Automatic Generation of Smell-free Unit Tests

João Afonso[1], José Campos[1,2]
[1]LASIGE, Faculdade de Ciências, Universidade de Lisboa, Lisboa, Portugal
[2]Faculty of Engineering, University of Porto, Porto, Portugal
fc51111@alunos.fc.ul.pt, jcmc@fe.up.pt

***Abstract*—Automated test generation tools, such as EvoSuite, typically aim to generate tests that maximize code coverage and do not adequately consider non-coverage aspects that may be relevant for developers, e.g., test's code quality. Hence, automatically generated tests are often affected by test-specific bad programming practices, i.e., test smells, that may hinder the quality of the test's source code and, ultimately, the quality of the code under test. Although EvoSuite uses secondary criteria and a post-processing procedure to optimize non-coverage aspects and improve the readability of the tests, it does not explicitly consider the usage of good programming practices. Thus, in this paper, we propose a novel approach to assist EvoSuite's search algorithm in generating smell-free tests out of the box. To this aim, we first compile a set of 54 test smell metrics from several sources. Secondly, we systematically identify 30 smells that do not affect the tests generated by EvoSuite and eight smells that cannot be automatically computed. Thirdly, we incorporate the remaining 16 test smells as metrics into EvoSuite and empirically identify that only 14 smells affect the tests generated by the tool (e.g., Indirect Testing). Fourthly, we describe and integrate an approach to optimize test smell metrics into EvoSuite. Finally, we conduct an empirical study to (i) understand to what extend EvoSuite's default mechanisms leads to the generation of fewer smelly tests. (ii) to assess whether our approach leads to the generation of fewer smelly tests. And (iii) how our approach affects the coverage and fault detection effectiveness of the generated tests. Our results report that our approach can generate 8.58% fewer smelly tests without significantly compromising their coverage or fault detection effectiveness.**

***Index Terms*—Software Testing, Automated Test Generation, Test Smells, Empirical Study**

I. Introduction

Software testing [1], which aims to ensure the development of high-quality software products, is an essential procedure in any software project [2, 3]. Although an effective process, it has been estimated that half of the total cost and time to develop a software product is dedicated to software testing [1] because: (i) assessing whether a software product performs correctly could be highly complex, and (ii) software testing is traditionally a manual process that is costly, time-consuming, and subject to incompleteness and other errors. To improve the effectiveness of software testing and reduce its cost, others have devised approaches to automate the generation of tests.

Automated test generation tools such as EvoSuite [4, 5] have become very effective at generating high-coverage tests [2, 6], detecting real bugs [7, 8], and reducing debugging costs (compared to manually-written tests) [9, 10]. However, such tools do not adequately optimize the quality of the generated tests; thus, they are often affected by test-specific bad programming practices [3, 11, 12] — **test smells** [13].

It is well known that automatically generated tests are harder to understand [14] and maintain [15] than manually-written tests, and the presence of test smells further exacerbates these problems [16, 17]. In particular, the presence of test smells:

- Hinders test comprehensibility and maintainability [17, 16].
- Compromises test code effectiveness [18, 19].
- Makes test code more prone to changes and faults [18].
- Makes code under test more fault-prone [18].

Thus, the primary goal of this study is to develop and integrate a novel approach into the EvoSuite tool, which would allow the tool to generate smell-free tests out of the box. To this aim, we incorporate a curated set of test smell metrics into the EvoSuite tool and optimize those metrics as **secondary criteria** [2, 3, 11]. Others have successfully incorporated non-coverage quality metrics into EvoSuite (e.g., [2, 11]) as secondary criteria.

The main contributions of this paper are as follows:

- A curated list of 54 test smells and a detailed analysis of the smells that can (1) affect the tests generated by EvoSuite and (2) be characterized by optimizable metrics.
- A novel approach to optimize test smell metrics.
- An empirical study to assess
  - The diffusion of smells in the tests generated by EvoSuite and to assess the test smells that affect a significant portion of the generated tests.
  - The impact of EvoSuite's default mechanisms on the number of smelly tests.
  - Whether the optimization of test smells significantly reduces the number of smelly tests and does not negatively affect their coverage or fault detection effectiveness.

II. Related Work

Code smells were initially defined by Fowler [20] in 2018 as patterns in the code that suggest the possibility of refactoring, thus helping one to decide when and how to refactor. Since then, others have extended the concept of code smells to test code and established different catalogs of test-specific smells (**test smells**) along with their symptoms, impact, causes, and refactoring operations to remove them [13, 21, 22, 23, 17, 16, 18, 19].

Test smells correspond to suboptimal design/programming practices specific to test code that correlate with test implementation, organization, documentation, and interactions [16, 21, 24]. These smells are symptoms of possible problems in the test code and are often highly diffused in manually written [17, 16] and automatically generated tests [3, 12].

Palomba et al. [12] investigated the extent to which the tests generated by EvoSuite are affected by test smells and concluded that (1) smells are highly diffused throughout automatically generated tests and (2) “Assertion Roulette”, “Eager Test”, and “Test Code Duplication” were the most diffused smells. Grano et al. [3], extended Palomba et al. [12]’s work and studied the diffusion of test smells in the tests generated by EvoSuite, Randoop, and JTExpert. The study revealed that: (1) all tools generate smelly test code; (2) “Assertion Roulette” and “Eager Test” are the most diffused smells in the tests generated by all tools; (3) the presence of some smells may imply the presence of other smells; (4) the size of the tests is associated with the occurrence of certain smells. We extend upon Palomba et al. [12]’s and Grano et al. [3]’s work as follows: (1) we perform our study on a newer version of EvoSuite; (2) we consider a larger set of 16 smells and implement the respective test smell metrics; (3) instead of just detecting smells, we also optimize the proposed test smell metrics to generate fewer smelly tests.

Panichella et al. [25] conducted a study to determine the effectiveness of test smell detection tools at identifying smells in automatically generated tests. They used two tools to detect six smells in the tests generated by EvoSuite: the tool developed by Bavota et al. [17, 16] and the tsDetect tool [26]. Firstly, Panichella et al. performed a manual investigation to assess the smelliness of the tests generated by EvoSuite and observed that: (1) automatically generated tests are affected by a small but non-trivial quantity of smells; (2) “Assertion Roulette” and “Eager Test” frequently co-occurred together; (3) “Indirect Testing” was the most diffused smell type. Secondly, they compared the identified test smells with the smells reported by both tools and concluded that both overestimated the smelliness of the generated tests. Panichella et al. [27] extended upon their work and confirmed the previous results.

III. Test Smells in Practice

In this section, we revisit the topic of how smells affect the test automatically generated by EvoSuite [3, 12, 25]. We combined two of the largest sets of test smells that have been proposed and evaluated in the literature [28]) into the list of 54 smells that we use throughout the remainder of this study. All smells are listed in Table I and described in detail in *URL redacted for anonymity* (see `SMELLS.md` file). In detail, we investigate which of the 54 test smells do not affect the tests generated by EvoSuite by design (Section III-A), which smells cannot be automatically computed (Section III-B), which smells can be computed and how (Section III-C), and which smells do indeed affect the generated tests (Section III-D). Note our investigation augments the results reported in prior studies [3, 12, 25] with a larger set of smells on the latest version of EvoSuite (i.e., v1.2.0).

Table I: Test smells considered in this study. The “Affect” column identifies the smells that can/cannot affect the generated tests. The “Metric” column identifies the smells that can also be characterized by computational metrics. The “Sec. Criteria” column identifies the smells with metrics that can be optimized as secondary criteria. The rows highlighted in gray denote the smells with metrics that we optimize as secondary criteria (see Section IV).

Name	Abbr.	Affect?	Metric?	Sec. Criteria?
Abnormal UTF-Use	AUU	NO	—	—
Anonymous Test	AT	YES	NO	—
Assertion Roulette	AR	YES	YES	NO
Brittle Assertion	BA	YES	NO	—
Conditional Test Logic	CTL	NO	—	—
Constructor Initialization	CI	NO	—	—
Dead Field	DF	NO	—	—
Default Test	DT	NO	—	—
Duplicate Assert	DA	YES	YES	NO
Eager Test	ET	YES	YES	YES
Empty Shared-Fixture	ESF	NO	—	—
Empty Test	EmT	NO	—	—
Erratic Test	ErT	NO	—	—
Exception Handling	EH	NO	—	—
For Testers Only	FTO	NO	—	—
Fragile Test	FT	YES	NO	—
Frequent Debugging	FD	YES	NO	—
General Fixture	GF	NO	—	—
Hard-to-Test Code	HTTC	NO	—	—
Ignored Test	IgT	NO	—	—
Indirect Testing	IT	YES	YES	YES
Lack of Cohesion of Methods	LCM	YES	YES	NO
Lazy Test	LT	YES	YES	NO
Likely Ineffective Object-Comparison	LIOC	YES	YES	YES
Magic Number Test	MNT	NO	—	—
Manual Intervention	MI	NO	—	—
Mixed Selectors	MS	NO	—	—
Mystery Guest	MG	NO	—	—
Non-Java Smells	NJS	NO	—	—
Obscure In-line Setup	OISS	YES	YES	YES
Overcommented Test	OCT	NO	—	—
Overreferencing	OF	YES	YES	YES
Proper Organization	PO	YES	NO	—
Redundant Assertion	RA	YES	YES	NO
Redundant Print	RP	NO	—	—
Resource Optimism	RO	NO	—	—
Returning Assertion	ReA	NO	—	—
Rotten Green Tests	RGT	YES	YES	YES
Sensitive Equality	SE	YES	YES	NO
Sleepy Test	ST	NO	—	—
Slow Tests	SloT	YES	NO	—
Teardown Only Test	TOT	NO	—	—
Test Code Duplication	TCD	YES	NO	—
Test Logic in Production	TLP	NO	—	—
Test Maverick	TM	NO	—	—
Test Pollution	TP	NO	—	—
Test Redundancy	TR	YES	YES	NO
Test Run War	TRW	NO	—	—
Test-Class Name	TCN	YES	NO	—
Unknown Test	UT	YES	YES	NO
Unused Inputs	UI	YES	YES	NO
Unusual Test Order	UTO	NO	—	—
Vague Header Setup	VHS	NO	—	—
Verbose Test	VT	YES	YES	YES

A. *Test smells that do not affect, by design, EvoSuite’s tests*

By design, some smells do not affect the tests generated by EvoSuite because the tool is unable to produce tests with particular characteristics, e.g., EvoSuite does not generate tests with conditional statements. In this subsection, we describe some of the 30 smells, due to the lack of space, (out of 54) that do not affect the tests generated by EvoSuite. Column ‘Affect’ in Table I lists those smells.

Implicit setups: The tests generated by EvoSuite do not use implicit setups (i.e., setup methods used by all tests in a test suite) or teardown methods. Each test contains the setup code. Hence, the “Dead Field”, “Empty Shared-Fixture”, “General Fixture”, “Test Maverick”, and “Teardown Only Test” smells do not affect the generated tests.

Improper setup not contained in a test case: The generated tests contain all the setup code, thus the “Constructor Initialization” and “Vague Header Setup” smells do not occur.

Problems that do not apply to JUnit tests: The “Default Test”, “Non-Java Smells”, and “Returning Assertion” smells are not related to JUnit tests, and therefore, do not occur on tests generated by EvoSuite.

B. Test smells that cannot be computed

Besides the 30 smells that do not affect the tests generated by EvoSuite, there are other eight test smells for which we could not find a computational metric in the literature or derive a way to efficiently compute it. Column 'Metric' in Table I lists those smells.

No metric: "Anonymous Test" and "Test-Class Name" require a human developer to assess whether the name of a test is (or not) meaningful. Despite recent advances [14], there is not yet a metric to automatically compute how meaningful a name might be. "Brittle Assertion" requires the usage of dynamic tainting [23, 29] (unavailable in EvoSuite). "Frequent Debugging" requires a manual analysis to check whether the root cause of a failure is unintuitive. "Proper Organization" requires a precise procedure, which is not available, to compute a subjective concept such as "organization".

Resource intensive metrics: "Fragile Test" requires changes to the code under test and the execution of the same test multiple times. "Slow Tests" requires the execution of the same test multiple times to assess its average runtime. "Test Code Duplication" requires the implementation and execution of, e.g., similarity metrics such as the Levenshtein distance, to assess whether there are repeated or similar statements in a test. Overall, these metrics would likely hamper the test generation process as they are very time-consuming.

C. Test smells that can be computed

Below, we describe the computational metrics of the remaining 16 test smells and the respective thresholds (either recommended in the literature or derived by us). A test case is smelly if the smelliness of the respective metric is greater than or equal to the established threshold.

Assertion Roulette (AR)
Metric: Number of assertions in a test that exceed the total number of statements that call methods of the class under test.
Threshold: 3 [19].

Duplicate Assert (DA)
Metric: Number of assertion statements of the same type that check the same method of the same class and have the same expected value.
Threshold: 1 [21, 30].

Eager Test (ET)
Metric: Total number of different methods of the class under test that are being exercised by a test.
Threshold: 4 [19].

Indirect Testing (IT)
Metric: Total number of methods of other classes (i.e., other than the class under test) that are being exercised by a test.
Threshold: 1 [17, 16].

Lack Of Cohesion Of Methods (LCM)
Metric: Number of test cases that do not exercise the class under test.
Threshold: 1 (a test that does not exercise the class under test can be considered pointless regarding the verification of the behavior of the class under test).

Lazy Test (LT)
Metric: Number of times a method of the class under test is called by more than one test.
Threshold: 1 [17, 16, 21, 30].

Likely Ineffective Object-Comparison (LIOC)
Metric: Number of times the "equals" method of a class other than the one under test is used to compare an object with itself.
Threshold: 1 (it only makes sense to use the "equals" method to compare an object with itself if the class under test implements said "equals" method).

Obscure In-line Setup (OISS)
Metric: Number of declared variables in a test (note that this metric does not consider the variables that store values returned from methods of the class under test).
Threshold: 10 [22].

Overreferencing (OF)
Metric: Number of class instances that are created but never used.
Threshold: 1 (every object created in a test should have a given purpose and, as such, should be used at least once).

Redundant Assertion (RA)
Metric: Number of assertions that check primitive statements.
Threshold: 1 [21, 30].

Rotten Green Tests (RGT)
Metric: Number of statements that exist after the statement that raises the first exception in a given test.
Threshold: 1 (any code after the first statement that raises an exception will not be executed; thus, it should be removed).

Test Redundancy (TR)
Metric: Number of tests that can be removed from the test suite without decreasing the suite's code coverage.
Threshold: 1 (tests that do not contribute to increase coverage serve no purpose, according to EvoSuite's main goal, and should therefore be considered redundant and discarded from the final test suite).

Unknown Test (UT)
Metric: Number of assertions in a test.
Threshold: 1 [21, 30].

Unrelated Assertions (UA)
Note: This test smell corresponds to an adaptation of the "Sensitive Equality" smell. We adapted its name because our newly proposed metric ended up diverging too far from the original definition.
Metric: Total number of assertions that check methods that are not declared in the class under test.
Threshold: 1 (assertions that check methods not declared in the class under test may be misleading and serve no purpose).

Unused Inputs (UI)
Metric: Number of assertionless statements that call methods (that also return values) of the class under test.

Table II: Diffusion of test smells on the tests generated by the EvoSuite tool. Columns $\bar{x}$, standard deviation (σ), and confidence intervals (CI) using bootstrapping at 95% significance level, report the distribution of test smell metrics.

Metric	$\bar{x}$	σ	CI
AssertionRoulette	2.66%	0.09	[0.02, 0.04]
DuplicateAssert	0.58%	0.04	[0.00, 0.01]
EagerTest	3.98%	0.12	[0.03, 0.05]
IndirectTesting	34.79%	0.27	[0.32, 0.38]
LackOfCohesionOfMethods	0.32%	0.06	[0.00, 0.01]
LazyTest	0.00%	0.00	[0.00, 0.00]
LikelyIneffectiveObjectComparison	0.01%	0.00	[0.00, 0.00]
ObscureInlineSetup	1.26%	0.05	[0.01, 0.02]
Overreferencing	5.12%	0.15	[0.03, 0.07]
RedundantAssertion	0.02%	0.00	[0.00, 0.00]
RottenGreenTests	0.81%	0.03	[0.00, 0.01]
TestRedundancy	0.00%	0.00	[0.00, 0.00]
UnknownTest	45.91%	0.27	[0.43, 0.49]
UnrelatedAssertions	16.19%	0.20	[0.14, 0.18]
UnusedInputs	25.85%	0.24	[0.23, 0.28]
VerboseTest	1.32%	0.05	[0.01, 0.02]
Average	8.68%	0.10	[0.08, 0.10]

Threshold: 1 (a statement that calls a method of the class under test that returns a value should necessarily have an assertion to capture the current behavior of the system under test).

Verbose Test (VT)
Metric: Total number of statements in a test.
Threshold: 13 [19].

D. Test smells that affect tests generated by EvoSuite

To investigate the extent to which the tests generated by EvoSuite are affected by the 16 test smells, we conducted an experiment on a set of 346 Java classes (see Section V-A for more information). We first implemented the metrics for the 16 smells on the latest version of EvoSuite. Then, we ran EvoSuite 30 times (as suggested by Arcuri et al. [31]) on each class, using EvoSuite's default settings, but setting up a search budget of 180 seconds as others have done [2]. Finally, we investigated the diffusion of smells in the generated tests.

Table II reports the diffusion of test smells on the tests generated by the EvoSuite tool. On average, 8.68% of all tests generated by EvoSuite are smelly. On one hand, the high percentage of tests affected by "Unknown Test" and "Unused Inputs" is most likely related to the existence of tests with try/catch exceptions in the tests instead of assert statements. Moreover, as stated by others [25, 27], EvoSuite generates many tests with assertions that check methods not declared in the class under test. Thus, the high percentage of tests affected by the "Unrelated Assertions" smell is most likely related to the high diffusion of the "Indirect Testing" smell. That is, tests that exercise methods not declared in the class under test are also likely to have assertions for those methods. On the other hand, the "Assertion Roulette", "Duplicate Assert", and "Redundant Assertion" smells only affected a small fraction of the generated tests: 2.66%, 0.58%, and 0.02%, respectively.

IV. Approach

Although the primary objective of EvoSuite is to maximize code coverage, others have already integrated non-coverage metrics as **secondary criteria** into the tool [11, 2] to improve the usefulness and quality of the generated tests. By default, EvoSuite uses a secondary non-coverage-based criterion that promotes tests that are as short as possible [32]. Moreover, after exhausting the search budget or achieving 100% code coverage, EvoSuite applies several **post-processing** steps to improve the quality and readability of the generated tests. For example, primitive values and null references are inlined, redundant tests and statements (which do not contribute to the final code coverage) are removed, and a minimized set of assertions is added to each test.

Palomba et al. [11] proposed a secondary criterion that allows EvoSuite to generate more cohesive and less coupled tests. Moreover, it leads to shorter tests that also achieve higher coverage (less likely early convergence).

Grano et al. [2] proposed an adaptive search algorithm, aDynaMOSA (extension of the DynaMOSA), which optimizes the runtime and memory consumption of the tests as secondary criteria. According to their results, aDynaMOSA generates less expensive tests (decreased runtime and memory consumption) with higher coverage and mutation score than DynaMOSA.

Palomba et al. [11] and Grano et al. [2] have demonstrated the viability of incorporating quality metrics into EvoSuite through the usage of secondary criteria. Thus, we hypothesize that it might be possible to use secondary criteria to optimize test smells with minimal impact (if any) on the final code coverage and/or fault ability of the generated tests.

A. Implementation

Let t_a and t_b designate two tests under evolution, and let $S = \{s_1, ..., s_N\}$ be the set of test smell metrics instantiated as secondary criteria. If both t_a and t_b cover the same target (e.g., line), EvoSuite uses the secondary criteria to either select the test that should form the next population or update the archive. Given S, we compare the smelliness of t_a and t_b as follows:

$$\text{compare}(t_a, t_b, S) = \sum_{s_k \in S} s_k(t_a) - s_k(t_b) \qquad (1)$$

where $s_k(t) \in [0, 1]$ denotes the smelliness of the metric k for the test t. After iterating over S, if $\text{compare}(t_a, t_b, S) < 0$, t_a is less smelly than t_b, and therefore, t_a is preferred.

Given EvoSuite's limited search budget, we have to ensure that the secondary criteria are computed as fast as possible. Thus, each test stores the results of its computed test smell metrics, i.e., unless a test is modified by the search procedure, the metrics are only computed once.

B. Perils

When it comes to use our approach, there are a few perils, which we describe in detail below.

1) Test smell metrics that cannot be directly optimized as secondary criteria: Seven out of the 14 considered test smell metrics cannot be computed during the search procedure and, as such, cannot be directly optimized as secondary criteria. In other words, we cannot optimize the six assertion-related test smells and one smell evaluated at the test suite level ("Lack of

Cohesion of Methods"). Still, as demonstrated by Grano et al. [3], the presence of specific test smells may imply the presence of other test smells. Thus, test smell metrics not optimized as secondary criteria can be indirectly optimized.

2) More or less code coverage: The optimization of test smell metrics might negatively/positively affect the coverage achieved by the generated tests due to "Genetic Drift" [33]. Genetic drift arises when the individuals of a population become too similar, thus diminishing the ability to explore the search space. Panichella et al. [32] demonstrated that this problem arises even when using test case length as the secondary criterion — during the search process, complex and large tests have higher evolutionary potential, i.e., they are more capable of change and promote diversity. As our secondary criteria focus on test smells, we expect the size of the tests to increase, which may promote more diversity and, therefore, increase coverage. Still, the optimization of certain smells might also promote less diversity.

3) Fewer generations of the underline evolution algorithm: The computation of test smell metrics is slower than the computation of EvoSuite's default secondary criterion (that just counts the number of statements per test case). This might decrease the number of generations the evolutionary algorithm can perform on a given search budget. With less time to evolve, the coverage is likely to be lower. This problem is unavoidable as the computation of test smell metrics takes longer (given its complexity) than the procedure to compute the length of a test. Still, we carefully developed each metric to avoid a significant impact (if any) on the effectiveness of the generated tests.

V. Empirical Study

We aim to investigate the following research questions:

RQ1: To what extent EvoSuite's default mechanisms, i.e., verbose test as secondary criteria and test minimization, lead to the generation of less smelly tests?

RQ2: Does the optimization of test smell metrics lead to the generation of fewer smelly tests?

RQ3: Does the optimization of test smell metrics affect the code coverage and fault detection effectiveness of the generated tests?

Firstly, we aim to shed light on EvoSuite's default mechanisms to improve the readability of the generated tests and likely their smelliness (**RQ1**). Secondly, we aim to investigate whether the optimization of the seven test smell metrics (Eager Test, Indirect Testing, Likely Ineffective Object Comparison, Obscure InlineSetup, Overreferencing, Rotten Green Tests, and Verbose Test) leads to fewer smelly tests (**RQ2**). Finally, in **RQ3**, we aim to investigate whether the coverage and fault detection effectiveness of the generated tests is affected by the optimization of test smell metrics.

A. Experimental Subjects

Previous studies have shown that the quality and complexity of the code under test can influence the presence of test smells in the tests generated by EvoSuite [3, 12]. Thus, in this study, we use a set of 346 non-trivial Java classes extracted from 117 open-source projects [34]. This corpus has been used to evaluate different test generation techniques [35, 32].

Given that complex classes typically imply the generation of smellier tests, experiments on this corpus should (1) provide insight into the presence of smells in automatically generated tests (relevant for the experiment performed in Section III-D) and (2) allow us to more thoroughly evaluate the capabilities of the proposed approach to optimize test smell metrics.

B. Experimental Metrics

In each execution of EvoSuite, we collected the code coverage and mutation score of the generated tests along with the value of each smell metric. To investigate the diffusion of test smells, we compute the percentage of tests affected by a specific test smell. In detail, we apply the threshold of each test smell metric s to each test t in a given test suite T, and calculate the percentage of test cases affected by s as $100 \times \frac{1}{|T|} \sum_{t \in T} \begin{cases} 1 & \text{if } t(s) \text{ is above or equal to threshold.} \\ 0 & \text{if } t(s) \text{ is below threshold.} \end{cases}$ Additionally, we also report relative improvements. Given two sets of (coverage, mutation score, smelliness) values, one of configuration A and another of configuration B, the relative average improvement is defined as $\frac{mean(A) - mean(B)}{mean(B)}$.

C. Experimental Procedure

We ran EvoSuite with its default settings on the selected corpus, that is, (1) DynaMOSA as the search algorithm [32] and (2) the default fitness function that includes: line, branch, exception, weak mutation, output, method, method exception, and cbranch coverage. We only modified EvoSuite's search budget default value from 60 to 180 seconds, as suggested by others [2]. Also, given that EvoSuite's underlying algorithm is randomized, we repeated each execution of EvoSuite 30 times, as suggested by Arcuri et al. [31]. All experiments were executed on the *redacted for anonymity*.

In **RQ1** we investigate the impact of EvoSuite's default secondary criteria (i.e., verbose test) and it's minimization procedure at reducing the number of smelly tests. In detail, to answer this research question we considered four configurations of EvoSuite:

- Conf-A: EvoSuite with no secondary criteria[1] and minimization disabled.
- Conf-B: EvoSuite with no secondary criteria and minimization enabled.
- Conf-C: EvoSuite with default secondary criteria (i.e., verbose test) and minimization disabled.
- Vanilla: EvoSuite's default configuration, i.e, default secondary criteria (i.e., verbose test) and minimization enabled.

where each configuration was executed 30 times on the set of 346 classes. We then performed pairwise comparisons between all configurations.

In **RQ2** we investigate to what extend the optimization of test smell metrics if effective at generating less smelly tests.

[1]It is not trully possible to disable EvoSuite's secondary criteria due to how DynaMOSA operates, thus we developed a random-based secondary-criteria which selects at random one solution instead.

In detail, to answer this research question we considered one additional configuration of EvoSuite:

- SMELLESS: EvoSuite's secondary criteria configured with the combination of all smell metrics that could be optimized as a secondary criteria (i.e., Eager Test, Indirect Testing, Likely Ineffective Object Comparison, Obscure InlineSetup, Overreferencing, Rotten Green Tests, and Verbose Test).

and executed it 30 times on the same set of 346 classes. We then performed a pairwise comparison to assess whether the SMELLESS configuration leads to the generation of tests that are less smelly than the ones generated by VANILLA.

In **RQ3** we perform a pairwise comparison between VANILLA and SMELLESS and assess whether the SMELLESS configuration generates tests that are as effective (in terms of coverage and mutation score) as those generated by the VANILLA configuration.

D. Experimental Analysis

We use the Vargha-Delaney ($\hat{A}_{12}$) effect size to determine whether a configuration A performs better than a configuration B. We also use the Wilcoxon-Mann-Whitney test with a significance level of 95% to assess whether the difference in performance between two configurations is statistically significant.

E. Threats to Validity

External Validity: We conducted our investigation on a corpus of 346 Java classes from 117 open-source Java projects. Our results may not generalize to other classes/projects (e.g., industrial systems), but we attempt to minimize this threat by using the largest and most diverse set of classes available that others have used. Also, our results and conclusions are limited to the tests generated by one single tool: EvoSuite. Although other tools have been proposed (e.g., Randoop [36]), EvoSuite is the only tool that already supports other secondary criteria and therefore allows us to, easily, implement our novel secondary criteria.

Internal Validity: Given that EvoSuite is randomized, it is necessary to run repetitions and do a statistical analysis of the data. To minimize this threat, we repeat each experiment 30 times, as suggested by Arcuri and Briand [31]. Any change performed in EvoSuite and all the scripts developed to perform the statistical analysis were reviewed by all the authors and formally tested—we have created unit tests for all implemented test smell metrics.

Construct Validity: We optimize test smell metrics inspired by the available definitions. Furthermore, when possible, we adapt test smell metric implementations and thresholds from tools with available source code.

VI. RESULTS

RQ1: EvoSuite's default mechanisms

Table III reports the diffusion of test smells on the tests generated by the CONF-A configuration vs. CONF-B, CONF-C, and VANILLA configurations, on the 63 classes under test for which all configurations achieved similar coverage.

On average, 37.13% of the tests generated by EvoSuite are considered smelly tests when it is configured without any of its mechanisms to improve the readability of the generated tests (i.e., CONF-A). The top-3 most diffused smells are VerboseTest (91.00%), Indirect Testing (84.98%), and Obscure Inline Setup (84.44%). None of the tests generated exhibits the Lack of Cohesion of Methods, Lazy Test, and Test Redundancy smells. Regarding the results achieved by the other configurations:

- When EvoSuite's post-procedure to minimize the generated tests is enabled (i.e., CONF-B), the number of smelly tests is statistically significantly reduced to 13.05% (-35.02%).
- When EvoSuite's default secondary criteria is enabled (i.e., CONF-C), the number of smelly tests is statistically significantly reduced to 20.31% (-24.73%).
- When EvoSuite is initialized with it's default configuration (i.e., VANILLA), the number of smelly tests is statistically significantly reduced to 9.13% (-35.35%).

> **RQ1:** EvoSuite's default mechanisms generates statistically significantly less smelly tests than EvoSuite with none of the mechanisms (-35.35%).

RQ2: Optimization of Test Smells

Table IV reports the diffusion of test smells on the tests generated by the SMELLESS configuration. On average, 8.58% of the test generated by the SMELLESS configuration are smelly, if we take into account all 16 smells (directy optimized and non-optimized), and only 6.52% are smelly if we only considered the seven optimized smells (highlighted in gray in the table). Worth noting that although Indirect Testing is optimized by SMELLESS, 33.05% of all generated tests are smelly. Similar values were reported in RQ1 for the VANILLA (see Table III). In the remaining six optimized smells, the % of smelly tests is less than 4%.

Table V reports the diffusion of test smells on the tests generated by the VANILLA configuration vs. the SMELLESS configuration, on the 165 classes under test for which both configurations achieved similar coverage. On one hand, the tests generated by the SMELLESS configurations are less smelly than the tests generated by VANILLA in four out of the seven smell metrics optimized, i.e., Indirect Testing, Likely Ineffective Object Comparison, Overreferencing, and Rotten Green Tests. On the other hand, tests generated by SMELLESS are smellier than the ones generated by VANILLA in the remaining three smells, i.e., Eager Test, Obscure Inline Setup, and Verbose Test.

Overall, the SMELLESS configuration generated fewer smelly tests if only the set of optimized smells is considered (-4.14%) or if all smells are considered (-2.61%). The performance achieved by the SMELLESS configuration is marginal, yet relevant.

> **RQ2:** On average, SMELLESS generates -2.61% smelly tests than VANILLA.

Table III: Diffusion of test smells on the tests generated by the CONF-A configuration vs. CONF-B, CONF-C, and VANILLA configurations, on the 63 classes under test for which all configurations achieved similar coverage. Column $\bar{x}$ reports the ratio of smelly tests generated by each configuration. $\hat{A}_{12}$ reports the effect size of X vs. Y. Note that statistically significantly effect size values, i.e., p-value ≤ 0.05, are annotated in **bold face**. Column 'Rel. impr.' reports the relative improvement of X over Y regarding the percentage of smelly tests generated by both configurations.

	CONF-A	CONF-B			CONF-C			VANILLA		
Metric	$\bar{x}$	$\bar{x}$	$\hat{A}_{12}$	Rel. impr.	$\bar{x}$	$\hat{A}_{12}$	Rel. impr.	$\bar{x}$	$\hat{A}_{12}$	Rel. impr.
AssertionRoulette	0.90%	2.37%	0.44	163.47%	1.82%	0.43	102.10%	3.24%	0.44	260.29%
DuplicateAssert	1.71%	0.49%	0.60	-71.14%	0.88%	0.55	-48.90%	0.36%	0.62	-79.23%
EagerTest	51.47%	6.57%	**0.88**	-87.23%	20.51%	**0.87**	-60.15%	3.36%	**0.89**	-93.47%
IndirectTesting	84.98%	51.20%	**0.95**	-39.76%	56.08%	**0.95**	-34.00%	37.90%	**0.98**	-55.41%
LackOfCohesionOfMethods	0.00%	0.00%	**0.50**	0.00%	0.00%	**0.50**	0.00%	0.00%	**0.50**	0.00%
LazyTest	0.00%	0.00%	**0.50**	0.00%	0.00%	**0.50**	0.00%	0.00%	**0.50**	0.00%
LikelyIneffectiveObjectComparison	0.24%	0.05%	**0.51**	-77.18%	0.06%	**0.51**	-73.99%	0.03%	**0.51**	-85.97%
ObscureInlineSetup	84.44%	6.21%	**1.00**	-92.64%	32.06%	**0.99**	-62.03%	1.15%	**1.00**	-98.64%
Overreferencing	51.60%	18.99%	**0.94**	-63.20%	20.62%	**0.95**	-60.05%	5.16%	**0.97**	-89.99%
RedundantAssertion	1.74%	0.00%	**0.69**	-99.84%	1.04%	0.54	-40.07%	0.00%	**0.69**	-99.85%
RottenGreenTests	57.98%	7.08%	**0.97**	-87.79%	27.63%	**0.95**	-52.35%	1.51%	**0.98**	-97.40%
TestRedundancy	0.00%	0.00%	**0.50**	0.00%	0.00%	**0.50**	0.00%	0.00%	**0.50**	0.00%
UnknownTest	73.62%	51.17%	0.90	-30.49%	56.58%	**0.90**	-23.14%	46.42%	**0.92**	-36.95%
UnrelatedAssertions	9.55%	16.14%	0.30	68.93%	14.32%	0.31	49.95%	16.77%	0.31	75.57%
UnusedInputs	84.89%	42.19%	**0.95**	-50.31%	56.00%	**0.96**	-34.03%	29.22%	**0.96**	-65.59%
VerboseTest	91.00%	6.29%	**1.00**	-93.09%	37.31%	**1.00**	-59.00%	0.94%	**1.00**	-98.97%
Average	37.13%	13.05%	**0.73**	-35.02%	20.31%	**0.71**	-24.73%	9.13%	**0.74**	-35.35%

Table IV: Diffusion of test smells on the tests generated by the SMELLESS configuration. The rows highlighted in gray correspond to the smells metrics optimized by SMELLESS. Columns $\bar{x}$, standard deviation (σ), and confidence intervals (CI) using bootstrapping at 95% significance level, report the distribution of test smell metrics.

Metric	$\bar{x}$	σ	CI
AssertionRoulette	2.68%	0.09	[0.02, 0.04]
DuplicateAssert	0.62%	0.04	[0.00, 0.01]
EagerTest	3.98%	0.12	[0.03, 0.05]
IndirectTesting	33.05%	0.27	[0.30, 0.36]
LackOfCohesionOfMethods	0.33%	0.06	[-0.01, 0.01]
LazyTest	0.00%	0.00	[0.00, 0.00]
LikelyIneffectiveObjectComparison	0.01%	0.00	[0.00, 0.00]
ObscureInlineSetup	1.58%	0.05	[0.01, 0.02]
Overreferencing	4.69%	0.15	[0.03, 0.06]
RedundantAssertion	0.02%	0.00	[0.00, 0.00]
RottenGreenTests	0.78%	0.03	[0.00, 0.01]
TestRedundancy	0.00%	0.00	[0.00, 0.00]
UnknownTest	45.97%	0.27	[0.43, 0.49]
UnrelatedAssertions	16.31%	0.20	[0.14, 0.18]
UnusedInputs	25.76%	0.24	[0.23, 0.28]
VerboseTest	1.54%	0.06	[0.01, 0.02]
Average (optimized smells)	6.52%	0.10	[0.05, 0.07]
Average (all smells)	8.58%	0.10	[0.07, 0.10]

Table V: Diffusion of test smells on the tests generated by the VANILLA configuration vs. the SMELLESS configuration, on the 165 classes under test for which both configurations achieved similar coverage.

Metric	VANILLA	SMELLESS	$\hat{A}_{12}$	Rel. impr.
AssertionRoulette	2.00%	2.10%	0.49	5.26%
DuplicateAssert	0.26%	0.23%	0.50	-10.31%
EagerTest	3.58%	3.70%	0.49	3.39%
IndirectTesting	35.97%	34.37%	0.57	-4.44%
LackOfCohesionOfMethods	0.00%	0.00%	**0.50**	0.00%
LazyTest	0.00%	0.00%	**0.50**	0.00%
LikelyIneffectiveObjectComparison	0.02%	0.01%	**0.50**	-39.45%
ObscureInlineSetup	1.38%	1.64%	0.48	18.68%
Overreferencing	4.16%	3.63%	0.52	-12.66%
RedundantAssertion	0.03%	0.03%	**0.50**	-6.90%
RottenGreenTests	0.71%	0.68%	0.50	-3.97%
TestRedundancy	0.00%	0.00%	**0.50**	0.00%
UnknownTest	45.97%	45.95%	0.51	-0.03%
UnrelatedAssertions	17.39%	17.51%	0.49	0.68%
UnusedInputs	28.17%	27.76%	0.50	-1.43%
VerboseTest	1.55%	1.69%	0.49	9.46%
Average (optimized smells)	6.77%	6.53%	0.51	-4.14%
Average (all smells)	8.82%	8.71%	0.50	-2.61%

RQ3: Impact on the coverage and fault detection effectiveness

On average, tests generated by VANILLA and SMELLESS achieved (1) similar coverage, 77.73% vs. 77.30% and (2) similar mutation score, 34.98% vs. 35.93%. Regarding coverage and mutation score, both configurations did not perform statistically significantly differently, $\hat{A}_{12} = 0.55$ with a p-value 0.376, and $\hat{A}_{12} = 0.53$ with a p-value 0.391, respectively.

RQ3: On average, the optimization of test smell metrics reduces the coverage achieved by the generated tests by -0.01% and their fault detection effectiveness by -0.03%.

VII. CONCLUSIONS

Others have shown that automatically generated tests (e.g., those generated by EvoSuite) are affected by test smells, i.e., bad programming practices. Thus, we first gathered all test smells described in the literature, filtered out the ones that were not applicable or that could not be automatically computed, and identified the ones that did affect the EvoSuite's tests. We then implemented 16 test smell metrics into the tool and performed an empirical study on 346 classes. We observed that "Unknown Test", "Indirect Testing", and "Unused Inputs" are the most diffused smells among all generated tests.

Then, and to be able to generate smell-free tests out-of-the-box, we augmented EvoSuite with a new secondary criteria that optimize test smell metrics and repeat the study. Our results indicate that: (1) the number of smelly tests was reduced by 3% when compared with EvoSuite's default, and (2) the generated tests have similar coverage and fault detection effectiveness to those generated by EvoSuite's default version.

As future work, we intend to investigate the optimization of assertion-based test smell metrics (e.g., "Unknown Test") as part of EvoSuite's post-processing procedure.

Acknowledgments: This work was supported by FCT through the LASIGE Research Unit, ref. UIDB/00408/2020 and ref. UIDP/00408/2020.

References

[1] Glenford J Myers, Corey Sandler, and Tom Badgett. *The art of software testing*. John Wiley & Sons, 2011.

[2] Giovanni Grano, Christoph Laaber, Annibale Panichella, and Sebastiano Panichella. "Testing with fewer resources: An adaptive approach to performance-aware test case generation". In: *IEEE Transactions on Software Engineering* (2019).

[3] Giovanni Grano, Fabio Palomba, Dario Di Nucci, Andrea De Lucia, and Harald C Gall. "Scented since the beginning: On the diffuseness of test smells in automatically generated test code". In: *Journal of Systems and Software* 156 (2019), pp. 312–327.

[4] Gordon Fraser and Andrea Arcuri. "Evosuite: automatic test suite generation for object-oriented software". In: *Proc. of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. 2011, pp. 416–419.

[5] Andrea Arcuri, José Campos, and Gordon Fraser. "Unit Test Generation During Software Development: EvoSuite Plugins for Maven, IntelliJ and Jenkins". In: *2016 IEEE International Conference on Software Testing, Verification and Validation, ICST 2016, Chicago, IL, USA, April 11-15, 2016*. IEEE Computer Society, 2016, pp. 401–408.

[6] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. "A large scale empirical comparison of state-of-the-art search-based test case generators". In: *Information and Software Technology* 104 (2018).

[7] Sina Shamshiri, René Just, José Miguel Rojas, Gordon Fraser, Phil McMinn, and Andrea Arcuri. "Do Automatically Generated Unit Tests Find Real Faults? An Empirical Study of Effectiveness and Challenges". In: *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2015, pp. 201–211.

[8] M. Moein Almasi, Hadi Hemmati, Gordon Fraser, Andrea Arcuri, and Jundefinednis Benefelds. "An Industrial Evaluation of Unit Test Generation: Finding Real Faults in a Financial Application". In: *Proc. of the 39th International Conference on Software Engineering: Software Engineering in Practice Track*. ICSE-SEIP '17. Buenos Aires, Argentina: IEEE Press, 2017, 263–272.

[9] Sebastiano Panichella, Annibale Panichella, Moritz Beller, Andy Zaidman, and Harald C. Gall. "The Impact of Test Case Summaries on Bug Fixing Performance: An Empirical Investigation". In: *Proc. of the 38th International Conference on Software Engineering*. ICSE '16. Austin, Texas: Association for Computing Machinery, 2016, 547–558.

[10] Mozhan Soltani, Annibale Panichella, and Arie van Deursen. "Search-Based Crash Reproduction and Its Impact on Debugging". In: *IEEE Transactions on Software Engineering* 46.12 (2020), pp. 1294–1317.

[11] Fabio Palomba, Annibale Panichella, Andy Zaidman, Rocco Oliveto, and Andrea De Lucia. "Automatic test case generation: What if test code quality matters?" In: *Proc. of the 25th International Symposium on Software Testing and Analysis*. 2016, pp. 130–141.

[12] Fabio Palomba, Dario Di Nucci, Annibale Panichella, Rocco Oliveto, and Andrea De Lucia. "On the diffusion of test smells in automatically generated test code: An empirical study". In: *IEEE/ACM 9th International Workshop on Search-Based Software Testing (SBST)*. 2016.

[13] Arie Van Deursen, Leon Moonen, Alex Van Den Bergh, and Gerard Kok. "Refactoring test code". In: *Proc. of the 2nd international conference on extreme programming and flexible processes in software engineering (XP2001)*. Citeseer. 2001, pp. 92–95.

[14] Ermira Daka, José Campos, Gordon Fraser, Jonathan Dorn, and Westley Weimer. "Modeling readability to improve unit tests". In: *Proc. of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 2015, pp. 107–118.

[15] Sina Shamshiri, José Miguel Rojas, Juan Pablo Galeotti, Neil Walkinshaw, and Gordon Fraser. "How do automatically generated unit tests influence software maintenance?" In: *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*.

[16] Gabriele Bavota, Abdallah Qusef, Rocco Oliveto, Andrea De Lucia, and David Binkley. "An empirical analysis of the distribution of unit test smells and their impact on software maintenance". In: *2012 28th IEEE International Conference on Software Maintenance (ICSM)*.

[17] Gabriele Bavota, Abdallah Qusef, Rocco Oliveto, Andrea De Lucia, and Dave Binkley. "Are test smells really harmful? an empirical study". In: *Empirical Software Engineering* 20.4 (2015), pp. 1052–1094.

[18] Davide Spadini, Fabio Palomba, Andy Zaidman, Magiel Bruntink, and Alberto Bacchelli. "On the relation of test smells to software code quality". In: *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE. 2018, pp. 1–12.

[19] Davide Spadini, Martin Schvarcbacher, Ana-Maria Oprescu, Magiel Bruntink, and Alberto Bacchelli. "Investigating severity thresholds for test smells". In: *Proc. of the 17th International Conference on Mining Software Repositories*. 2020, pp. 311–321.

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

[21] Anthony Peruma, Khalid Saeed Almalki, Christian D Newman, Mohamed Wiem Mkaouer, Ali Ouni, and Fabio Palomba. "On the Distribution of Test Smells in Open Source Android Applications: An Exploratory Study". In: *Proc. of the 29th Annual International Conference on Computer Science and Software Engineering*. 2019.

[22] Michaela Greiler, Arie Van Deursen, and Margaret-Anne Storey. "Automated detection of test fixture strategies and smells". In: *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*. IEEE. 2013, pp. 322–331.

[23] Chen Huo and James Clause. "Improving oracle quality by detecting brittle assertions and unused inputs in tests". In: *Proc. of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 2014, pp. 621–631.

[24] Michele Tufano, Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, and Denys Poshyvanyk. "An empirical investigation into the nature of test smells". In: *Proc. of the 31st IEEE/ACM International Conference on Automated Software Engineering*. 2016, pp. 4–15.

[25] Annibale Panichella, Sebastiano Panichella, Gordon Fraser, Anand Ashok Sawant, and Vincent J. Hellendoorn. "Revisiting Test Smells in Automatically Generated Tests: Limitations, Pitfalls, and Opportunities". In: *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2020, pp. 523–533.

[26] Anthony Peruma, Khalid Almalki, Christian D Newman, Mohamed Wiem Mkaouer, Ali Ouni, and Fabio Palomba. "Tsdetect: An open source test smells detection tool". In: *Proc. of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2020.

[27] Annibale Panichella, Sebastiano Panichella, Gordon Fraser, Anand Ashok Sawant, and Vincent J. Hellendoorn. *Test Smells 20 Years Later: Detectability, Validity, and Reliability*. 2022.

[28] Wajdi Aljedaani, Anthony Peruma, Ahmed Aljohani, Mazen Alotaibi, Mohamed Wiem Mkaouer, Ali Ouni, Christian D Newman, Abdullatif Ghallab, and Stephanie Ludi. "Test Smell Detection Tools: A Systematic Mapping Study". In: *Evaluation and Assessment in Software Engineering* (2021), pp. 170–180.

[29] Junhyoung Kim, TaeGuen Kim, and Eul Gyu Im. "Survey of dynamic taint analysis". In: *2014 4th IEEE International Conference on Network Infrastructure and Digital Content*. IEEE. 2014, pp. 269–272.

[30] Anthony Peruma, Mohamed Wiem Mkaouer, Khalid Almalki, Christian D. Newman, Ali Ouni, and Fabio Palomba. *Software Unit Test Smells*. https://testsmells.org. Accessed: 2022-03-25.

[31] Andrea Arcuri and Lionel Briand. "A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering". In: *Software Testing, Verification and Reliability* 24.3 (2014), pp. 219–250.

[32] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. "Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets". In: *IEEE Transactions on Software Engineering* 44.2 (2017), pp. 122–158.

[33] Fitsum M Kifetew, Annibale Panichella, Andrea De Lucia, Rocco Oliveto, and Paolo Tonella. "Orthogonal exploration of the search space in evolutionary test case generation". In: *Proc. of the 2013 International Symposium on Software Testing and Analysis*.

[34] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. *Non-trivial Java Classes to Study the Performance of Search-based Software Testing Approaches*. https://github.com/jose/non-trivial-java-classes-to-study-search-based-software-testing-approaches. Accessed: 2022-03-25.

[35] José Campos, Yan Ge, Nasser Albunian, Gordon Fraser, Marcelo Eler, and Andrea Arcuri. "An empirical evaluation of evolutionary algorithms for unit test suite generation". In: *Information and Software Technology* 104 (2018), pp. 207–235.

[36] Carlos Pacheco and Michael D Ernst. "Randoop: feedback-directed random testing for Java". In: *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*. 2007, pp. 815–816.