Empirical Software Engineering manuscript No.
(will be inserted by the editor)

Test Smells 20 Years Later: Detectability, Validity, and Reliability

Annibale Panichella · Sebastiano Panichella · Gordon Fraser · Anand Ashok Sawant · Vincent J. Hellendoorn

Received: date / Accepted: date

Abstract Test smells aim to capture design issues in test code that reduces its maintainability. These have been extensively studied and generally found quite prevalent in both human-written and automatically generated test-cases. However, most evidence of prevalence is based on specific static detection rules. Although those are based on the original, conceptual definitions of the various test smells, recent empirical studies indicate that developers perceive warnings raised by detection tools as overly strict and non-representative of the maintainability and quality of test suites. This leads us to re-assess test smell detection tools' detection accuracy and investigate the prevalence and detectability of test smells more broadly. Specifically, we construct a hand-annotated dataset spanning hundreds of test suites both written by developers and generated by two test generation tools (EvoSuite and JTExpert) and performed a multi-

A. Panichella
Delft University of Technology
Delft, The Netherlands
E-mail: A.Panichella@tudelft.nl

S. Panichella
Zurich University of Applied Science
Zurich, Switzerland
E-mail: sebastiano.panichella@zhaw.ch

G. Fraser
University of Passau
Passau, Germany
E-mail: Gordon.Fraser@uni-passau.de

A. A. Sawant
University of California Davis
Davis CA, USA
E-mail: asawant@ucdavis.edu

V. J. Hellendoorn
Carnegie Mellon University
Pittsburgh PA, USA
E-mail: vhellendoorn@cmu.edu

stage, cross-validated manual analysis to identify the presence of six types of test smells in these. We then use this manual labeling to benchmark the performance and external validity 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 primarily show that the current vocabulary of test smells is highly mismatched to real concerns: multiple smells were ubiquitous on developer-written tests but virtually never correlated with semantic or maintainability flaws; machine-generated tests actually often scored better, but in reality, suffered from a host of problems not well-captured by current test smells. Current test smell detection strategies poorly characterized the issues in these automatically generated test suites; in particular, the older tool's detection strategies 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, (ii) more accurate detection strategies to be evaluated primarily in industrial contexts.

Keywords Test Generation · Test Smells · Software Quality

1 Introduction

A core goal of software engineering research is automatically assessing the quality and maintainability of software, including code used for testing. Doing so automatically is challenging (Deursen et al., 2001; Spadini et al., 2020), 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 (Bavota et al., 2012; Tsantalis et al., 2018). Ideally, developers can be alerted to such issues automatically, and they can resolve them through meaning-preserving refactorings. This is the goal of "test smell" detection (Deursen et al., 2001).

The canonical catalog of test smells was proposed in 2001 by van Deursen *et al.* (Deursen et al., 2001) and consists of 11 smell types with the associated recommended refactorings. These smells are well-defined and grounded in the definition of unit tests to reflect improper testing practice. Research since has created tools that automatically mark these smells in test code (Bavota et al., 2015b; Grano et al., 2019). This is no small chore: many of these require complex inference about the code; *e.g.*, detecting "indirect testing" requires knowing whether (unmocked) calls to other classes constitute "testing" those classes. Present tools, therefore, rely on heuristic, static, rule-based detection of test smells, which they tend to find ubiquitous.

As testing habits and tools continue to evolve, however, an over-reliance on both this vocabulary of test smells risks our field becoming increasingly divorced from testing practice. Indeed, a recent empirical study showed that developers do not perceive test smells as reflective of test suite quality and maintainability, finding them overly strict (Spadini et al., 2020). The same applies to heuristic detection rules, especially when it comes to *automatically generated* tests, which are often markedly different in structure and semantics. Recent work based on heuristic detection suggests that test smells are especially prevalent among such tests (Grano et al., 2019). Yet, considering that most of the heuristics used were crafted for manually-written test cases, some long before such tools proliferated, we may well expect that they are not directly transferable to automatically generated test suites.

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, with specific attention to automatically generated test cases.

In our previous conference paper published at ICSME, 2020 (Panichella et al., 2020b), we replicated and expanded over the empirical studies by Grano *et al.* (Grano et al., 2019) and Palomba *et al.* (Palomba et al., 2016). Both studies investigated the distribution of test smells in automatically generated tests and used warnings raised by static detection tools as the *ground truth*. Our study focused on test cases generated by EvoSuite and two different test smell detection tools – the one used in those works and tsDetect, a more recent tool tailored for manually written tests. Our results were quite different from what was reported in the aforementioned studies: we showed that (1) *test smell detection tools are highly inaccurate* for automatically generated test cases, and (2) test smells are present in generated tests but *in a much smaller portion than previously reported*; rather (3) these tests are plagued by many other, obvious concerns not captured by test smells. This discrepancy in findings was due to the many false positives and false negatives produced by test smell detection tools compared to manual inspection. This raises critical concerns about the validity of empirical studies that rely on such tools without carefully and manually validating the raised warnings.

In this work, we extend our conference paper in several ways. We build a substantially larger dataset of manually annotated test smells by expanding our annotations to tests generated by two tools and actual developers, leading to annotations for hundreds of test suites spanning thousands of test cases. We provide the following contributions:

Internal Validity: *How widespread are test smells in automatically generated test cases?* We consider two different test-case generation tools, namely EvoSuite (Fraser and Arcuri, 2011) and JTExpert (Sakti et al., 2014, 2017). We find that both tools effectively abstract away many resources (*e.g.*, database and file access) and cannot produce (by design) test smells related to external resources and files as they use mocks and customized runners. Besides, the post-processing test optimization in EvoSuite dramatically minimizes the portion of test cases affected by test smells, which is much smaller than reported in prior studies. Nevertheless, test generation tools still introduce a substantial number of indirect tests and often generate tests that check a wide range of behaviors simultaneously. However, these issues are poorly detected by smell detection tools. We suggest that a notion of semantic objective is needed to refine its focus.

How widespread are test smells in manually written test cases? We compare the distribution of test smells across manually written test cases with their automatically generated counterparts. Our results indicate that manually written tests are affected by more test smells. However, this effect is almost entirely artificial: current definitions of test smells are far too strict and fail to capture design flaws in test code. Even though the test smell detection tools capture many smelly tests, these instances are unlikely to affect future test maintenance activities negatively.

Tool Validity: *How accurate are automated tools in detecting test smells for both manually-written and 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 most of these issues in their own manual validation, raising concerns about self-evaluating the validity of heuristics –a common practice in our field – as opposed to industry-based evaluations.

External Validity: *How well do test smells reflect real problems in 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"). These were ubiquitous in real, developer-written tests, even in mature, well-engineered projects, but virtually never correlated with *semantic coherence* or, anecdotally, readability. In turn, we highlight several issues not yet described by any smells, particularly ones that reflect atypical patterns found in the tests produced automatically.

2 Background

This section summarizes the background notions and the main related work related to test smells, test-case generation, and the limitations of prior work.

2.1 Test Smells

The idea of code smells dates back to Fowler's book (Fowler, 1999). Code that smells is not necessarily faulty but may contain quality issues that inhibit maintenance or lead to introducing bugs in later development stages (Tufano et al., 2017). The notion of code smells was later extended to test code by van Deursen *et al.* (Deursen et al., 2001). To help developers pinpoint and address such issues, several tools have been introduced to automatically flag smelly test code (Bavota et al., 2012; Spadini et al., 2020). 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 (Bavota et al., 2015b; Peruma, 2018; Spadini et al., 2020).

2.2 Test Case Generation

Writing test code can be tedious and methodical. To alleviate this burden, a longstanding goal of researchers is to generate 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 (Csallner and Smaragdakis, 2004) or violations of general object contracts (Pacheco et al., 2007), but the generated tests can also be used as automated regression tests. The effectiveness of random test generators can be increased by integrating heuristics (Ma et al., 2015; Sakti et al., 2014). In search-based testing, evolutionary search algorithms are popular, which gradually improve random initial sequences of calls to maximize code coverage (Andrews et al., 2011; Baresi and Miraz, 2010; Fraser and Arcuri, 2012; Tonella, 2004).

Research studies have shown that test case generation techniques are effective at achieving high code coverage (Campos et al. (2018); Panichella et al. (2018b)) and fault detection (Almasi et al. (2017); Fraser and Arcuri (2015a)). The generate tests can also help developers reducing the time to complete debugging tasks (Panichella et al. (2016); Soltani et al. (2018)) and are useful in regression testing (Birchler et al. (2022a), Birchler et al. (2022b)).

However, automatically generated tests do tend to be harder to read and interpret than manually crafted ones, which negatively impacts their maintainability (Shamshiri et al., 2018). Research has focused on improving this maintainability primarily by addressing the amount of code generated: RANDOOP (Pacheco et al., 2007) removes redundant tests after-the-fact, while EVOSUITE (Fraser and Arcuri, 2012) 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 (i.e., method calls and assertions) that cannot be removed without reducing the achieved coverage and mutation score. JTEXPERT (Sakti et al., 2017) generates bounded sequences of method calls based on the number of attributes in the class under test. This strategy helps prevent the *bloating* effect, i.e., creating test suites that grow increasingly large in every iteration.

Various other post-processing steps are common for automated test generators; for example, most tools integrate some form of regression oracle generation (Xie, 2006) 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 (Fraser and Zeller, 2011). Further optimizations for maintainability include applying specific heuristics from industrial application (Robinson et al., 2011), semantic simplification (Zhang, 2013), prioritizing using literals taken from the source code (Rojas et al., 2016), generation of meaningful test names tests (Daka et al., 2017) and variable names (Roy et al., 2020), measuring readability using prediction models (Daka et al., 2015), and adding textual summaries (Panichella et al., 2016; Roy et al., 2020), to support newcomers understanding and testing software systems (Panichella, 2015).

2.3 Limitations of Prior Work

Grano *et al.* (Grano et al., 2019) and Palomba *et al.* (Palomba et al., 2016) investigated test smells in automatically generated test cases and found them to be widespread. Specifically, the vast majority (81%) of test suites generated by EvoSuite and almost all test suites generated by JTExpert (92%) contained at least one test smell. That rate is remarkably high, especially considering that in the case of EvoSuite the tool applies post-search test suite optimization to reduce the cost of the oracle problem and the chance of generating failing and flaky tests (Panichella et al., 2020a). This is especially surprising given how widely EvoSuite is used, which raises the question of whether those smells are indeed indicative of quality and maintainability issues. A careful manual analysis of such tests and their purported smells thus seems in order, especially in contrast to developer-written tests. We focus on five systematic concerns with the established numbers:

1. Grano *et al.* and Palomba *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 (Bavota et al., 2015a; Grano et al., 2019). Hence, the extent of the reported test smell instances depends on the large number of warnings raised by detection tools. This was not manually validated, so it is not clear if this truly indicates fundamental design flaws in test case generation tools. In this paper, we address this issue by building a *gold standard* with manually annotated classes, to gain a more clear view of test smells occurrences in both automatically generated and manually written tests.

2. The authors used test case generation tools with default parameter values, ignoring more recent studies on generating shorter and more maintainable test cases (e.g., (Daka et al., 2017; Panichella et al., 2017)). Indeed, tools like EvoSuite are equipped with tunable 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 by Grano *et al.* (Grano et al., 2019) and Palomba *et al.* (Palomba et al., 2016). 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). Modern tools also use execution sandboxes and disable calls to the file system. However, this aspect was not considered nor discussed by Grano *et al.* and Palomba *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.

4. The prior studies by Grano *et al.* and Palomba *et al.* did not compare the distribution of test smells between automatically generated tests and their manually-created counterparts. This comparison is necessary to better understand whether test case generation tools are *flawed by design* (as claimed by Grano *et al.* (Grano et al., 2019)) or if they can generate tests that are as maintainable as a developer's, at least from a test smell perspective.

5. Even if a test suite contains test smell instances, this alone does not imply design flaws or maintainability concerns. In this paper, we question the implicit equation "*test smell ≡ design flaw*" that is implicitly used in prior studies (Grano et al., 2019; Palomba et al., 2016). In fact, by definition, test smells are *symptoms that could indicate potential design problems in test code* (Bavota et al., 2015b; Deursen et al., 2001). Hence, a manual analysis (or an assessment involving developers) of the test smell instances is needed to verify whether they capture real design flaws in test code.

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 does 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 (Fraser and Arcuri, 2014; Panichella et al., 2017), thus, altering the test structure. If the majority of the generated tests are indeed smelly, the root cause is to be found elsewhere.

3 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 at the link:

`https://figshare.com/s/7b8bf9a7580001929f63`.

3.1 Research Questions

The following research questions guide 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 automatically generated test suites?*
- **RQ4**: *How does test smell diffusion in manually written tests compare to automatically generated tests?*
- **RQ5**: *How well do test smells capture real problems in manually written tests?*

We first determine the spread and distribution of test smells in automatically generated tests based on manual analysis. Previous work (Grano et al., 2019) 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 (Grano et al., 2019). In this study, we answer **RQ1** by building a curated dataset of automatically generated test cases (further described in Section 3.3) and manually identifying the presence of six types of test smells (see Section 3.5). Our approach addresses an important *threat to construct validity* of the 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. **RQ3** reflects on these test smells, asking whether and how often they relate to actual problems in the test suites, which is illustrated through a series of examples.

Both the previous study by Grano *et al.* and this study (**RQ1**-**RQ3**) so far focused on quantifying issues in automatically generated tests, to what extent they are hard to maintain. The quantitative study is done by means of manual validation (our study) and running test smell detection tools (Grano et al., 2019). However, test smell detection tools have been designed and assessed for manually-written tests. Hence, one could argue that their accuracy might be different when applied to test cases produced in different ways. To the best of our knowledge, no prior study investigated whether test cases written by developers and their automatically generated counterparts (for the same Java classes) manifest the same maintainability issues and are affected by the same types of test smells. Finally, it is worth re-assess the accuracy of test smell detection tools considering both types of test cases.

Hence, we formulated two additional research questions with regards to manually written tests as well. **RQ4** first asks how the diffusion of smells compares to that in the automatically generated test suites. **RQ5** then investigates the natural question of whether test smell definitions appropriately capture issues in these, developer-written tests, for which they were originally designed. In other words, we investigate the extent to which the test smell catalog is up to date with the current testing framework and practices.

3.2 Test class selection

To build a set of test suites for manual analysis, we consider the same 100 Java classes used by Grano *et al.* (Grano et al., 2019). These classes are extracted from the SF110 dataset (Fraser and Arcuri, 2014), which contains 110 projects from `SourceForge.net`. The selected classes are non-trivial as identified using a well-established *triviality test* (Panichella et al., 2017), 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. As a consequence of this selection criteria, there are no precisely equivalent hand-written test suites for the classes that were selected in the original study (Grano et al., 2019), for us to use in answering **RQ4** and **RQ5**. In particular, only eight out of 100 classes selected by Grano *et al.* have a manually-written test suite. Hence, we extended the benchmark used to answer **RQ4** and **RQ5** by selecting other 41 test suites from the top 10 most popular Java projects (Fraser and Arcuri, 2014) in the SF110 dataset and that meet similar complexity criteria. We manually validated the suites in this extended benchmark to validate the (eventual) test smells.

3.3 Test Case Generation

Once we selected the classes for which to generate test suites, we next generated JUnit test cases for the selected classes using two tools: EvoSuite (Fraser and Arcuri, 2011) version 1.0.7 and JTExpert (Sakti et al., 2014, 2017). EvoSuite is a state-of-the-art unit-test generator for Java programs that has won several editions of the SBST tool competition (Devroey et al., 2020; Kifetew et al., 2019; Panichella and Molina, 2017), including the edition of 2021 (Panichella et al., 2021), against competing random and search-based unit test generation tools; it produces test suites with high code coverage (Fraser and Arcuri, 2014), has documented fault detection capability (Almasi et al., 2017; Fraser and Arcuri, 2015a), and is publicly available on `GitHub`.[1] EvoSuite also applies several post-processing optimizations to reduce the size of the test cases, identify and remove flaky tests, as well as minimizing the number of generated assertions (see Section 2.2). JTExpert is a well-known testing framework for Java programs that was ranked second in the 2017 SBST tool competition (Sakti et al., 2017). JTExpert utilizes a random strategy with static analyses to automatically generate a complete test suite based on a branch coverage criterion, and is publicly available online.[2,3] Differently from EvoSuite, JTExpert does not apply any post-processing optimizations. A limitation of JTExpert is that it does not always generate test cases. For instance, in the SBST tool competition of 2017, it did not generate test cases for 415 out of 1450 runs (27%).

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

[2] https://sites.google.com/site/saktiabdel/JTExpert

[3] https://github.com/dldbb/Automate-Testing-Tools-Test

We decided to extend our conference work (Panichella et al., 2020b) by considering two different test case generation tools, as they differ on many technical details, including (1) encoding schema for test cases/suites, (2) the search algorithm, (3) the test criteria they optimize for, (4) the mocking mechanisms, and (5) post-process test optimization. We detail the core differences in the following subsections.

3.3.1 EvoSuite

The default search algorithm in EvoSuite is DynaMOSA (Panichella et al., 2017, 2018a), a many-objective genetic algorithm that iteratively evolves test cases. In DynaMOSA, a test case t is encoded as a sequence of statements, whose length is variable; t is evaluated against all uncovered coverage targets (e.g., branches). There are six types of statements that can compose a test case: (i) primitive statement, (ii) constructor statement, (iii) field statement, (iv) method statement, (v) assignment statement, and (vi) mock statement.

Since DynaMOSA works at the test-case level, the genetic operators (i.e., mutation and crossover) are also defined and implemented at that granularity level. In particular, the single-point crossover recombines pairs of test cases (called *parents*) by exchanging statements between the two parent test cases. The *uniform* mutation randomly deletes, changes, or adds statements in a given test case. Each generated test case is evaluated only w.r.t the yet uncovered coverage targets (e.g., branches). Generated tests that reach uncovered targets are added to an *archive*, which keeps track of the shortest test case satisfying each target. The final test suite is obtained by post-processing the test cases stored in the archive.

EvoSuite optimizes multiple test adequacy criteria (or coverage criteria) simultaneously using multi-objective optimization algorithms (Canfora et al., 2013, 2015; Panichella et al., 2018a; Rojas et al., 2015). The tool includes the following criteria: (1) branch coverage, (2) line coverage, (3) weak mutation, (4) input coverage, (5) output coverage, (6) exception coverage, and (7) method coverage. Branch, line, and method coverage are traditional white-box criteria that measure how many code elements are covered by the generated test cases (Ammann and Offutt, 2016). Indeed, these criteria count the number of coverage targets (either branches, lines, or methods) that are covered. Method coverage measures the number of methods directly or indirectly invoked by the tests. Weak mutation counts the number of mutants (artificial faults) that a test case *weakly* kills. A test case t weakly kills a mutant if the internal execution state differs when executing t against the mutant compared to the original program (Just et al., 2014). Input and output coverage are *black-box* criteria and measure the diversity in the input and output space of the program under test (Ammann and Offutt, 2016). Finally, exception coverage measures the number of exceptions triggered when executing the generated test cases. This criterion allows storing test cases that likely lead to crashes into the final test suite.

Another important feature in EvoSuite is the post-process optimization. Once the search process ends, the produced test suite (collected in the archive) undergoes a sequence of optimization steps aiming to minimize the oracle cost and address potential flaky behaviors. The optimization steps are:

– *Test minimization*: the test cases are first minimized by removing spurious statements that do not contribute to any coverage criteria. In fact, crossover and mutation may add statements to a test case that do not lead to covering any additional coverage targets (e.g., branches). Removing these unnecessary statements can potentially reduce the oracle cost (i.e., the time for manually inspecting the test) and address test smells.
– *Assertion minimization*: the generated test cases are first enriched with assertions, that assert the output values returned by method calls, get methods, as well as the values of public and protected attributes. The assertions are then filtered based on their ability to strongly kill mutants. A mutant is strongly killed by a test case t if t passes on the original program but fails when executed against the mutant. Assertions that do not contribute to killing new mutants are removed during this phase.
– *Flakiness detection and removal*: the final test suite is re-executed to detect potential flaky tests, i.e., tests with non-deterministic behaviors. Flaky test cases are removed in this step.

These post-process steps have their timeouts. Hence, carefully tuning these timeouts is critical to guarantee enough time for these steps to produce concise and effective test suites. Proper post-processing is critical in our context as one would argue that, at the same coverage, larger tests are less maintainable than smaller ones (Fraser and Arcuri, 2014; Panichella et al., 2017).

3.3.2 JTExpert

JTExpert generates whole-suites that maximize branch coverage criterion for Java classes (Sakti et al., 2014). While EvoSuite relies on evolutionary algorithms (and DynaMOSA in particular) to generate test cases, JTExpert uses a specialized random search that targets every uncovered branch at the same time (Sakti et al., 2014). Test cases are randomly generated using a Source Code Analyzer (SCA) and the Test Case Candidates Builder (TDCB). SCA statically analyzes the source code of the Java class under test and identifies two types of methods: (1) methods that are likely to change the internal states of the class, and (2) methods that can lead to reaching a given branch. The data collected by the analyzer are used by TDCB to generate test cases. A test case is encoded as a vector of means-of-instantiation of the target class (i.e., a constructor, a method factory) and a sequence of method calls (Sakti et al., 2017). The length of the method sequence is bounded by the number of declared data members in the target Java class.

When generating new test cases, JTExpert focuses only on the uncovered branches. This means that newly generated tests will invoke only methods that, according to TDCB, can likely reach an uncovered branch or can change the state of the objects for the class under test. Hence, this strategy prevents generating test cases with "arid" method calls targeting already covered or infeasible branches. To further speed up the search, input data (e.g., input parameters) are generated using both dynamic and stating seeding. The SCA collects all constants for each primitive data type (e.g., `int`) and strings that appear in the source code and store them in a seeding pool. Then, with a given probability, input data are generated at random or selected from the seeding pool. Besides, JTExpert also seeds the null value with a constant probability while generating instances of classes (Sakti et al., 2017).

Another critical difference between EvoSuite and JTExpert is that the latter does not post-process the final test suite. Therefore, the test cases and the assertions are not minimized, which might lead to longer test cases. Finally, JTExpert does not use/generate mocks.

3.3.3 Parameter Setting

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*" (Grano et al., 2019). However, there are several problems with this assumption: first, the default values are optimized for code coverage (Arcuri and Fraser, 2013), but these parameters affect other test suite properties of practical relevance, such as length or number of assertions. Second, the claim only holds for hyper-parameters of the meta-heuristic 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 as well as on the size of the generated tests (Campos et al., 2018; Panichella et al., 2017, 2018b; Rojas et al., 2017). Finally, the search budget has a substantial impact on the overall performance of both EvoSuite and JTExpert, as shown in the SBST tool competitions, e.g., (Devroey et al., 2020). Running these tools with a longer search budget will lead to higher/better coverage but potentially larger test suites. As shown by a recent study (Böhme et al., 2022), there is a strong correlation between coverage and fault detection capability of the generated test suites; hence, larger coverage is preferred.

Parameter Settings for JTExpert For JTExpert, we use the same settings used by Grano *et al.* (Grano et al., 2019). Specifically, we used the tool with its default configuration parameters to ensure a fair comparison; we did not tune its parameters, in contrast to EvoSuite, since it does not include post-processing optimizations after the generation.

Parameter Settings for EvoSuite The default setting in 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 the replication package by Grano *et al.*[4] If the test case minimization and the assertion generation reach their respective timeouts, EvoSuite terminates the post-processing and returns the original, *non-minimized*, test suites. To ensure that this phase completes successfully, we increase the timeouts from 60 seconds —used in (Grano et al., 2019)— to ten minutes per test suite (or equivalently per class under test). Note that 10 minutes is the upper bound to the time given for the post-process and, for small/medium classes, the actual post-process time is below 2 minutes.

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 (Campos et al., 2018). 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* (Panichella et al., 2018a; Rojas et al., 2015). We select DynaMOSA over the *whole-suite approach* as the former can achieve higher coverage with shorter tests.

Coverage Results. Table 1 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 (Grano et al., 2019). The larger coverage is due to both using DynaMOSA as the underlying search algorithm and extending the search budget to 10 minutes. On average, the generated test suites have more test cases with our setting (see the number of test cases in Table 1) but the corresponding test cases are shorter (see the total test length in Table 1). This result is due to the extended time given to the post-process optimization.

3.4 Detection Tool Selection

We select two test smell detection tools. The first tool, proposed by Bavota *et al.* (Bavota et al., 2015a), uses hand-crafted, static detection rules; they report that it achieves 100% recall and 88% precision when detecting test smells in test cases written by developers. It has since been widely used in previous work (Bavota et al., 2015a; Spadini et al., 2018; Tufano et al., 2016) to study the distribution of test smells in manually-written tests from open-source projects. Grano *et al.* (Grano et al., 2019) use this same tool to detect

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

Table 1: Comparison between the test suites generated with our setting vs. Grano *et al.* (Grano et al., 2019) setting. We report the median values, the interquartile range (IQR), and confidence intervals (CI) using bootstrapping at 95% significance level.

	Settings by Grano *et al.* (Grano et al., 2019)			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]

smells in automatically generated test suites. On those, they report an average precision of 75%, with 100% precision on four test smell types, namely *Assertion Roulette*, *Mystery Guest*, *Sensitive Equality*, and *For Tester Only*.

The second detection tool we study is TSDETECT (Peruma, 2018), which is publicly available on `GitHub`.[5] Recently, Spadini *et al.* (Spadini et al., 2020) 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.* (Grano et al., 2019) detects just seven; in our analysis, we focus on these seven, five of which overlap between 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 as it does not apply to automatically generated test suites, which, 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 2.

3.5 Manual validation

To create a golden set of test smells in automatically generated test suites, we manually evaluated a large number of test suites – 100 each for EVOSUITE and JTEXPERT, and 49 manually written test suites. For each test suite, we analyze whether any of the test cases in the test suite suffers from one of the six smells specified in Table 2. Out of these six smells, we do not explicitly annotate *Mystery Guest* and *Resource Optimism* for the automatically generated tests, as those tools cannot generate tests that suffer from these issues by construction, i.e., by using mocks, dedicated runners, or disabling file manipulation rights. Instead, we manually annotate the remaining four types of

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

Table 2: Test smells considered in this paper.

Test smell	**Definition by van Deursen *et al.* (Deursen et al., 2001)**	**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 for automatically generated tests, since tools like EvoSuite use runners that 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 a 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 for automatically generated tests, since tools like EvoSuite use runners that by definition mocks out all accesses to external resources.

test smells since EvoSuite and JTExpert do not have mechanisms that prevent them by constructions. We conduct this manual analysis in a multi-stage cross-validated manner:

Step 1. Each test suite was independently inspected by two authors of this paper. For each test suite, the analysis is done across the dimensions corresponding to the selected test smells (5 for generated tests, 7 for manually written ones). 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 2. We note that the EvoSuite analysis involved four authors annotating 50 suites each; this annotation was part of our original conference paper (Panichella et al., 2020b) and served to calibrate our annotation protocol. The other two sets were annotated in full by two authors.

Step 2. For each set of test suites, the two authors responsible for the analysis discuss their findings and any disputed cases to come to a resolution. We generally encountered disagreement levels between 10% and 20%, which could mostly be resolved through discussion with reference to the guidelines.

Step 3. Any remaining controversial cases that could not be resolved between two annotators were discussed by all authors to come to a final agreement and improvement in the protocol. This discussion involved ten cases in total (of which seven on the first dataset, from EvoSuite), and led to slight

Table 3: Distribution statistics of the selected test smells in 100 test suites spanning all collected systems.

Smell	Manually Validated		Reported by (Grano et al., 2019)	
	EvoSuite	JTExpert	EvoSuite	JTExpert
Eager Test	21%	61%	57%	62%
Assertion Roulette	17%	64%	74%	74%
Indirect Testing	32%	47%	-	-
Sensitive Equality	19%	53%	7%	66%
Mystery Guest*	0%	0%	11%	15%
Resource Optimism*	0%	0%	3%	-

* EvoSuite and JTExpert never generate tests requiring external resources.

refinements in the guidelines for corner-cases, as test smells manifest in many complex ways. Furthermore, during this phase, test cases that are not smelly but still demonstrate interesting anomalies were also discussed. At the end of this phase, the classification of all test suites is set.

4 Empirical Results

This section discusses the results of our empirical study, addressing in turn each research question.

4.1 RQ1: How widespread are test smells in automatically generated test cases?

To evaluate the accuracy of automated test smell annotation tools (RQ2) and to study the gaps in test smell coverage (RQ3), we need to understand whether and the extent to which generated tests are characterized by the presently used test smells. This necessarily requires manual annotation, which we conducted as described in Section 3.5. To be consistent with prior work (Grano et al., 2019), 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 was annotated accordingly. It is worth noting that this leads to a considerable overestimation of the incidence of test smells at a test case level due to coarse-grain analysis.

Table 3 shows the resulting general incidence rate of each smell across all test suites, grouped by their sources. Overall, test smells are commonly present in a non-trivial portion of automatically generated test suites, with JTExpert incurring significantly more than EvoSuite. The respective incidence rates are all quite similar between different smells, but their distributions vary. For both test generation tools, the percentage of test suites affected by test smells is remarkably different from what has been reported in prior studies that did not manually validate the warnings raised from test smell detection tools.

For EvoSuite, about half the test suites contained no smells at all; another five contained every possible smell; the remainder involved certain especially frequent pairings. This is a remarkable difference with the incidence of 81% reported by Grano *et al.* for this test case generation tool. Eager tests and assertion roulette often co-occurred (appearing together in 12 out of their respective 20 and 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. While eager tests and assertion roulette are present in a non-negligible portion of the generated suites, these percentages (obtained after manually validating the test cases) are three-fold smaller than previously reported. Remarkable are also the differences for other test smells. 32% of the generated suite contains at least one test method with indirect testing, while the original studies never reported any indirect test. Finally, our manual validation indicates that EvoSuite and JTExpert never produce test suites with resource optimism or mystery guest instances, which is in contrast with the percentages reported in prior studies for the same set of classes under test. These differences in the results are due to limitations of test smell detection tools, as we will elaborate in Section 4.2.

We noticed that test methods in a test suite were often very similar to each other, with typically just a few archetypes repeated with a 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.

The incidence rate of the test smells is much larger for the test suites generated with JTExpert compared to the ones generated with EvoSuite. In particular, 61% of the suites generated by JTExpert contain at least one eager test method. This percentage is very close to 61% reported in prior studies (Grano et al., 2019; Palomba et al., 2016). Assertion roulette is three times more frequent among the test suites generated by JTExpert than those produced with EvoSuite. Despite being quite common, assertion roulette is remarkably less frequent when manually validating the test cases compared to the incidence rate reported by test smell detection tools. For all other types of test smells, our results are very different from what was reported by Grano *et al.* (Grano et al., 2019). Indirect testing was never reported in the original study; however, our manual validation revealed that almost 50% of the suites contain indirect tests. This further highlights existing detection tools' inability to correctly identify indirect testing.

Very remarkable are also the results for mystery guests. Grano *et al.* (Grano et al., 2019) reported that 15% of the suites generated by JTExpert use external resources such as files and databases. However, after manually validating and analyzing the test suites, we found that this test smell cannot occur with JTExpert. JTExpert does not create files and databases for security reasons. Doing so to improve code coverage is neither allowed nor recommended as

Table 4: 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.

Test smell	Tool used by (Grano et al., 2019)					tsDetect calibrated by (Spadini et al., 2020)				
	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	—	—	—	—	—	—

this can compromise the hosting machine used to run the test-case generation tools, e.g., by deleting files systems or saturating the hard drive.

Are automatically generated tests "scented since the beginning" as suggested by Grano et al. *(Grano et al., 2019)?.* Our manual analysis indicates that it is not the case. Both EvoSuite and JTExpert create a non-negligible ratio of test suites with at least one test smell instance. However, the incidence rate is much lower than previously reported. There is a very large difference between the test suite produced by EvoSuite compared to those by JTExpert. This indicates that not all tools are the same, and certain implementation and algorithmic choices matter when looking at the maintainability of the generated test suite.

In particular, we observe that EvoSuite produces test suites with much fewer test smells than JTExpert. This lower incidence of test smell is due to the different choices w.r.t. the search algorithms used by the two tools. EvoSuite uses DynaMOSA (Panichella et al., 2017), which minimizes the length of the test cases as a secondary objective in addition to the main objectives, i.e., the distances to the test coverage targets (e.g., branches and lines). Instead, JTExpert uses the whole-suite approach, which does not optimize for test length (Panichella et al., 2017). Another critical factor is the post-process test suite optimization implemented in EvoSuite, as already explained in Section 3.3. EvoSuite minimizes the test suites by removing redundant test cases; it reduces the size of the test cases by removing spurious statements that do not contribute to the final code coverage, and it also filters out assertions that do not contribute to killing mutants in (strong) mutation testing. These test suite optimization heuristics are widely known in the literature for controlling the size of the test suites (Yoo and Harman, 2012), the number of assertions (Fraser and Arcuri, 2015b), and improving fault localization (Xuan and Monperrus, 2014).

Finding 1. Test smells are commonly present in a non-trivial portion of automatically generated test suites. However, they occur substantially less often than previously reported. The incidence rate of the test smells also differs remarkably between test case generating tools; the use of test suite optimization heuristics plays a major role.

Table 5: Detection performance of different automated test smell detection tools for test cases generated by JTExpert. FPR denotes the False Positive Rate and FNR is the False Negative Rate.

Test smell	Tool used by (Grano et al., 2019)					tsDetect calibrated by (Spadini et al., 2020)				
	FPR	FNR	Prec.	Rec.	F-measure	FPR	FNR	Prec.	Rec.	F-measure
Assertion Roulette	0.03	0.03	0.98	0.97	0.97	0.06	0.11	0.96	0.89	0.92
Eager Test	0.00	0.34	1.00	0.66	0.79	0.05	0.38	0.95	0.62	0.75
Mystery Guest	0.11	—	—	—	—	0.01	—	—	—	—
Sensitive Equality	0.00	0.00	1.00	1.00	1.00	0.04	0.04	0.96	0.96	0.96
Resource Optimism	0.01	—	—	—	—	0.01	—	—	—	—
Indirect Testing	0.00	1.00	—	0.00	—	—	—	—	—	—

```
@Test(timeout = 4000)
public void test5() throws Throwable {
    ActionRegistry a0 = new ActionRegistry();
    ErrorPage errorPage0 = new ErrorPage();
    Label label0 = new Label(errorPage0, errorPage0);
    a0.addEntry("+}0IdF3!uYBcSb=", "+}0IdF3!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.

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

Table 4 and Table 5 report the false-positive rates (FPR), false-negative rates (FNR), precision, recall, and F-measure that each test smell tool achieves. 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 (Grano et al., 2019). 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 among 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.

Tables 4 and 5 suggest that tsDetect works fairly well on this test smell, reaching a higher precision: 67% for EvoSuite and 96% for JTExpert. For the test suites generated by EvoSuite, tsDetect achieves a lower recall compared to the tool used by Grano *et al.*, giving it a 21% higher F-measure. This is mainly due to the higher threshold value (three assertions) used by Spadini *et al.*. However, this simple heuristic 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.* (Spadini et al., 2018), "*the test method name accurately reflects the reason for the test to fail*" even when no further comments are provided in the tests.

For JTExpert, both detection tools achieve a very high F-measure (0.97 for the tool used by Grano *et al.* and 0.92 for tsDetect) with very low false positive and false negative rates. These results are mostly due to how JTExpert adds assertions (compared to EvoSuite): it adds as many assertions as possible (using getters and return values) without any post-process minimization (such as is implemented in EvoSuite). We manually analyzed the tests generated by JTExpert and found that these tests often either have zero assertions (if the tool could not find public getters to use) or many (using all possible getters). Test cases with one single assertion are rare for JTExpert, which are the most common instances of false positives for the tool used by Grano *et al.*.

Eager Test. The tool used by Grano *et al.* achieves high recall (95%) but low precision for EvoSuite. 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. For the test suites generated with JTExpert, the detection tool achieves a precision of 100%; however, the recall is much lower (66%) compared to the value achieved for EvoSuite (95%). Therefore, the performance of this tool varies depending on the test generation tool under analysis.

This test-smell detection tool uses the number of method calls (not including constructors) in a test method to determine whether it is eager or not. However, eagerness is properly concerned with *functionality* – whether more than one requirement is tested — and not with the number of invoked methods. As an example of false positive (e.g., test method incorrectly classified as eager), let us consider the test depicted in Figure 2 and that invokes two methods on the object `s1`. 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 (Spadini et al., 2018) for the same metric (the number of method invocations). This also causes it to again miss some instances, resulting in a

```
@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

lower recall (55%). For example, tsDetect correctly annotates the test in Figure 2 as non-smelly, but misses the case in Figure 4. The resulting F-measure is again higher for tsDetect, reinforcing that research with developers and human participants is critical to calibrating test smell detection tools properly.

Once again, both test smell detection tools achieve higher accuracy in detecting Eager Tests in the suites generated by JTExpert. The F-measure is quite similar, with 0.79 for the tool used by Grano *et al.* and 0.75 for tsDetect. The former detection tool has both a better precision and a better recall than the latter. It is worth noting that the better accuracy of the test smell detection tools for JTExpert compared to EvoSuite is because JTExpert generates large test cases, which are not minimized during post-processing. Hence, JTExpert rarely generates test cases on the edge of thresholds used by the test smell detection tools. In other words, the (non-minimized) tests generated by JTExpert are trivially detectable due to their large sizes.

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 (Arcuri et al., 2014). 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 constructors) mocked (Arcuri et al., 2014). EvoSuite also replaces general calls to the Java Virtual Machine (*e.g.*, `System.currentTimeMillis`) with mock classes/methods with deterministic behaviors. 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.

Unlike EvoSuite, JTExpert does not use mocks or customized runners. Instead, it simply does not create databases or external resources for security reasons. Creating random databases is not recommended as it risks saturating the hard disk, removing operating system files, or dropping existing databases. For file systems, JTExpert could instead create temporary files as it does not use mocks; however, the README file of the original repository[6] explicitly recommends not to use JTExpert for classes that may manipulate files systems. In our case, we run the JTExpert without file manipulation permissions. Therefore, test cases that include statements to create and manipulate files (e.g., `File f = new File();`) were not executed, triggering run-time errors. Even if a few test cases created with JTExpert statically include file manipulation statements, they must be considered false positives since no file was created dynamically at run-time. This scenario further highlights the limitations of static-based test smell detection tools.

Indirect testing. 32% of the test suites by EvoSuite and 47% of test suites by JTExpert contain test cases affected by indirect testing, according to our *gold standard* (see Table 3). This makes it the most widespread smell in automatically generated tests. However, the tool used in prior work (Grano et al., 2019) 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 for EvoSuite. 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

[6] `https://github.com/dldbb/Automate-Testing-Tools-Test/blob/master/README.md`

```
@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.

using first storing the result of `toString` in a local variable and then asserting its value against the target – a common pattern with EvoSuite.

On the other hand, both test smells detection tools achieve a very high F-measure for the test suites generated by JTExpert. This is because, unlike EvoSuite, JTExpert does not use local variables to store the results of `toString`. When the `toString` method is directly used inside the assertions, test smell detection tools correctly identify this type of smell.

Our results for sensitive equality raise critical concerns about the validity and effectiveness of using string matching to detect this type of test smells. First, small code tricks (e.g., using local variables) can easily fool test smell detection tools. Second, developers may use slightly different names (e.g., `toText()`, `prettyPrint()`) for methods that prints objects as `String` instances or implements string-based equality checks.

Finding 2. Test smell detection tools overestimate the occurrence of test smells, especially for test suites generated with EvoSuite, sometimes by large margins. Accurate automated detection of indirect testing, mystery guest, resource optimism, and sensitive equality remains an open challenge. Involving human participants is likely critical for improving the accuracy of test smell detection tools.

4.3 RQ3: How well do test smells reflect real problems in automatically generated test suites?

The goal of test smells is to reflect real, rectifiable issues in test cases. It is thus important to ascertain that detected smells are 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.

```
@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.

Eager test. We avoided mislabeling tests as eager when they check 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 (21% frequency as reported in Table 3), and 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, 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 rele-

```
@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 `Team-FinderImpl`.

Fig. 6: Examples of the indirect testing smell.

vant 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` (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 (19% frequency for EvoSuite as reported in Table 3)

```
@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.

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 (Arcuri et al., 2014). 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 increasing 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 bytecode instrumentation techniques used in EvoSuite effectively mitigate the concerns of mystery guests and resource optimism.

4.4 RQ4: How does test smell diffusion in manually written tests compare to automatically generated tests?

The previous results quantify the incidence of various test smells in automatically generated tests and additionally suggest that many of these poorly encapsulate problems in those. It is natural to ask, then, how these results relate to developer-written tests, both in terms of distributional characteristics and in severity and detectability.

Table 6 reflects that we found a (sometimes very) high number of smelly test suites in our manually annotated set of developer-written test suites. For instance, compared to EvoSuite, manually written tests were four times as

Table 6: Distribution of test smells in manually-written test suites.

Smell	**Manual Tests**
Eager Test	80%
Assertion Roulette	82%
Indirect Testing	20%
Sensitive Equality	10%
Mystery Guest	0%
Resource Optimism	10%

Table 7: Detection performance of different automated test smell detection tools for manually-written test cases. FPR denotes the False Positive Rate and FNR is the False Negative Rate.

Test smell	Tool used by (Grano et al., 2019)					tsDetect calibrated by (Spadini et al., 2020)				
	FPR	FNR	Prec.	Rec.	F1	FPR	FNR	Prec.	Rec.	F1
Assertion Roulette	0.74	0.23	0.62	0.76	0.68	0.00	0.20	1.00	0.80	0.89
Eager Test	0.60	0.31	0.81	0.69	0.75	0.10	0.61	0.94	0.39	0.55
Mystery Guest	0.11	—	—	—	—	0.01	—	—	—	—
Sensitive Equality	0.09	0.80	0.20	0.20	0.20	0.00	0.00	1.00	1.00	1.00
Resource Optimism	0.05	0.80	0.33	0.20	0.25	0.05	0.80	0.33	0.20	0.25
Indirect Testing	0.00	1.00	—	0.00	—	—	—	—	—	—

likely to contain the Eager Test smell and three times as likely to have the Assertion Roulette smell. Automatically generated test suites, on the other hand, are much more likely to contain the sensitive equality smell, as they frequently base assertions on invocations to a `toString` method. On the other hand, EvoSuite and JTExpert did not contain any instances of Resource Optimism test smell, whereas we do observe some test suites that make optimistic assumptions about file system availability and run time performance among developer-written ones.

We also study how the test smell detection tools perform on the manually written test suites in Table 7. We see that just as in the case of the automatically generated test suites, tsDetect implemented with the new thresholds introduced by Spadini *et al.* (Spadini et al., 2020) outperforms the tool used by Grano *et al.* (Grano et al., 2019) in terms of both precision and recall. When compared to the automatically generated test suites by EvoSuite, the tool used by Grano *et al.* performs better on manually written test suites, which appears to reflect that this tool was developed with human-written tests in mind. Surprisingly, we observe that the tool suffers in precision when detecting Sensitive Equality. This might be because the `toString` method is almost never directly invoked in an `Assert` method, which the Grano *et al.* tool cannot handle. We also note that both tools, which each claim to be able to detect Resource Optimism, perform poorly in detecting actual cases of this smell. This may be because their definitions and detecting strategies for this smell are overly narrow. Finally, the tool used by Grano *et al.* performs very poorly at detecting Indirect Testing, finding 0 instances of it. This is identical to this tool's performance on the automatically generated test suites – indeed, to the best of our knowledge, it is *incapable* of detecting this smell, despite claiming to.

Finding 4. Human-written test suites differ markedly from automatically generated ones in some smells (sensitive equality, indirect testing), while yielding comparable or higher rates on others (eager testing, assertion roulette). The performance of the test smell detection tools is better for some of the basic smells such as assertion roulette, however, for smells such as resource optimism and indirect testing we see very poor performance.

4.5 RQ5: How well do test smells capture real problems in manually written tests?

Our previous research questions focused on the distributions of test smells among manually-written and automatically generated test cases and the accuracy of existing automated tools in detecting such smells. However, the existing catalog of test smells was introduced two decades ago but never updated despite the progress made in unit testing frameworks. In this section, we investigate whether test smells still adequately capture flaws in real-world tests. This investigation is very critical since it potentially impacts the practical usefulness of existing test smell detection tools.

Eager Testing. The results in Section 4.3 indicate that eager tests are ubiquitous in automatically generated test-suites, but that their detection criterion is overly restrictive and reflects poorly actual concerns found in these tests. Eager tests are defined as testing more than one feature and are detected by tools based on whether more than one method invocation are made to the class under test. In a unit test, this occurs commonly, either in an assert statement or outside, due to the different setups needed for an object or a variety of intermediate results that need to be tested. This is especially true for tests that are designed to exercise more complex scenarios (integration tests).

Present analyzers failed to separate genuinely problematic cases from common testing conventions due to the rigid standards associated with both the definition and, even more so, detection of eager tests. In other words, its criterion is a poor discriminator. We now investigate this phenomenon more deeply on this human-authored set of test cases. In particular, we include a new criterion - Semantic Coherence - which challenges the notion of an eager test. We consider a test Semantically Coherent if it asserts only (transitive) properties and attributes of the class under test that all relate to a single testing scenario. To detect scenario(s) tested, we analyze the set of properties asserted within a test case and then compare this to the stated purpose of the test, per its nomenclature and comments. Two annotators annotated the same 49 samples and resolved differences in discussion. This proved to be a straightforward exercise with little ambiguity: the method name and structure typically made its

```
public void testCreate() throws Exception {
    String pk = ServiceTestUtil.randomString();
    Counter counter = _persistence.create(pk);
    Assert.assertNotNull(counter);
    Assert.assertEquals(counter.getPrimaryKey(), pk);
}
```

Fig. 8: Example of a semantically coherent test from the freemind project, that is nevertheless "eager" according to traditional test smell detection standards.

purpose very clear. If the asserts relate to more than one distinct testing scenario, the test is marked as incoherent for the purposes of unit testing, whose guidelines state that such a test should be refactored into multiple independent test cases. Compared to the common detectors for eager tests, this label is more permissive of multiple asserts and *transitive* properties and attributes, which includes asserts on fields, or parameters passed to constructors, as in Figure 8. Such properties all concern the full scope of the expected state of the class under test following a singular scenario (here: create a 'Counter') and are frequently tested together in test cases in practice.

In our manual analysis, we marked 39 out of the 49 manually written test suites studied[7] as Eager (ca. 80%). Yet, out of these 39, all but 4 were entirely semantically coherent. Meanwhile, among the remaining 10 non-eager suites we found one other non-coherent test case. These results indicate several key findings: for one, Eager Tests are so abundant in what were largely well-written test cases as to render the label *irrelevant*; secondly, eagerness correlated with semantic incoherence in just 10% of the cases, making it a very poor predictor for such problems in modern-day unit tests. In fact, that rate is no different than what we found among the 10 non-eager test suites (albeit with a caveat on the small sample size). We thus found no evidence that the Eager Test smell is a useful discriminator.

An example of a test case that was marked eager, but we consider to be coherent can be seen in Figure 8, which involves on method invocation to the `Counter` class (the class under test) and one that creates an initial state for the class by invoking a method on `ServiceTestUtil`. These method invocations have to co-occur for the purpose of the test scenario and cannot be refactored into separate tests; as such, this test should not be considered smelly.

An example of a test case that can be considered to be eager, as well as semantically incoherent is shown in Figure 9. Here we observe that the same scenario is being tested twice; *i.e.*, two distinct inputs to the `tokenize` method are being tested with no object state relation between them. In this case, the test would be better off getting split into two distinct tests, or into a parameterized test. We see similar aspects in the other three test suites that are both eager and incoherent. This might be due to the developers not deeming it worthwhile to split tests into multiple tests, for fear of introducing clones;

[7]One test suite was disregarded as it contained no test cases.

```java
public void testNumberOfGeneratedTokens() {
    String s;
    String[] result;
    // no numbers included
    s = "HOWEVER, the egg only got larger and larger, and more and more human";
    try {
      result = Tokenizer.tokenize(m_Tokenizer, new String[]{s});
      assertEquals("number of tokens differ (1)", 13, result.length);
    }
    catch (Exception e) {
      fail("Error tokenizing string '" + s + "'!");
    }
    // numbers included
    s = "The planet Mars, I scarcely need remind the reader, revolves about the sun at a"
                                     + "mean distance of 140,000,000 miles";
    try {
      result = Tokenizer.tokenize(m_Tokenizer, new String[]{s});
      assertEquals("number of tokens differ (2)", 19, result.length);
    }
    catch (Exception e) {
      fail("Error tokenizing string '" + s + "'!");
    }
  }
```

Fig. 9: Example of a semantically *incoherent* test from the Weka project

or, these test suites may still be using JUnit 3 (as in the case of Figure 9), where test parameterization is hard.

Assertion Roulette. We found that 30 out of the 49 test suites contain the assertion roulette smell (ca. 60%). Out of these, 10 test suites are based on JUnit 3. For these test suites/cases, it is important that the failure reason behind an assert is documented, as JUnit 3 will not relate the exact failing assert to the developer. A significant caveat, even there, is that many asserts that were undocumented were of the kind `assertNull` or `assertNotNull`, whose failure is rather self explanatory. Meanwhile, the other 20 test suites are based on JUnit 4 where an assert failure is explained by the JUnit runtime, and the very notion of the lack of documentation as a "smell" is thus rather doubtful.

Indirect Testing. We found 10 test suites that contained the indirect testing smell (ca. 20%). In contrast to the previous two smells, this is sharply lower than the figures for automatically generated tests. This is evidently a concern that developers do seek to avoid, lending more credence to its definition and usefulness. However, even 20% is somewhat inflated by an overly zealous definition: in many of these cases, the indirect feature being tested belonged to the Java language API, *e.g.*, using `List.size()`.

Figure 10 shows one such case: we see that the first assert checks that the iterator has an element before the loop is executed. The `hasNext` invocation is on the iterator and indirectly tests it. In the context of this test case, we see that the assert on this invocation is required and tests to ensure that a part of the parser works correctly. Under the strictest definition of indirect testing, this would be considered problematic. However, the usefulness of such

```
@Test
public void iterator(){
     Iterator<Nucleotide> expected = Nucleotides.parse(gappedBasecalls).iterator();
     Iterator<Nucleotide> actual = sut.iterator();
     assertTrue(actual.hasNext());
     while(actual.hasNext()){
         assertEquals(expected.next(), actual.next());
     }
     assertFalse(expected.hasNext());
 }
```

Fig. 10: Example of a test with the indirect testing smell from the JVCI Java commons project

```
@Test
public void testGetSecurePassword() {
    long start = System.currentTimeMillis();

    for (int i = 0; i < 100000; i++) {
        PwdGenerator.getPassword();
    }

    long end = System.currentTimeMillis();

    long delta = end - start;

    if (_log.isInfoEnabled()) {
        _log.info(
            "Generated 100 thousand secure passwords in " + delta + " ms");
    }

    Assert.assertTrue(delta < 2000);
}
```

Fig. 11: Example of a test case from the life ray project that is optimistic of the resources it has at it disposal

a standard is dubious; the Java core API is surely the most thoroughly vetted Java code in existence so that requiring mocking of basic `List` or `Iterator` methods is virtually pointless. The developers who wrote these tests evidently agree on this front; so, our findings suggest that invocation of a Java API should be exempt.

Sensitive Equality. We found five test suites that contain the sensitive equality test smell (ca. 10%). This is again vastly lower than the rate of incidence in automatically generated tests, suggesting that this too is a pattern developers try to avoid. What is more, in two of these five test suites we observe that the `toString` method is invoked as part of a test that explicitly means to test the `toString` method, immediately rendering the smell incorrect. In the other three cases, a cursory investigation suggested that the only way to get the value of the object under test was through its string representation. While this is arguably indicative of poor implementation of the underlying object, that should hardly qualify as a smell in the *testing* code.

Resource Optimism. Five out of the 49 test suites contain the resource optimism test smell. In all four cases, the test suite makes assumptions about the presence of a file system with a certain structure and containing one or more files. Such tests will fail in the event that (1) the file system is unavailable to the test, (2) the folder structure (along with its naming convention *e.g.*, base path starts with `tmp`) is incorrect or, (3) the operating system on which the test is being executed is non-Unix based. One especially strange case, shown in Figure 11, encodes the strong assumption that enough processing power is available to generate 100,000 passwords in 2 seconds.

Finding 5. In most manually-written test cases that contain a test smell (based on strict adherence to the test smell definition), real concerns were exceedingly rare and test smells indicated those very poorly.

5 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.

5.1 Manually Written Tests: Are Smells a Problem?

We investigate how the *"smelliness"* of tests written by actual developers compares to automatically generated test suites. Overall, if we adhere to the strictest definitions of these test smells, then the manually written tests in our dataset are highly smelly – often more so than automatically generated ones – and therefore low in maintainability. However, diving deeper into the nature of the smells themselves, we found that those smelly human-written tests were rarely problematic; instead, the detected smells were often just an artifact of the feature being tested, *e.g.*, as in Figure 8, where the test is eager but semantically coherent. These tests cannot be easily refactored or altered to mitigate their test smells, nor would doing so improve their quality or coherence.

This prevalence of smelly-yet-decent tests stands in contrast to the automatically generated tests. For example, `toString`-based comparisons were far more rampant and nearly always problematic there, whereas we found them to be both rare and essentially harmless in developer-written tests. In contrast, the bulk of these test suites contained eager tests, more so than in generated ones; yet, nearly none of the former were semantically incoherent, whereas poorly related assertions were abundant in the latter. Anecdotally, the test smells were simply *less useful* on human-written test suites.

In fact, this was a pattern across the board on human-written tests: the vast majority of each category of test smell detected based on rule-based heuristics (whether ours or a tool's) held up just fine to manual inspection. Highly prevalent test smells virtually never correlated with issues of semantic incoherence, and rarer ones often had a justifiable explanation. We identified just a few test cases that could be refactored, such as the one in Figure 9, where the same feature is being tested twice with different inputs. Most of these were semantically incoherent, as per our introduced definition (Section 4.5). Here parametric tests could be of use to alleviate the test smell, although there may be technological limitations in place as the project uses JUnit 3. In Figure 11, we see a test that makes assumptions about the processing speed of the device on which the test is being executed. This test needs certain guards to ensure that it is not flaky and does not fail in cases where insufficient processing power is available.

All this calls into question the nuance with which current test smell detection operates. Not only do current tools tend to be relatively inaccurate to our labeling (Table 4), this labeling itself was based on the blind application of largely outdated and overly restrictive rules that lead to an abundance of false positives when it comes to identifying real concerns. Based on a deep dive into these smells and their relation to maintainability, only a very small set of smelly tests could be considered to have real issues, and *no single test smell* effectively separated those from the rest. Taking a step back, this dataset serves as a reality check on the field of test smell detection: experienced developers wrote and maintained these tests over many years – some of these test suites are more than 10 years old – and we found the bulk to be highly readable and coherent; yet, test smells are apparently rampant across this dataset. If the objective of test smells is indeed to identify maintainability issues, then they have failed in both definition and implementation.

5.2 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 (Deursen et al., 2001), 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.* (Spadini et al., 2020) set a higher threshold than Grano *et al.* (Grano et al., 2019), 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.* (Deursen et al., 2001). 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 (Grano et al., 2019; Spadini et al., 2020), 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 (Grano et al., 2019); 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 (Grano et al., 2019). 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(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. 12: Example of tests with no assertions.

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 bytecode 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.

5.3 On Issues in Automatically Generated Tests 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 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 12a, where the assert is commented out by EvoSuite due to instability concerns, which results in this test case having no asserts. In Figure 12b, EvoSuite generates a test case with just a constructor invocation

```
@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. 13: Example of a test case with failed setup.

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[8]. This reflects a disconnect between the optimization metric of “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 13 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`;

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

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.

6 Threats to Validity

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

Threats to external validity. In this study, we assess the presence of test smells in tests generated by EvoSuite, JTExpert, and on a sample of manually written tests. The selective nature of the sample set of tests can have an impact on the generalizability of our observations and results. Nonetheless, many of our observations concern problems caused by the discrepancy between generated tests and those a human might write. The limited sample size does not invalidate our specific findings for the six smells we study, but does imply that further studies are needed to confirm whether similar conclusions apply to other test smells. The broader results 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.

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.

Validating our annotations for the manual tests with the projects' developers could have potentially strengthened our observations and conclusions. However, this was not possible for the projects in the SF110 dataset, which consists of projects collected from SourceForge in 2014. These projects are not actively maintained, or their versions within the dataset are outdated. Despite these limitations for the SF110 benchmark, we opted to use the same dataset and the same classes selected by Grano et al. (2019) to ease the comparison with their results and analysis. Extending our analysis with actively-maintained projects and developers is part of our future agenda.

7 Discussion, Lessons Learned, and Future Directions

In this section, we first summarize our results and presents possible future research directions in test smells and test case generation.

7.1 Lesson Learned

Lesson 1. *A non-trivial portion of the generated test cases contains at least one test smell. However, the occurrence is much less frequent than reported in prior studies* (Grano et al., 2019; Palomba et al., 2016). For example, mystery guest and resource optimism have been reported in prior studies as being frequent in automatically generated tests. However, these smells cannot occur due to the mechanisms tools like EvoSuite and JTExpert use to prevent creating random files, potentially damaging the machine on which experiments are performed. The substantial differences between our results and those reported in prior studies are due to the different evaluation processes. Grano et al. (Grano et al., 2019), and Palomba et al. (Palomba et al., 2016) did not manually validate the warnings raised by test smell detection tools. This was done under the very optimistic (and not realistic) assumption that these tools are highly accurate.

Lesson 2. *test smell detection tools are very inaccurate in detecting test smells for automatically generated test cases.* First, the two state-of-the-art detection tools largely overestimate the occurrences of assertion roulette, eager tests, resource optimism, and mystery guest. The root causes for the low accuracy differ depending on the type of test smell under analysis. We can summarize our findings as follows:

– For assertion roulette and eager tests, existing tools simply rely on rule sets that count the number of assertions and method calls in a test case as a proxy for "the number of functionalities" under test and that are asserted. Our results suggest that such simple heuristics are highly inaccurate.
– Test smells detection tools based on pure static rules cannot adequately determine if test cases actually access external files and resources, leading to a very large false-positive rate for mystery guests and resource optimism.
– None of the test smell detection tools could detect indirect testing instances, which are the most frequent test smell in the test cases generated by EvoSuite.
– Many instances of sensitive equality were undetected due to incomplete rule sets for covering (many) corner-case scenarios and patterns.

Lesson 3. Given the results of our first two research questions, in RQ3 we investigated whether the existing catalog of test smells reflects real maintenance problems in automatically generated test cases. Our results indicate that *generated test cases are affected by eager tests and multiple assertions, but their severity is debatable.* As shown by Spadini et al. (Spadini et al., 2018), developers consider these types of smells as non-problematic with a very low severity or priority for test fixing operations. Instead, mystery guest and resource optimism do not represent real problems for modern test case generation tools that use mocks or bytecode instrumentation techniques.

Lesson 4. Since test smell detection tools have been designed for manually-written tests, one could argue that the low accuracy observed in RQ2 is due to the intrinsic differences between test cases written by developers and those generated with automated techniques. The results of RQ4 showed that *the distribution of test smells and their occurrences differ between manually-written and generated tests.* Eager tests, assertion roulette, mystery guests, and resource optimism are more frequent among test cases written by developers compared to those automatically generated. Vice versa, sensitive equality and indirect testing are more common in test cases automatically generated.

Lesson 5. *The accuracy of test smell detection tools is higher for manually-written test cases compared to those generated in an automated fashion.* This is partially due to the fact that these tools have been designed and tuned for test cases written by humans. However, *the detection tools assessed in our study poorly perform for indirect testing, resource optimism, and mystery guests.*

Lesson 6. After carefully validating the test smells instances among manually written test cases, we conclude that *test smells do not reflect real concerns with respect to test maintainability.* Our results lead to similar conclusions by Spadini at al. (Spadini et al., 2018), whose study reported a large misalignment between test smells instances and what developers consider actual test maintainability concerns. Tufano et al. (Tufano et al., 2016) reported that the developers did not recognize any problems with the test code snippets they were presented with, although, in theory, those tests were affected by test smells. While prior studies questioned the developers' ability to recognize test smells, our results and analysis led to a completely different conclusion: many test smells (as currently defined and detected) do not reflect real concerns.

7.2 The Path forward Concerning Test Smells

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. Much the same applies to developer-written tests. Having analyzed the main causes for these tools' false positives, and failures of the test smell definitions themselves, it stands out that these are all static in nature and use rather simple heuristics (*e.g.*, relying on specific numerical thresholds). To better reflect concerns in development practice (and dramatically reduce false-positive rates), we propose the following steps to review test smell definitions and 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 are 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 and Eager Testing, appeared to be imprecise and incomplete. Currently, their definitions strongly relate both to the presence of any invocation to a class besides the one under test. This is incongruent with the practice for multiple reasons (*e.g.*, using Java libraries, tests that naturally involve two steps). There is thus a need for revising these, and indeed all test smells', definitions that are more precise and captures the notion of a *semantic objective* – a specific, coherent, 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 the 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.
4. While our study focuses on Java code and tests, our conclusions with regards to test smells and their practical relevance also apply to other programming languages. Nowadays, all unit testing frameworks (e.g., `unittest` in Python) explicitly indicate which assertion fail and the reason for it, making assertion roulette outdated as potential test smell.

7.3 Research Opportunity in Test Case Generation

When manually validating the generated test cases, we also identified a number of research opportunities for the test case generation (and fuzzing) community:

1. EvoSuite already implements a greedy test minimization routine that removes unnecessary statements from test cases. However, the results test case might still be affected by test smells, like semantic coherence and eager test. While most of the research effort has been dedicated to reaching high coverage, more research is needed w.r.t. post-process optimization. Techniques like test slicing (Messaoudi et al., 2021), carving (Elbaum et al., 2006), and purification (Xuan and Monperrus, 2014) have been applied to manually-written tests and could be potentially included in test case generation tools.
2. Existing heuristics used to guide test case generation tools measure how far each test case is from covering a given branch b independently of whether b is directly covered or not. This explains why indirect testing is common (<50%) among the randomly generated tests. The state-of-the-art heuristics used to give the search should differentiate between direct and indirect coverage and promote (by giving better fitness values to) test cases that

directly cover branches in the code. This could be achieved by using secondary objectives (Panichella et al., 2017) or changing how the archive is updated during the search (Rojas et al., 2017)

3. As indicated in Section 5.3, test generation tools sometimes have problems in generating a proper setup for classes that are complex to instantiate. A possible research direction to address these limitations is to focus on seeding strategy with particular attention to complex object instantiation (e.g., (Derakhshanfar et al., 2020)). Besides, researchers in the test case generation tools should also focus on generating test fixtures, e.g., setup code shared among all test cases in a test suite.
4. Last but not least, a lot of research effort has been devoted to improving the readability of the generated test cases, e.g., by generating more readable inputs (Afshan et al., 2013), better test names (Daka et al., 2017; Roy et al., 2020), and automated documentation (Panichella et al., 2016; Roy et al., 2020). These studies proposed approaches that make the generated test cases easier to understand and manually validate. To the best of our knowledge, no study has been conducted to assess whether the generated tests are equally/more/less difficult to maintain in the longer term than manually written ones.

8 Conclusions

This paper investigates test smell occurrence in automatically generated test cases and the extent to which contemporary test smell detection tools can identify them. We built a dataset spanning hundreds of automatically generated and human-written test cases for complex Java classes and conducted multi-stage, manual cross-validation to identify six types of test smells. Our results show that test smells are commonly present in a small but non-trivial portion of the test suites generated by EvoSuite. However, they occur far less often than reported by the tool (and analysis) of Grano *et al.* (Grano et al., 2019), while tsDetect achieves somewhat better results (Spadini et al., 2018). The incidence rate of test smells is much larger for the test suites generated by JTExpert; this number could evidently be dramatically reduced by using post-process test suite optimization heuristics well-known in the testing literature (e.g., (Fraser and Arcuri, 2015b; Xuan and Monperrus, 2014; Yoo and Harman, 2012)) and implemented in EvoSuite.

Test smell detection tools that rely on static rules are limited and inaccurate in detecting certain types of more complex test smells in automatically-generated tests, e.g., indirect testing and resource optimism. In addition, although EvoSuite 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 (Spadini et al., 2018)).

On the other hand, tests written by developers were ostensibly riddled with test smells, some much more so than tool-generated ones – although here, too, detection tools often overestimate their incidence. However, upon detailed inspection, nearly all of these failed to reflect genuine problems, which were rare and uncorrelated with any of the smells. As such, the definition and detection of test smells is poorly matched to modern testing practice.

The discrepancies between tool quality, definition, and practical accuracy exposed by our work suggest the need for further studies that involve human participants (preferably in industrial contexts), which will be critical to ensure that the notion and design of test smells and their detection tools better reflect developer practice.

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. Annibale Panichella and Sebastiano Panichella gratefully acknowledges the Horizon 2020 (EU Commission) support for the project *COSMOS* (DevOps for Complex Cyber-physical Systems), Project No. 957254-COSMOS.

Conflicts of interests

The authors declared that they have no conflict of interest.

References

S. Afshan, P. McMinn, and M. Stevenson. Evolving readable string test inputs using a natural language model to reduce human oracle cost. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, pages 352–361. IEEE, 2013.

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*, pages 263–272, 2017.

P. Ammann and J. Offutt. *Introduction to software testing.* Cambridge University Press, 2016.

J. H. Andrews, T. Menzies, and F. C. Li. Genetic algorithms for randomized unit testing. *Ieee transactions on software engineering*, 37(1):80–94, 2011.

A. Arcuri and G. Fraser. Parameter tuning or default values? an empirical investigation in search-based software engineering. *Empirical Software Engineering*, 18(3):594–623, 2013.

A. Arcuri, G. Fraser, and J. P. Galeotti. Automated unit test generation for classes with environment dependencies. In *International conference on Automated software engineering*, pages 79–90, 2014.

L. Baresi and M. Miraz. Testful: Automatic unit-test generation for java classes. In *International Conference on Software Engineering*, volume 2, pages 281–284, 2010.

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*, pages 56–65, 2012.

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

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

C. Birchler, N. Ganz, S. Khatiri, A. Gambi, and S. Panichella. Cost-effective simulationbased test selection in self-driving cars software with sdc-scissor. In *2022 IEEE 29th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2022a. doi: toappear.

C. Birchler, S. Khatiri, P. Derakhshanfar, S. Panichella, and A. Panichella. Single and multi-objective test cases prioritization for self-driving cars in virtual environments. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2022b. doi: toappear.

M. Böhme, L. Szekeres, and J. Metzman. On the reliability of coverage-based fuzzer benchmarking. 2022.

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*, 104:207–235, 2018.

G. Canfora, A. D. Lucia, M. D. Penta, R. Oliveto, A. Panichella, and S. Panichella. Multi-objective cross-project defect prediction. In *Sixth IEEE International Conference on Software Testing, Verification and Validation, ICST 2013, Luxembourg, Luxembourg, March 18-22, 2013*, pages 252–261. IEEE Computer Society, 2013. doi: 10.1109/ICST.2013.38. URL `https://doi.org/10.1109/ICST.2013.38`.

G. Canfora, A. D. Lucia, M. D. Penta, R. Oliveto, A. Panichella, and S. Panichella. Defect prediction as a multiobjective optimization problem. *Softw. Test. Verification Reliab.*, 25(4):426–459, 2015. doi: 10.1002/stvr.1570. URL `https://doi.org/10.1002/stvr.1570`.

C. Csallner and Y. Smaragdakis. Jcrasher: an automatic robustness tester for java. *Software: Practice and Experience*, 34(11):1025–1050, 2004.

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*, pages 107–118, 2015.

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*, pages 57–67, 2017.

P. Derakhshanfar, X. Devroey, G. Perrouin, A. Zaidman, and A. van Deursen. Search-based crash reproduction using behavioural model seeding. *Software Testing, Verification and Reliability*, 30(3):e1733, 2020.

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)*, pages 92–95, 2001.

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. doi: 10.1145/3387940.3392265.

S. Elbaum, H. N. Chin, M. B. Dwyer, and J. Dokulil. Carving differential unit test cases from system test cases. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 253–264, New York, NY, USA, 2006. Association for Computing Machinery.

M. Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.

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*, pages 416–419, 2011.

G. Fraser and A. Arcuri. Whole test suite generation. *IEEE Transactions on Software Engineering*, 39(2):276–291, 2012.

G. Fraser and A. Arcuri. A large-scale evaluation of automated unit test generation using evosuite. *Trans. Softw. Eng. Methodol.*, 24(2):8:1–8:42, Dec. 2014. ISSN 1049-331X.

G. Fraser and A. Arcuri. 1600 faults in 100 projects: automatically finding faults while achieving high coverage with evosuite. *Empirical software engineering*, 20(3):611–639, 2015a.

G. Fraser and A. Arcuri. Achieving scalable mutation-based generation of whole test suites. *Empirical Software Engineering*, 20(3):783–812, 2015b.

G. Fraser and A. Zeller. Mutation-driven generation of unit tests and oracles. *Transactions on Software Engineering*, 38(2):278–292, 2011.

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*, 156:312–327, 2019.

R. Just, M. D. Ernst, and G. Fraser. Efficient mutation analysis by propagating and partitioning infected execution states. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 315–326, 2014.

F. Kifetew, X. Devroey, and U. Rueda. Java unit testing tool competition-seventh round. In *International Workshop on Search-Based Software Testing*, pages 15–20, 2019.

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)*, pages 212–223. IEEE, 2015.

S. Messaoudi, D. Shin, A. Panichella, D. Bianculli, and L. C. Briand. Log-based slicing for system-level test cases. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 517–528, 2021.

C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *International Conference on Software Engineering*, pages 75–84, 2007.

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

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

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*, 44(2):122–158, 2017.

A. Panichella, F. M. Kifetew, and P. Tonella. Incremental control dependency frontier exploration for many-criteria test case generation. In *International Symposium on Search Based Software Engineering*, pages 309–324. Springer, 2018a.

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*, 104:236–256, 2018b.

A. Panichella, J. Campos, and G. Fraser. Evosuite at the sbst 2020 tool competition. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, pages 549–552, 2020a.

A. Panichella, S. Panichella, G. Fraser, A. A. Sawant, and V. J. Hellendoorn. Revisiting test smells in automatically generated tests: limitations, pitfalls, and opportunities. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 523–533. IEEE, 2020b.

S. Panichella. Supporting newcomers in software development projects. In R. Koschke, J. Krinke, and M. P. Robillard, editors, *2015 IEEE International Conference on Software Maintenance and Evolution, ICSME 2015, Bremen, Germany, September 29 - October 1, 2015*, pages 586–589. IEEE Computer Society, 2015. doi: 10.1109/ICSM.2015.7332519. URL `https://doi.org/10.1109/ICSM.2015.7332519`.

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*, pages 547–558, 2016.

S. Panichella, A. Gambi, F. Zampetti, and V. Riccio. Sbst tool competition 2021. In *International Conference on Software Engineering, Workshops, Madrid, Spain, 2021*. ACM, 2021.

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

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*, pages 23–32, 2011.

J. M. Rojas, J. Campos, M. Vivanti, G. Fraser, and A. Arcuri. Combining multiple coverage criteria in search-based unit test generation. In *International Symposium on Search Based Software Engineering*, pages 93–108.

Springer, 2015.

J. M. Rojas, G. Fraser, and A. Arcuri. Seeding strategies in search-based unit test generation. *Software Testing, Verification and Reliability*, 26(5): 366–401, 2016.

J. M. Rojas, M. Vivanti, A. Arcuri, and G. Fraser. A detailed investigation of the effectiveness of whole test suite generation. *Empirical Software Engineering*, 22(2):852–893, 2017.

D. Roy, Z. Zhang, M. Ma, V. Arnaoudova, A. Panichella, S. Panichella, D. Gonzalez, and M. Mirakhorli. Deeptc-enhancer: Improving the readability of automatically generated tests. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 287–298. IEEE, 2020.

A. Sakti, G. Pesant, and Y.-G. Guéhéneuc. Instance generator and problem representation to improve object oriented code coverage. *Transactions on Software Engineering*, 41(3):294–313, 2014.

A. Sakti, G. Pesant, and Y. Guéhéneuc. Jtexpert at the SBST 2017 tool competition. In *10th IEEE/ACM International Workshop on Search-Based Software Testing, SBST@ICSE 2017, Buenos Aires, Argentina, May 22-23, 2017*, pages 43–46. IEEE, 2017. doi: 10.1109/SBST.2017.5. URL `https://doi.org/10.1109/SBST.2017.5`.

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*, pages 250–261, 2018.

M. Soltani, A. Panichella, and A. Van Deursen. Search-based crash reproduction and its impact on debugging. *IEEE Transactions on Software Engineering*, 46(12):1294–1317, 2018.

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*, pages 1–12, 2018.

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

P. Tonella. Evolutionary testing of classes. *ACM SIGSOFT Software Engineering Notes*, 29(4):119–128, 2004.

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

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*, pages 4–15, 2016.

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.*, 43(11):1063–1088, 2017.

T. Xie. Augmenting automatically generated unit-test suites with regression oracle checking. In *European Conference on Object-Oriented Programming*, pages 380–403. Springer, 2006.

J. Xuan and M. Monperrus. Test case purification for improving fault localization. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 52–63, 2014.

S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: a survey. *Software testing, verification and reliability*, 22(2):67–120, 2012.

S. Zhang. Practical semantic test simplification. In *International Conference on Software Engineering*, pages 1173–1176, 2013.