Automatic Generation of Smell-free Unit Tests

João Afonso¹, José Campos¹,²

¹LASIGE, Faculdade de Ciências, Universidade de Lisboa, Lisboa, Portugal
²Faculty 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 extent 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 JTest. 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 test 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.

Panicella 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, Panicella 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. Panicella 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 tests 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 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).

#### 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

<table>
<thead>
<tr>
<th>Name</th>
<th>Abbr.</th>
<th>Affect?</th>
<th>Metric?</th>
<th>Sec. Criteria?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abnormal UTF-Use</td>
<td>AUU</td>
<td>NO</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Anonymous Test</td>
<td>AT</td>
<td>YES</td>
<td>NO</td>
<td>—</td>
</tr>
<tr>
<td>Assertion Roulette</td>
<td>AR</td>
<td>YES</td>
<td>NO</td>
<td>—</td>
</tr>
<tr>
<td>Brittle Assertion</td>
<td>BA</td>
<td>NO</td>
<td>NO</td>
<td>—</td>
</tr>
<tr>
<td>Conditional Test Logic</td>
<td>CTL</td>
<td>NO</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Constructor Initialization</td>
<td>CI</td>
<td>NO</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Dead Field</td>
<td>DF</td>
<td>NO</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Default Test</td>
<td>DT</td>
<td>NO</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Duplicate Assert</td>
<td>DA</td>
<td>YES</td>
<td>YES</td>
<td>—</td>
</tr>
<tr>
<td>Eager Test</td>
<td>ET</td>
<td>YES</td>
<td>YES</td>
<td>—</td>
</tr>
<tr>
<td>Empty Shared-Fixture</td>
<td>ESF</td>
<td>NO</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Empty Test</td>
<td>EmT</td>
<td>NO</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Erratic Test</td>
<td>ErT</td>
<td>NO</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Exception Handling</td>
<td>EH</td>
<td>NO</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>For Testers Only</td>
<td>FTO</td>
<td>NO</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Fragile Test</td>
<td>FT</td>
<td>YES</td>
<td>NO</td>
<td>—</td>
</tr>
<tr>
<td>Frequent Debugging</td>
<td>FD</td>
<td>YES</td>
<td>NO</td>
<td>—</td>
</tr>
<tr>
<td>General Fixture</td>
<td>GF</td>
<td>NO</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Hard-to-Test Code</td>
<td>HTTC</td>
<td>NO</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Ignored Test</td>
<td>IgT</td>
<td>NO</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Indirect Testing</td>
<td>IT</td>
<td>YES</td>
<td>YES</td>
<td>—</td>
</tr>
<tr>
<td>Lack of Cohesion of Methods</td>
<td>LCM</td>
<td>NO</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Lazy Test</td>
<td>LT</td>
<td>YES</td>
<td>NO</td>
<td>—</td>
</tr>
<tr>
<td>Likely Ineffective Test-Comparison</td>
<td>LITOC</td>
<td>YES</td>
<td>YES</td>
<td>—</td>
</tr>
<tr>
<td>Magic Number Test</td>
<td>MNT</td>
<td>NO</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Manual Intervention</td>
<td>MI</td>
<td>NO</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Mixed Selectors</td>
<td>MS</td>
<td>NO</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Mystery Guest</td>
<td>MG</td>
<td>NO</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Non-Java Smells</td>
<td>NJS</td>
<td>NO</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Obsolete In-line Setup</td>
<td>OBIS</td>
<td>YES</td>
<td>YES</td>
<td>—</td>
</tr>
<tr>
<td>Overcommented Test</td>
<td>OCT</td>
<td>NO</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Overreferencing</td>
<td>OF</td>
<td>YES</td>
<td>YES</td>
<td>—</td>
</tr>
<tr>
<td>Proper Organization</td>
<td>PO</td>
<td>YES</td>
<td>NO</td>
<td>—</td>
</tr>
<tr>
<td>Redundant Assertion</td>
<td>RA</td>
<td>YES</td>
<td>NO</td>
<td>—</td>
</tr>
<tr>
<td>Redundant Print</td>
<td>RP</td>
<td>NO</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Resource Optimism</td>
<td>RO</td>
<td>NO</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Returning Assertion</td>
<td>ReA</td>
<td>NO</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Rotten Green Tests</td>
<td>RGT</td>
<td>YES</td>
<td>YES</td>
<td>—</td>
</tr>
<tr>
<td>Sensitive Equality</td>
<td>SE</td>
<td>YES</td>
<td>YES</td>
<td>—</td>
</tr>
<tr>
<td>Sleepy Test</td>
<td>ST</td>
<td>NO</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Slow tests</td>
<td>SSt</td>
<td>YES</td>
<td>NO</td>
<td>—</td>
</tr>
<tr>
<td>Teardown Only Test</td>
<td>TOT</td>
<td>NO</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Test Code Duplication</td>
<td>TCD</td>
<td>YES</td>
<td>NO</td>
<td>—</td>
</tr>
<tr>
<td>Test Logic in Production</td>
<td>TLP</td>
<td>NO</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Test Maverick</td>
<td>TM</td>
<td>NO</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Test Pollution</td>
<td>TP</td>
<td>NO</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Test Redundancy</td>
<td>TR</td>
<td>YES</td>
<td>NO</td>
<td>—</td>
</tr>
<tr>
<td>Test Run War</td>
<td>TRW</td>
<td>NO</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Test-Class Name</td>
<td>TCN</td>
<td>NO</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Unknown Test</td>
<td>UT</td>
<td>YES</td>
<td>NO</td>
<td>—</td>
</tr>
<tr>
<td>Unused Inputs</td>
<td>UI</td>
<td>YES</td>
<td>YES</td>
<td>—</td>
</tr>
<tr>
<td>Unusual Test Order</td>
<td>UTO</td>
<td>NO</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Vague Header Setup</td>
<td>VHS</td>
<td>NO</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Verbosity Test</td>
<td>VT</td>
<td>YES</td>
<td>YES</td>
<td>—</td>
</tr>
</tbody>
</table>
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” re-
quires 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 re-
mainig 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 ($\sigma$), and confidence intervals (CI) using bootstrapping at 95% significance level, report the distribution of test smell metrics.

<table>
<thead>
<tr>
<th>Metric</th>
<th>$\bar{x}$</th>
<th>$\sigma$</th>
<th>CI</th>
</tr>
</thead>
<tbody>
<tr>
<td>AssertionRoulette</td>
<td>2.66%</td>
<td>0.09</td>
<td>[0.02, 0.04]</td>
</tr>
<tr>
<td>DuplicateAssert</td>
<td>0.58%</td>
<td>0.04</td>
<td>[0.00, 0.01]</td>
</tr>
<tr>
<td>EagerTest</td>
<td>3.98%</td>
<td>0.12</td>
<td>[0.03, 0.05]</td>
</tr>
<tr>
<td>IndirectTesting</td>
<td>34.79%</td>
<td>0.27</td>
<td>[0.32, 0.38]</td>
</tr>
<tr>
<td>LackOfCohesionOfMethods</td>
<td>0.32%</td>
<td>0.06</td>
<td>[0.00, 0.01]</td>
</tr>
<tr>
<td>LazyTest</td>
<td>0.00%</td>
<td>0.00</td>
<td>[0.00, 0.00]</td>
</tr>
<tr>
<td>LikelyIneffectiveObjectComparison</td>
<td>0.01%</td>
<td>0.00</td>
<td>[0.00, 0.00]</td>
</tr>
<tr>
<td>ObscureInLineSetup</td>
<td>1.26%</td>
<td>0.05</td>
<td>[0.01, 0.02]</td>
</tr>
<tr>
<td>Overreferencing</td>
<td>5.12%</td>
<td>0.15</td>
<td>[0.03, 0.07]</td>
</tr>
<tr>
<td>RedundantAssertion</td>
<td>0.02%</td>
<td>0.00</td>
<td>[0.00, 0.00]</td>
</tr>
<tr>
<td>RottenGreenTests</td>
<td>0.81%</td>
<td>0.03</td>
<td>[0.00, 0.01]</td>
</tr>
<tr>
<td>TestRedundancy</td>
<td>0.00%</td>
<td>0.00</td>
<td>[0.00, 0.00]</td>
</tr>
<tr>
<td>UnknownTest</td>
<td>45.91%</td>
<td>0.27</td>
<td>[0.43, 0.49]</td>
</tr>
<tr>
<td>UnusedAssertions</td>
<td>16.19%</td>
<td>0.20</td>
<td>[0.14, 0.18]</td>
</tr>
<tr>
<td>UnusedInputs</td>
<td>25.85%</td>
<td>0.24</td>
<td>[0.23, 0.28]</td>
</tr>
<tr>
<td>VerboseTest</td>
<td>1.32%</td>
<td>0.05</td>
<td>[0.01, 0.02]</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td>8.68%</td>
<td>0.10</td>
<td>[0.08, 0.10]</td>
</tr>
</tbody>
</table>

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, oDynaMOSA (extension of the DynaMOSA), which optimizes the runtime and memory consumption of the tests as secondary criteria. According to their results, oDynaMOSA 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:

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

where $s_k(t)$ ∈ [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
Does the optimization of test smell metrics affect the code coverage?

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:

\[
\frac{100 \times \sum_{t \in T} 1}{|T|} \begin{cases} 1 & \text{if } t(s) \text{ is above or equal to threshold.} \\ 0 & \text{if } t(s) \text{ is below threshold.} \end{cases}
\]

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:

\[
\frac{\text{mean}(A) - \text{mean}(B)}{\text{mean}(B)}.
\]

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 crbanch 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 reducted for anonymity.

In RQ1 we investigate the impact of EvoSuite’s default secondary criteria (i.e., verbose test) and its 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 extent the optimization of test smell metrics if effective at generating less smelly tests.

\(^1\)It is not truly 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 Inline Setup, 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 (i.e., Eager Test, Indirect Testing, Likely Ineffective Object Comparison, Obscure Inline Setup, Overreferencing, Rotten Green Tests, and Verbose Test).

**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 Verbose Test (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%)

#### 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 (directly 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.
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 generated similar coverage. Column $\bar{x}$ reports the ratio of smelly tests generated by each configuration. $\bar{A}_{12}$ reports the effect size of $X$ vs. $Y$. Note that statistically significantly effect size values, i.e., $p$-value $\leq 0.05$, are annotated in bold. Column ‘Rel. impr.’ reports the relative improvement of $x$ over $Y$ regarding the percentage of smelly tests generated by both configurations.

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. Column $\bar{x}$, standard deviation ($\sigma$), and confidence intervals (CI) using bootstrapping at 95% significance level, report the distribution of test smell metrics.

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.

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, $\bar{A}_{12} = 0.55$ with a $p$-value 0.376, and $\bar{A}_{12} = 0.53$ with a $p$-value 0.391, respectively.

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.