Analysis, Evolution and Reengineering, SANER 2018, Campobasso, Italy, March 20-23, 2018*, R. Oliveto, M. D. Penta, and D. C. Shepherd, Eds. IEEE Computer Society, 2018, pp. 4–14.

[5] G. Bavota, A. Qusef, R. Oliveto, A. D. Lucia, and D. W. Binkley, "Are test smells really harmful? an empirical study," *Empirical Software Engineering*, vol. 20, no. 4, pp. 1052–1094, 2015.

[6] G. Grano, F. Palomba, D. Di Nucci, A. De Lucia, and H. C. Gall, "Scented since the beginning: On the diffuseness of test smells in automatically generated test code," *Journal of Systems and Software*, vol. 156, pp. 312–327, 2019.

[7] G. Fraser and A. Arcuri, "Evosuite: automatic test suite generation for object-oriented software," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, 2011, pp. 416–419.

[8] M. Fowler, *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 2018.

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

[10] A. S. A. Peruma, "What the smell? an empirical investigation on the distribution and severity of test smells in open source android applications," 2018.

[11] C. Csallner and Y. Smaragdakis, "Jcrasher: an automatic robustness tester for java," *Software: Practice and Experience*, vol. 34, no. 11, pp. 1025–1050, 2004.

[12] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," in *International Conference on Software Engineering*, 2007, pp. 75–84.

[13] L. Ma, C. Artho, C. Zhang, H. Sato, J. Gmeiner, and R. Ramler, "Grt: Program-analysis-guided random testing (t)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2015, pp. 212–223.

[14] A. Sakti, G. Pesant, and Y.-G. Guéhéneuc, "Instance generator and problem representation to improve object oriented code coverage," *Transactions on Software Engineering*, vol. 41, no. 3, pp. 294–313, 2014.

[15] P. Tonella, "Evolutionary testing of classes," *ACM SIGSOFT Software Engineering Notes*, vol. 29, no. 4, pp. 119–128, 2004.

[16] L. Baresi and M. Miraz, "Testful: Automatic unit-test generation for java classes," in *International Conference on Software Engineering*, vol. 2, 2010, pp. 281–284.

[17] G. Fraser and A. Arcuri, "Whole test suite generation," *IEEE Transactions on Software Engineering*, vol. 39, no. 2, pp. 276–291, 2012.

[18] J. H. Andrews, T. Menzies, and F. C. Li, "Genetic algorithms for randomized unit testing," *Ieee transactions on software engineering*, vol. 37, no. 1, pp. 80–94, 2011.

[19] S. Shamshiri, J. M. Rojas, J. P. Galeotti, N. Walkinshaw, and G. Fraser, "How do automatically generated unit tests influence software maintenance?" in *International Conference on Software Testing, Verification and Validation*, 2018, pp. 250–261.

[20] T. Xie, "Augmenting automatically generated unit-test suites with regression oracle checking," in *European Conference on Object-Oriented Programming*. Springer, 2006, pp. 380–403.

[21] G. Fraser and A. Zeller, "Mutation-driven generation of unit tests and oracles," *Transactions on Software Engineering*, vol. 38, no. 2, pp. 278–292, 2011.

[22] B. Robinson, M. D. Ernst, J. H. Perkins, V. Augustine, and N. Li, "Scaling up automated test generation: Automatically generating maintainable regression unit tests for programs," in *International Conference on Automated Software Engineering*, 2011, pp. 23–32.

[23] S. Zhang, "Practical semantic test simplification," in *International Conference on Software Engineering*, 2013, pp. 1173–1176.

[24] J. M. Rojas, G. Fraser, and A. Arcuri, "Seeding strategies in search-based unit test generation," *Software Testing, Verification and Reliability*, vol. 26, no. 5, pp. 366–401, 2016.

[25] E. Daka, J. M. Rojas, and G. Fraser, "Generating unit tests with descriptive names or: Would you name your children thing1 and thing2?" in *International Symposium on Software Testing and Analysis*, 2017, pp. 57–67.

[26] E. Daka, J. Campos, G. Fraser, J. Dorn, and W. Weimer, "Modeling readability to improve unit tests," in *Joint Meeting on Foundations of Software Engineering*, 2015, pp. 107–118.

[27] S. Panichella, A. Panichella, M. Beller, A. Zaidman, and H. C. Gall, "The impact of test case summaries on bug fixing performance: an empirical investigation," in *International Conference on Software Engineering*, 2016, pp. 547–558.

[28] G. Bavota, A. Qusef, R. Oliveto, A. De Lucia, and D. Binkley, "Are test smells really harmful? an empirical study," *Empirical Software Engineering*, vol. 20, no. 4, pp. 1052–1094, 2015.

[29] G. Fraser and A. Arcuri, "A large-scale evaluation of automated unit test generation using evosuite," *Trans. Softw. Eng. Methodol.*, vol. 24, no. 2, pp. 8:1–8:42, Dec. 2014.

[30] A. Panichella, F. M. Kifetew, and P. Tonella, "Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets," *Transactions on Software Engineering*, vol. 44, no. 2, pp. 122–158, 2017.

[31] A. Panichella, S. Panichella, G. Fraser, A. A. Sawant, and V. J. Hellendoorn, "Replication package of "Revisiting Test Smells in Automatically Generated Tests: Limitations, Pitfalls, and Opportunities"," Aug. 2020. [Online]. Available: https://doi.org/10.5281/zenodo.4000852

[32] A. Panichella and U. R. Molina, "Java unit testing tool competition-fifth round," in *International Workshop on Search-Based Software Testing*, 2017, pp. 32–38.

[33] F. Kifetew, X. Devroey, and U. Rueda, "Java unit testing tool competition-seventh round," in *International Workshop on Search-Based Software Testing*, 2019, pp. 15–20.

[34] X. Devroey, S. Panichella, and A. Gambi, "Java Unit Testing Tool Competition - Eighth Round," in *International Conference on Software Engineering Workshops*, Seoul, Republic of Korea, 2020.

[35] G. Fraser and A. Arcuri, "1600 faults in 100 projects: automatically finding faults while achieving high coverage with evosuite," *Empirical software engineering*, vol. 20, no. 3, pp. 611–639, 2015.

[36] M. M. Almasi, H. Hemmati, G. Fraser, A. Arcuri, and J. Benefelds, "An industrial evaluation of unit test generation: Finding real faults in a financial application," in *ICSE SEIP*, 2017, pp. 263–272.

[37] A. Arcuri and G. Fraser, "Parameter tuning or default values? an empirical investigation in search-based software engineering," *Empirical Software Engineering*, vol. 18, no. 3, pp. 594–623, 2013.

[38] J. M. Rojas, M. Vivanti, A. Arcuri, and G. Fraser, "A detailed investigation of the effectiveness of whole test suite generation," *Empirical Software Engineering*, vol. 22, no. 2, pp. 852–893, 2017.

[39] J. Campos, Y. Ge, N. Albunian, G. Fraser, M. Eler, and A. Arcuri, "An empirical evaluation of evolutionary algorithms for unit test suite generation," *Information and Software Technology*, vol. 104, pp. 207–235, 2018.

[40] A. Panichella, F. M. Kifetew, and P. Tonella, "A large scale empirical comparison of state-of-the-art search-based test case generators," *Information and Software Technology*, vol. 104, pp. 236–256, 2018.

[41] M. Tufano, F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk, "An empirical investigation into the nature of test smells," in *International Conference on Automated Software Engineering*, 2016, pp. 4–15.

[42] D. Spadini, F. Palomba, A. Zaidman, M. Bruntink, and A. Bacchelli, "On the relation of test smells to software code quality," in *International Conference on Software Maintenance and Evolution*, 2018, pp. 1–12.

[43] A. Arcuri, G. Fraser, and J. P. Galeotti, "Automated unit test generation for classes with environment dependencies," in *International conference on Automated software engineering*, 2014, pp. 79–90.