Scented Since the Beginning: On the Diffuseness of Test Smells in Automatically Generated Test Code

Giovanni Grano[a], Fabio Palomba[a], Dario Di Nucci[b], Andrea De Lucia[c], Harald Gall[a]

[a] *University of Zurich, Zurich, Switzerland*
[b] *Vrije Universiteit Brussel, Brussels, Belgium*
[c] *University of Salerno, Fisciano, Italy*

Abstract

Software testing represents a key software engineering practice to ensure source code quality and reliability. To support developers in this activity and reduce testing effort, several automated unit test generation tools have been proposed. Most of these approaches have the main goal of covering as more branches as possible. While these approaches have good performance, little is still known on the maintainability of the test code they produce, *i.e.,* whether the generated tests have a good code quality and if they do not possibly introduce issues threatening their effectiveness. To bridge this gap, in this paper we study to what extent existing automated test case generation tools produce potentially problematic test code. We consider seven test smells, *i.e.,* suboptimal design choices applied by programmers during the development of test cases, as measure of code quality of the generated tests, and evaluate their diffuseness in the unit test classes automatically generated by three state-of-the-art tools such as Randoop, JTExpert, and Evosuite. Moreover, we investigate whether there are characteristics of test and production code influencing the generation of smelly tests. Our study shows that all the considered tools tend to generate a high quantity of two specific test smell types, *i.e., Assertion Roulette* and *Eager Test*, which are those that previous studies showed to negatively impact the reliability of production code. We also discover that test size is correlated with the generation of smelly tests. Based on our findings, we argue that more effective automated generation algorithms that explicitly take into account test code quality should be further investigated and devised.

Keywords: Test Smells, Test Case Generation, Software Quality, Empirical Studies

1. Introduction

Software testing is widely recognized as a fundamental practice to ensure the correct behavior of any successful software project [1]. As such, with the help of testing frameworks like, *e.g.,* JUnit, developers create test cases and run them periodically [2, 3]. In addition, developers can integrate manually created tests with automatically generated ones, which aim at testing behaviors of production code that humans can rarely exercise [4]. To support developers in this activity, many techniques and tools to automatically produce test suites with high code coverage have been proposed [5–9]. Besides techniques aimed at maximizing code coverage as main goal, also multi-objective approaches have been proposed to achieve additional objectives, such as the minimization of (i) oracle cost [10], (ii) dynamic memory consumption [11], (iii) number of test cases [12], (iv) execution time [13], (v) number of targets that are accidentally covered [14], and (vi) test code quality [15, 16]. Although the effort devoted by the research community to define techniques able to automatically generate test cases, there is still a lack of empirical investigations about the characteristics of the test code produced by such tools. More specifically, while recent studies have been conducted to evaluate the effectiveness of test cases [17–22] and the usability of these tools in practice [23], it is still unclear whether the test code automatically generated is immune from design problems that possibly negatively affect its effectiveness or the ability of developers to interact with them.

Motivation. As pointed out by Fraser and Arcuri [24], having well-designed generated tests is a key challenge for automated tools because the goal of these tests it to target difficult faults for which automated oracles are not available and that, therefore, must be *manually verified* after the generation process [24]. As such, developers are required to first comprehend automatically generated tests in order to properly design assertions. Furthermore, recent studies have shown that poorly designed test code may (i) induce the so-called flakiness, that is, a non-deterministic behavior of a test [19] and (ii) limit the fault detection capabilities of test cases [22]. Finally, in case one of the generated test cases fails, developers are still required to understand what caused the failure by manually analyzing the test.

For all these reasons, even if automatically generated tests can be continuously re-generated and not maintained

```java
1   public void test53()  throws Throwable  {
2     OrganizationModelImpl organizationModelImpl0 =
       new OrganizationModelImpl(''tmp.txt'');
3     OrganizationSoap organizationSoap0 = new
       OrganizationSoap();
4     Organization organization0 =
       OrganizationModelImpl.toModel(organizationSoap0);
5     organizationModelImpl0.setPrimaryKey(4294967295L)
       ;
6     OrganizationWrapper organizationWrapper0 = new
       OrganizationWrapper(organization0);
7     boolean boolean0 = organizationModelImpl0.equals(
       organizationWrapper0);
8     assertEquals(4294967295L, organizationModelImpl0.
       getPrimaryKey());
9     assertFalse(boolean0);
10  }
11
```

Listing 1: Example of test case generated by EvoSuite.

during the evolution of a project, there are still a number of major challenges making the design of test code relevant for automated test generation techniques.

To better illustrate the problem statement, let consider the following scenario. *John* is a developer of the Liferay project,[1] an open source portal to build personal and professional websites. Once implemented a new code change, *John* wants to ensure that such a change did not introduce regressions. Besides running the test cases already available in the project, he decides to complement them with those automatically generated by EvoSuite [5]—one of the state-of-the-art automated test case generation tools—in order to enlarge the test suite and possibly identify faults missed by existing tests. Before running the entire set of tests, *John* has to manually check all the assertions of the generated tests: this is a time-consuming activity, especially because he has to understand what each automatically test case is about. For instance, one of those generated test cases is shown in Listing 1. Before checking the expected values in the `assert` statements, *John* needs to figure out what is the target production method: in this case, it is not immediately clear because the test exercises the `getPrimaryKey` of the class `OrganizationModelImpl` (line 8 of Listing 1) as well as checking whether the object `organizationModelImpl0` is equal to `organizationWrapper0` (line 9 of Listing 1). As such, understanding the correct value to assign as first parameter of the assertions requires the analysis of the program flow. In any case, once completing this manual analysis, *John* can finally execute the test suite. As a result, the test shown in Listing 1 fails; for this reason, *John* starts debugging the code to find the cause of the failure. He starts by looking at the assertions, however they do not provide any explanation, thus making the debugging phase harder. At the end, *John* realizes that the production code does not exhibit any real fault, but the test case failed just because it relies on an external resource that is never initialized (line 2 of Listing 1). Once fixed this test design problem, the process runs fine and *John* can finally put his source code change in production.

[1]https://dev.liferay.com

Our work and contribution. In our previous work [25], we started analyzing the design of automatically generated test code by conducting a large-scale empirical study on the `SF110` dataset, a set of 110 open source software projects [26], to investigate to what extent the test classes automatically generated by EvoSuite [5] are affected by *test smells* [27], *i.e.,* symptoms of the presence of poor design or implementation choices in test code. We explicitly focused on test smells for two reasons. In the first place, they represent a good measure to quantify the quality of tests with respect to their design. Furthermore, test smells have been connected by previous research to all the key issues related to the design of automatically generated test cases: they indeed hinder test code comprehensibility and understandability [28] as well as test code effectiveness [19, 22]. As such, they may significantly impact the developers' ability to use automatically generated test cases for improving source code reliability. The results achieved in our preliminary study showed that two test smells (*i.e., Assertion Roulette* and *Eager Test*) were particularly diffused in automatically generated test classes, but also that the presence of test smells was strongly correlated to characteristics of the project such as size and number of classes [25].

In this paper, we build on top of our previous analyses with the aim of enlarging our empirical knowledge on the relation between test smells and automated test case generation tools. Specifically, we extend our original work with four new main contributions:

1. We investigate the behavior of three state-of-the-art automated test case generation tools such as Randoop [29], JTExpert [9], and EvoSuite [5] which have different underlying generation algorithms. This analysis aims at providing a wider overview on whether and how test smells represent a problem for different testing tools.

2. We improve our analysis method by running our experiments on a curated dataset composed of 100 *non-trivial* classes. Indeed, Shamshiri *et al.* [30] showed that the vast majority of the classes available in the `SF110` corpus are branchless and, therefore, the tests generated for those classes are meaningless and worthless to analyze in studies aimed at understanding the characteristics of automated tools [7, 30]. Thus, we verify the presence of test smells only considering test classes that actually put automated test case generation tools in action.

3. We perform an analysis on the relationships between test/production code quality (as measured by the Chidamber and Kemerer's software metrics suite

[31]) and the likelihood that the experimented automated test case generation tools produce smelly tests.

4. We conduct a fine-grained manual analysis aimed at investigating whether the characteristics of the generation algorithms employed by the considered tools impacts on the test smell introduction.

The key results of our study show that all the considered tools naturally output a large amount of two test smell types, namely *Assertion Roulette* and *Eager Test*, which are the ones that previous findings revealed to significantly hinder test code effectiveness [22]. The nature of these tools is the cause of "smelliness". Indeed, they are too much eager to cover parts of source code, being focused on code coverage, independently from any other aspect. Furthermore, we find that the test size is correlated with the generation of smelly tests, meaning that the higher the amount of code generated by these tools the higher the chance of introducing test smells.

Structure of the paper. Section 2 reports the background and the literature related to test smells and automated test case generation. In Section 3 we describe the empirical study definition and design, while in Section 4 we report the results. The discussion and the implications of the study are part of Section 5, while we report the possible threats to the validity in Section 6. Finally, Section 7 concludes the paper.

2. Background and Related Work

The main goal of our study is to investigate the prominence of design problems in the test code automatically generated by existing tools. Therefore, it is at the intersection of two main topics, *i.e.,* test smells and automated test case generation. In the following subsections, we provide an overview of the both the topics as well as of the related literature.

2.1. Code Smells in Test Cases

Like production code, test code should also be designed following good programming practices [32]. During the last decade, the research community spent a lot of effort on the definition of methods and tools for detecting design flaws in production code [33–43], as well as empirical studies aimed at assessing their impact on maintainability [44–62].

However, design problems affecting test code have been only partially explored. The importance to have well designed test code was originally highlighted by Beck [63], while Van Deursen *et al.* [27] defined a catalog of 11 test smells, *i.e.,* a set of a poor design solutions to write tests, together with refactoring operations able to remove them. Such a catalog takes into account different types of bad design choices made by developers during the implementation of test fixtures (*e.g.,* a too generic `setUp()` method where test methods only access a part of it), or of single test cases (*e.g.,* test methods checking several objects of the class to be tested). In the context of this work, we limit our focus on a subset of the test smells defined by Van Deursen *et al.* [27], as not all of them can be studied when considering automatically generated code. Indeed, almost all the automated tools for generating test classes do not produce test fixtures [5, 7, 9, 64, 65]. For this reason, we focus on the seven test smell types that can potentially affect test code automatically generated and that are presented in the following:

Mystery Guest (MG). This smell arises when a test uses external resources (*e.g.,* file containing test data), and thus it is not self contained [27]. Such tests are difficult to comprehend and maintain, due to the lack of information to understand them. Furthermore, it may lead the test to be flaky [19]. To remove a *Mystery Guest* a *Setup External Resource* operation is needed [27].

Resource Optimism (RO). Tests affected by such smell make assumptions about the state or the existence of external resources, providing a non-deterministic result that depends on the state of the resources [27]. Also in this case, to remove the smell a *Setup External Resource* refactoring [27] is needed.

Eager Test (ET). A test is affected by *Eager Test* when it checks more than one method of the class to be tested [27], making the comprehension of the actual test target difficult. The solution is represented by the application of an *Extract Method* refactoring, able to split the test method in order to specialize its responsibilities [66].

Assertion Roulette (AR). As defined by Van Deursen *et al.* [27], this smell *"comes from having a number of assertions in a test method that have no explanation"*. Thus, if an assertion fails, the identification of the cause behind the failure can be difficult. Besides removing the unneeded assertions, to remove this smell and make the test clearer an operation of *Add Assertion Explanation* can be applied [27]. It is worth noting that one the explanatory power of an `assert` statement could be subjectively assessed, *e.g.,* a developer may easily understand an assertion because s/he an expert of the source code under test. Nevertheless, this test smell arises when the cause leading an assertion to fail is not *explicitly* stated within the assert statement. In particular, JUnit developers can include, as first parameter of an `assert` statement, a `String` message reporting the motivation behind a failure caused by that assertion. An *Assertion Roulette* appears if there are more than one assert statement where such a first parameter is not set.

Indirect Testing (IT). A test that checks the corresponding production class using methods of another class [27]. Such indirection, in addition to being a design error, can create problems in the comprehension of the sequence of calls performed by the test case during its activities and make the test prone to flakiness [19]. Van Deursen *et al.* [27] suggest to remove this smell by applying an *Extract Method* refactoring, followed by a *Move Method* one, in order to re-organize such indirection moving the methods to the appropriate test class.

For Testers Only (FTO). This smell arises when a production class contains methods only used by test methods [27]. This kind of production classes should be removed, since it does not provide functionalities used by other classes in the system. From the testing side, this smell involves an extra effort needed in order to comprehend and modify assertions [27].

Sensitive Equality (SE). When an assertion contains an equality check through the use of the `toString` method, the test is affected by a *Sensitive Equality* smell. In this case, the failure of a test case can depend on the details of the string used in the comparison, *e.g.,* commas, quotes, spaces etc. [27]. A simple solution for removing this smell is the application of an *Introduce Equality Method* refactoring, in which the use of the `toString` is replaced by a real equality check.

On top of the work done by Van Deursen *et al.* [27], Meszaros defined other smells affecting test code [67]. Starting from these two catalogs, Greiler *et al.* [68, 69] showed that test smells related to fixture set-up frequently occur in industrial projects and, therefore, presented a static analysis tool, namely TestHound, to identify fixture-related test smells. Van Rompaey *et al.* [70] proposed a heuristic structural metric-based approach to identify *General Fixture* and *Eager Test* instances. However, empirical results demonstrated that structural metrics have low accuracy in the detection of both test smells. This was later confirmed by Palomba *et al.* [71], who compared the performance of the techniques by Greiler *et al.* [68, 69] and Van Rompaey *et al.* [70] with the one achievable by Taste, a textual-based detector exploiting information retrieval techniques to identify three test smell types: they concluded that textual analysis can be more precise than the structural one when detecting test smells.

As for the empirical studies, Bavota *et al.* [28] conducted an empirical investigation in order to study (i) the diffusion of test smells in 18 software projects, and (ii) their effects on software maintenance. The results of the study demonstrated that 82% of JUnit classes in their dataset are affected by at least one test smell, but also that the presence of design flaws has a strong negative impact on maintainability. Tahir *et al.* [72] conducted an empirical study on the relation between production code quality metrics and test smells, finding that some design aspects, *e.g.,* Cyclomatic Complexity, are strongly related to the emergence of design flaws in test code. With respect to the work by Bavota *et al.* [28] and Tahir *et al.* [72], it is important to note that they investigated manually written tests, while automatically generated tests are diametrically different because they are created algorithmically. As such, it may be possible that the diffuseness of some test smells as well as their co-occurrences may be different than those of manually generated tests, as all generation algorithms may have peculiar ways to construct tests. This is indeed what happens in practice: for example, Bavota *et al.* [28] found that the *General Fixture* test smell often appears in combination with other test smells; however, this is not possible in automatically generated tests because the currently available tools do not generate fixtures at all. Similarly, several *Indirect Testing* instances were found in previous works targeting manually created tests, while in our case this is unlikely because each automatic test case generated explicitly covers a single production class. For this reason, we argue that, on the one hand, the diffuseness and co-occurrences of test smells in automatically generated tests are different from those of manually written ones and, on the other hand, being aware of the co-occurrences generated by automated tools would potentially help tool vendors and researchers in building novel algorithms that are aware of them while optimizing code coverage.

Spadini *et al.* [22] studied the relation of test smells to software quality, analyzing whether (i) smelly tests are more change- and fault-prone than other tests and (ii) production code exercised by smelly tests is more fault-prone. Results of their study reported that test smells represent an important problem for maintainability and reliability of software systems, as they negatively influence the quality of both production and test code. These findings were also supported by Palomba and Zaidman [19], who discovered that three test smells (*i.e., Indirect Testing*, *Resource Optimism*, and *Test Run War*) can induce tests to have a non-deterministic behaviour. Tufano *et al.* [73] studied (i) when test smells occur in source code, (ii) what is their survivability, and (iii) whether their presence is associated with the presence of code smells in production code. Key findings from their study highlight that test smells are generally introduced when the corresponding test code is committed in the repository for the first time, and that they tend to remain in a system for a long time. Furthermore, they discovered some relationships between test and code smells. Finally, De Bleser *et al.* [74] noted that most of the studies on test smells have been limited to Java in combination with the JUnit testing framework. Therefore, they studied the diffusion and perception of test smells in Scala in combination with the ScalaTest testing framework. They transposed the original test smell definitions to this new context, and implemented SoCRATES, a tool for their detection. Their results show that (i) test smells have a low diffusion across test classes; (ii) the most frequently occurring test smells are Lazy Test, Eager Test, and Assertion Roulette; and (iii) many developers are able to perceive but not to identify the smells.

With respect to the papers described so far, our work can be considered complementary. Indeed, to the best of our knowledge the empirical study proposed in this paper is the first one investigating the prominence of test smells in automatically generated test code.

2.2. *Automated Generation of Test Cases for Object Oriented Code*

Automated test case generation techniques have been conceived to reduce the cost of software testing and helping developers to maximize the percentage of code elements (*e.g.,* statements or branches) being exercised according to well-established code coverage criteria [75]. Broadly speaking, the process of automated generation of a test case for a certain method requires the selection of a test targets, the creation of method sequences able to cover it, and the introduction of test assertions. Last decades witnessed the introduction of several techniques for test case generation. In this section, we mainly focus on approaches based on both random and search-based software testing. Moreover, we describe the inner working of the three tools we compare in the presented study.

2.2.1. *Random Test Case Generation*

Tools based on random testing [9, 29, 76–78] randomly select methods and object constructors and invoke them by using previously computed values as input. In particular, JCrasher [76] finds method calls whose return values can be used as input parameters, but it does not analyze the execution feedback. Eclat [78] discards the sequences of method calls that make the program behave differently than a set of correct training runs. Therefore, the performances of the tool are strongly influenced by the quality of the sample execution given as input.

Randoop. Randoop [29] similarly incrementally generates inputs by randomly selecting a method call to apply. The algorithm that builds the sequences works as follow. The sequences are built incrementally, starting from an initial empty set. At first, a method m belonging to the production class is randomly selected. Thus, Randoop applies an extension operator to m: it creates a new sequence s by concatenating a set of input sequences followed by the method call m. Such input sequences are randomly extracted by $\mathcal{S}$, *i.e.,* the set of the valid sequences that do not violate any contract. Once a new sequence s is built, it is executed to assure that it is both non-redundant and valid. If s is valid, the algorithm adds it to $\mathcal{S}$. In some particular cases multiple calls to m are necessary to reach a desired object state. For such a reason, Randoop implements a repetition mechanism: with a given probability, instead of appending a single call m to create a new sequence, it appends M calls with a maximum value set by default to 100. At the end of the search —*i.e.,* when the given time limit is over— Randoop creates the regressions test from $\mathcal{S}$ while the non-valid sequences are used to generate error-revealing tests. Finally, Randoop places the assertions by iterating over each statement of the generated valid sequences $\mathcal{S}$: it firstly looks at the return values of each non-void method; then, it places the corresponding assertion. In particular cases, *e.g.,* for strings that represent raw object references, the assertion is not inserted.

JTExpert. As explained, traditional random testing approaches generate candidate solutions that run against the *class under test* (CUT), resulting in an achieved coverage. On the contrary, JTExpert [9] performs an informed random search in the sense that it targets only the uncovered branches. At first, it performs static analysis to identify i) the ways the class under test might be instantiated, ii) the methods that change the state of the CUT, called *state-modifier*, and iii) the methods that reach a given target, called *target-viewfinder*. A method sequence is composed of a set of *state-modifier* methods and a *target-viewfinder* method. The creation of the method sequence works as follow. A number of n *state-modifier* methods is selected and added to the sequence. n is calculated accordingly to domain vector for the potential test candidates. Such a vector is the sum of the CUT-instantiators, the possible *state-modifier* methods and the set of the *target-viewfinder* methods. To conclude the sequence, JTExpert always puts a *target-viewfinder* method call at the bottom of the sequence. It is worth to note that only the *target-viewfinder* methods that can hit a previously uncovered target are considered. To actually write the test data, JTExpert randomly selects one of the uncovered branches and then generates a method sequence for such a branch relying on the aforementioned approach. The sequence is then executed and evaluated: in case it hits uncovered branches, the sequence it is added to the final suite and the branches are marked as covered; in the opposite case, the test is discarded. At the end of the generation process, JTExpert places the assertions relying on an approach similar to the one detailed for Randoop.

2.2.2. *Search-Based Test Case Generation*

Besides random approaches, search-based software testing approaches have also been developed [7, 24, 79–81]. The resolution of an optimization problem using a search algorithm requires the definition of the solution representation and the fitness function. In the context of test case generation for object oriented programming, a solution is represented by a not fixed sequence of constructor and method invocations, including parameter values [75]. The fitness function is a combination of two measures: *approach level* [80] and *branch distance* [79]. The *approach level* represents how far is the execution path of a given test case from covering the target branch, while the *branch distance* represents how far is the input data from changing the boolean value of the condition of the decision node nearest to the target branch. As the branch distance value could be arbitrarily greater than the approach level, it is common to normalize the value of the branch distance [82]. The approach implemented in the eToc tool [75] selects the branches to cover incrementally. In details, it (i) enumerates all targets (*e.g.,* branches); (ii) performs a search, for each target, until all targets are covered or the total search budget is consumed; (iii) combines all generated test cases in a single test suite. This can lead to an important limitations. In particular, it (i) is sensible to branches that are infeasible or difficult to cover and (ii) it is not able to share potentially useful information across individual searches. To deal with such limitations, Fraser and Arcuri proposed the *whole test suite generation* approach [24], implemented in the EvoSuite tool [5]. This approach evolves testing goals simultaneously. A candidate solution is represented as a test suite and the fitness function is represented by the sum of all branch distances and approach levels of all the branches of the program under test. Nonetheless the whole suite approach proposed by Fraser and Arcuri [24] has a drawback: it tends to reward the whole coverage more than the coverage of single branches [7]. Thus, in some cases, trivial branches are preferred to branches that are harder to cover, affecting the overall coverage. To mitigate such a problem, Panichella *et al.* [7] formulate the test data generation problem as a many-objective problem. In particular, the authors consider the branch distance and the approach level of each branch as a specific fitness function. In this reformulation, a test case is considered as a candidate solution, while the fitness function is evaluated according to all branches at the same time. Since the number of fitness functions could be very high, the authors introduced a novel many-objective GA, named *MOSA* (Many-Objective Sorting Algorithm) and integrated the new approach in EvoSuite. Panichella *et al.* [7] further refined MOSA by presenting DynaMOSA. Such a variant focuses the search on a subset of uncovered targets computed relying on the *control dependency graph* of the CUT, to discern the targets free of control dependencies. At the beginning of the search, DynaMOSA considers as objectives only the targets that are free of control dependencies. The objectives vector is thus updated as soon as new targets are covered. Panichella *et al.* [83] recently introduced a multi-criteria variant of DynaMOSA that considers heterogeneous coverage targets simultaneously.

EvoSuite. While implementing most of the previously described approaches, the *whole test suite generation* approach is the one used by default by EvoSuite. It starts randomly creating an initial population of test suites having a maximum size M specified in input (50 by default). While doing that, EvoSuite aims at keeping the size of each test case constrained by an upper bound. This is done because due to the absence of automated oracle, software testers have to manually check the outputs and modify the assert statements. For the same reason, EvoSuite employs a *Bloat Control* mechanism that tends to prefer shorter test cases over long ones. Once generated the initial population, the evolutionary search starts evolving it by performing crossover and mutation operations. In the former, a new suite (an *offspring*) is generated by splitting and combining part of the test cases from two selected parents. On the other contrary, mutation operators randomly change statements of a test at a finer-grained level. Given that, it is worth noting that the method sequences forming the final tests do not change after the random generation performed to initialize the population. By default, the test cases evolved by EvoSuite do not contain assertions. However, it is possible to generate them at the bottom of the generation process applying mutation testing [5]. To this end, EvoSuite generates at first a large number of assertions that are subsequently filtered by their ability to detect mutants: only the minimum set of assertions able to detect all the mutants generated for the CUT is kept in the generated test cases. Such a mechanism is able to reduce the number of redundant assertions.

2.2.3. *Test Case Generation and Code Quality*

While most of the approaches described so far only considers coverage as the primary goal to achieve, some consider also the quality of generated test code as an objective to maximize. Afshan *et al.* [84] noticed that a critical goal to achieve when generating test cases is code readability. For this reason, they proposed the use of natural language models for the generation of tests having readable string input. Subsequently, Daka *et al.* [85] defined a post-processing technique able to optimize readability by mutating generated tests leveraging a domain-specific model of unit test readability based on human judgement. Finally, Palomba *et al.* [15] incorporated cohesion and coupling metrics into the process of test case generation of MOSA with the goal of producing more maintainable test cases. As a result, they showed that the test cases generated by the proposed approach are not only more cohesive and less coupled, but also able to increase the branch coverage and shorter tests than test cases generated by the standard MOSA algorithm. Our work is complementary those described above, since it has the goal to empirically understand the extent to which automated tools for test case generation tend to produce smelly code.

3. Empirical Study Definition and Design

In this section we report the planning of the study that we conducted to analyze the distribution of the test smells defined by Van Deursen *et al.* [27] in the context of automatically generated test cases.

3.1. *Goals and Research Questions*

The *goal* of the study was to empirically determine the extent to which unit tests generated by existing automated test case generation tools are affected by design problems, with the *purpose* of understanding (i) whether different families of algorithms are more/less prone to produce smelly unit tests thus possibly increasing maintenance costs and (ii) the likely factors influencing test smell diffuseness. We designed the following research questions:

- **RQ$_1$:** *What is the diffuseness of test smells in automatically generated test cases?*
- **RQ$_2$:** *Which test smells occur together?*
- **RQ$_3$:** *Is there a relationship between the presence of test smells and the characteristics of the production classes?*
- **RQ$_4$:** *Is there a relationship between the presence of test smells and the size of the test suite?*

The first two research questions aimed at quantifying the prominence of test smells in the tests automatically generated by existing tools, exploring their diffuseness in isolation (**RQ$_1$**) and their co-occurrence (**RQ$_2$**). Subsequently with **RQ$_3$** and **RQ$_4$** we analyzed possible factors influencing the presence of smells in test code, studying characteristics related to both production and test code.

3.2. Context Selection

The *context* of the study was composed of 100 classes contained in the publicly available `SF110` dataset [26]. This corpus is composed of Java classes belonging to a representative sample of 100 projects from the SourceForge.net repository, augmented with 10 of the most popular projects in the repository. Such benchmark has been widely used to assess test case generations tools [7, 24, 86], but findings in the field [30] demonstrated that most of the classes of the dataset are trivial to cover (*e.g.,* they contain branchless methods that can be fully covered by a simple method call) and, therefore, are not suitable for studies aimed at understanding characteristics and performance of automated testing tools [7, 30]. Therefore, we selected the non-trivial classes starting from an entire initial set of 23,886 Java classes. To this end, we first computed the McCabe's cyclomatic complexity for each method of the dataset by using CKJM [87]. The McCabe's cyclomatic complexity is defined as the number of independent paths in the control flow graph and it is equal to the number of branches plus one. As expected we found that most of the classes are trivial (*i.e.,* the cyclomatic complexity is one) and they could be covered by a simple method call. Hence, we decided to consider only methods with at least two conditional statements (*i.e.,* cyclomatic complexity equal to five). Details about the considered classes, including their name and cyclomatic complexity, are available in our online appendix [88].

3.3. Extracting the Automatically Generated Test Classes

To test the behavior of different algorithms, we generated test classes by employing three publicly available tools in their default configurations, *i.e.,* RANDOOP 3.1.5 [29], JTEXPERT 1.4 [9], and EVOSUITE 1.0.5 [89]. As detailed in Section 2, RANDOOP exploits a feedback-directed random testing, JTEXPERT relies on guided random testing, and EVOSUITE is based on evolutionary algorithms. Note that the use of default values does not impact the performance of automated test case generation tools, as shown by Arcuri and Fraser [90]. To address the intrinsic non-deterministic nature of the tools, we generate each test suite ten times for each class under test. This led to the definition of 1,000 test classes for each experimented tool (10 runs * 100 subjects * 3 tools = 3,000 test classes). It is important to note that the link between a generated test and a production class is intrinsically given by the generation process itself. Indeed, all the test case generation tools work as follow: the user has to specify a given class under test (CUT) and, afterward, the tool runs. At the end of the search process, the tool will generate a correspondent unit test, linked to the given CUT.

3.4. Extracting Test Smells

In our study, we focused on the seven test smell types introduced in Section 2, *i.e., Mystery Guest*, *Resource Optimism*, *Eager Test*, *Assertion Roulette*, *Indirect Testing*, *For Testers Only*, *Sensitive Equality*. Due to the high number of test classes to analyze (*i.e.,* 1,000 for each tool), we could not perform a manual detection. Thus, we relied on an automated test smell detector previously implemented and empirically evaluated by Bavota *et al.* [28]. Unlike other existing detection tools [68, 70, 71], the selected detector is able to identify all the test smells considered in this study by applying a heuristic metric-based technique that overestimates the presence of test design flaws in order to detect all the instances (100% of recall), having an average precision of 88%. Table 1 reports the set of rules used by the tool to detect smelly tests. While most of the detection rules applied by the detector are rather straightforward, it is worth discussing the one related to the *Assertion Roulette*. Indeed, one can think that the explanatory power of an assert statement could be subjectively assessed, *e.g.,* a developer may easily understand an assertion because s/he an expert of the source code under test. Nevertheless, this test smell arises when the cause leading an assertion to fail is not *explicitly* stated within the assert statement. In particular, JUnit developers can include, as first parameter of any assert statement, a `String` message reporting the motivation behind a failure caused by that assertion. The *Assertion Roulette* smell is detected if there are more than one assert statement where such a first parameter is not set. Thus, the associated detection rule simply counts the number of assertions in a test method and, for each of them, verifies whether a message is reported as first parameter. Finally, the detector marks a test method as affected by the smell if the `String` message is not set for more than one assertions. Note that the heuristic implemented by Bavota *et al.* [28] detects the assert statements by looking at the method

Table 1: Rules used to detect candidate test smells.

Name	Abbr.	Description	Precision
Mystery Guest	MG	JUnit classes that use an external resource (*e.g.,* a file or database).	100%
Resource Optimism	RO	JUnit classes that use an external resource that is not present on the disk.	65%
Eager Test	EG	JUnit classes having at least one method that uses more than one method of the tested class.	69%
Assertion Roulette	AR	JUnit classes containing at least one method having more than one assertion statement, and having at least one assertion statement without an explicit explanation.	100%
Indirect Testing	IT	JUnit classes invoking, besides methods of the tested class, methods of other classes in the production code.	68%
For Testers Only	FTO	Classes in the production code having structural relationship (*e.g.,* method invocations, inheritance) with only JUnit classes.	100%
Sensitive Equality	SE	JUnit classes having at least one assert statement invoking a `toString` method.	100%

calls of the `org.junit.Assert` class, *e.g.,* `assertEquals` or `fail`.

Despite the accuracy of the detector was already assessed by the original authors, it might still be possible that its performance in our context were lower. To make sure that the detector was really suitable for our purposes, we re-evaluated its precision[2] on a statistically significant sample of 341 JUnit test classes marked as smelly by the tool across the $3,000$ test classes generated by the three investigated automated test case generation tools over the different runs. Such a (stratified) sample is deemed to be statistically significant for a 95% confidence level and ±5% confidence interval [91]. Specifically, two authors of this paper first performed the validation independently: they were provided with (i) the source code of the considered tests and corresponding production classes and (ii) a spreadsheet file reporting the list of all the tests to classify. The task was to assign a truth value in the set {`true`, `false`} to each test of the list: the inspector assigned the value `true` when a test was actually affected by a test smell, `false` otherwise. Once the inspectors had completed this task, the produced outcomes were compared, and the inspectors discussed the differences, namely test smell instances marked as such by one inspector, but not by the other. All those tests positively classified as smelly by both the inspectors were considered as actual smells. Regarding the other instances, the inspectors opened a discussion in order to solve the disagreement and took a joint decision; such a discussion took three hours in total, concerned 27 tests (11 *Eager Test* and 16 *Indirect Testing* candidate instances), and was based on the definition of the smells as well as the argumentation of the inspectors. To measure the level of agreement between the two inspectors, we computed the Jaccard similarity coefficient [92], *i.e.,* number of test smell instances identified by both the inspectors over the union of all the instances identified by them. The overall agreement between the two inspectors before the discussion was 92%. As a result of the manual validation, we found that the precision of the tool is 75% overall. The last column of Table 1 shows the precision achieved by the tool on the individual test smells considered: from the table, we could observe that it has 100% precision when considering four of the seven smells, while for *Resource Optimism*, *Eager Test*, and *Indirect Testing* the precision ranges between 65% and 69%. In these cases, the lower precision of the detector is due to the intrinsic complexity of the detection mechanisms related to these smells: as an example, the detection of *Assertion Roulette* instances is much easier and precise than *Eager Test* because the former can be simply identified by considering the structural composition of an `assert` statement, while the latter requires the analysis of the behavior of a test, namely the identification of its multiple targets. Nevertheless, our validation of the statistically significant sample of instances given by the tool allowed us to claim that it is decently accurate in all cases to allow our experiment.

[2]Note that the recall could not be evaluated because of the lack of a comprehensive oracle of test smells.

3.5. Data Analysis and Metrics

Once detected the test smells affecting each of the $3,000$ JUnit test classes considered, we answered $\mathbf{RQ}_1$ by verifying the distribution of test smells in the analyzed classes. In other words, we measured how many times a certain test smell type was found over the classes automatically generated by each testing tool. We also computed the relative diffuseness, *i.e.,* the number of smelly JUnit classes over the total number of classes generated.

We considered the JUnit classes generated by the testing tools in different runs as independent, *i.e.,* the ten JUnit classes—produced in the ten execution runs by the testing tools—which test each production class in the dataset were considered as ten different instances with no relation with each other. This choice was done to balance the non-determinism of the generation algorithms: indeed, as explained in Section 3.3 the selected tools may behave in different ways during each run (*e.g.,* generating a test smell in one execution and producing a clean class in the subsequent run). For this reason, we preferred to consider the general behavior of the tools, focusing on a large-scale analysis done with as many test classes as possible, rather than focusing on the behavior of the tools with respect to single test classes.

As for $\mathbf{RQ}_2$, we investigated how often the presence of a test smell in a JUnit class implies the presence of another test smell. To this aim, we generated a database of transactions composed of the test smell instances found in each class considered. Such a database was then analyzed by means of Association Rule Mining [93]: this unsupervised learning technique is used to identify local patterns highlighting attribute value conditions that occur together in a given dataset. More formally, let $I = \{i_1, \ldots, i_n\}$ be a set of n binary attributes called *items* and indicating the presence of a certain property in the element under consideration, and let $T = \{t_1, \ldots, t_m\}$ a set of m *transactions* indicating the set of all the elements analyzed, an association rule is defined as an implication of the form $X \Rightarrow Y$, where $X, Y \subseteq I$ and $X \cap Y = \emptyset$. In our work, the set T is composed by all the test classes generated by the experimented automated test case generation tools, while each item in the set I indicates the presence of a given test smell in that test class. Therefore, an association rule $TS_{left} \Rightarrow TS_{right}$, between two disjoint sets of test smells implies that, if a JUnit test class is affected by each $ts_i \in TS_{left}$, then the same class should be affected by each $ts_j \in TS_{right}$. The validity and strength of an association rule is determined by its support and confidence [93]:

$$Support = \frac{|TS_{left} \cup TS_{right}|}{T}$$

$$Confidence = \frac{|TS_{left} \cup TS_{right}|}{|TS_{left}|}$$

where T is the total number of classes considered. To compute the association rules, we used the statistical software R and the package `arules` which implements the well-known *Apriori* algorithm [93]. In Section 4 we discuss the top-5 rules output for each analyzed testing tool. Moreover, to complement the quantitative findings we also performed a fine-grained qualitative analysis aimed at understanding the causes making a certain generation algorithm more/less prone to produce test smells: we manually analyzed the test code generated by the different tools with the aim of relating the peculiarities of the exploited algorithms and the presence of test smells. To this aim, two authors of this paper independently reviewed all the test classes produced by the tools looking for the root causes which make the test code smelly.

To answer **RQ**$_3$ and understand the relationship between test code and production class characteristics, we computed the Kendall's rank correlation (Kendall's τ) [94] between the distribution of the test smells and the values of four different Object-Oriented metrics able to characterize a class under different perspectives (*i.e.,* Lines of Code (LOC), Lack of Cohesion of Methods (LCOM), Coupling Between Object Classes (CBO), Response for a Class (RFC), and Depth of Inheritance Tree (DIT)) [31] — all the code quality metrics were computed using the publicly available CKJM tool[3] developed by Spinellis [87].

In particular, Kendall's τ is a measure of correlation between two variables X and Y defined in [-1; +1], where

Table 2: Relative diffuseness for the seven considered test-smells over the tests generated by EvoSuite, Randoop, and JTExpert. We report the values per class and per method

Smells	Randoop		JTExpert		EvoSuite	
	Method	Class	Method	Class	Method	Class
Assertion Roulette	0.812	0.976	0.880	0.737	0.744	0.736
Eager Test	0.533	0.487	0.743	0.622	0.623	0.567
Mystery Guest	0.063	0.104	0.119	0.150	0.069	0.114
Sensitive Equality	-	-	0.576	0.667	0.016	0.077
Resource Optimism	-	-	-	-	0.006	0.031
For Testers Only	-	0.012	-	-	-	0.011
Indirect Testing	-	-	-	-	-	-

+1 represents a perfect positive linear relationship, -1 represents a perfect negative linear relationship, and values in between indicate the degree of linear dependence between X and Y. To interpret the results, we used established guidelines [94] reporting that there is no correlation when $0 \leq \rho < 0.1$, small correlation when $0.1 \leq \rho < 0.3$, medium correlation when $0.3 \leq \rho < 0.5$, and strong correlation when $0.5 \leq \rho \leq 1$. Similar intervals also work in cases of negative correlations.

In **RQ**$_4$ we verified whether the presence of test smells can depend on the size of the test class generated. Similarly to **RQ**$_3$, we employed the Kendall's rank correlation [94] to measure the relation between the distribution of the test smells and the size of the corresponding JUnit test classes.

In addition to the above quantitative analyses, we conducted a qualitative investigation to assess which characteristics and peculiarities of the test case generation algorithms lead to the introduction of test smells. More specifically, two authors of this paper jointly analyzed the source code of the three considered tools in order to understand how they (i) actually generate method sequences and (ii) add assertions to the test code. During this process, the two authors first cloned the projects from their respective repositories; then, they both analyzed the algorithm definitions, the generated test classes, and the source code when available (*i.e.,* for Randoop and EvoSuite). Through this analysis, the two authors could come up with qualitative insights that aim at explaining the reasons behind the results coming from the correlation analyses previously performed. In Section 4 we discuss the results of our study by reporting the quantitative findings first, followed by the qualitative observations made by the automatic test case generation tool inspectors.

4. Analysis of the Results

In this section we report and discuss the results of the study aimed at answering our research questions.

4.1. *RQ*$_1$*: The Diffuseness of Test Smells*

[3]The tool is available at: https://www.spinellis.gr/sw/ckjm/.

Our first research question aimed at studying how diffused test smells are in test code automatically generated by existing testing tools.

Results. Table 2 reports, for each automated testing tool considered, the percentage of smelly JUnit test classes and methods found when running the test smell detector over the generated tests. The first thing that leaps to the eye is the high diffuseness of test smells in automatically generated JUnit classes for all the experimented test-generation tools. On the one hand, this confirms the results previously achieved on EvoSuite [25], while on the other hand, it demonstrates that the problem of test smells is widely diffused also for other tools like Randoop and JTExpert. More specifically, almost 81% of the JUnit classes generated by EvoSuite can be considered *smelly* (*i.e.,* they contain at least one test smell instance); the percentage is even higher for the other tools, *i.e.,* 98% and 92% for Randoop and JTExpert, respectively. The results are very similar when lowering the granularity at method-level. Among the considered test smells, *Assertion Roulette* is the most diffused (*e.g.,* 81% and 98% of methods and classes generated by Randoop are affected by this smell, respectively), followed by *Eager Test* (*e.g.,* 73% of test methods generated by JTExpert are smelly).

Discussion. The first result of our study clearly points out that, despite different heuristics for test case generation are employed by the considered tools, the distribution of test smells is nearly the same (as reported in Table 2). This is due to multiple reasons. Let first focus on the most diffused smell, *i.e., Assertion Roulette.* According to the formal definition, this smell arises when a JUnit class contains at least one assertion statement without an explicit declaration (*i.e.,* no use of the optional first argument provided by the JUnit framework to add explanations). The main problem leading to the high diffuseness of *Assertion Roulette* relates to the fact that none of the considered tools is able to generate those kind of comments for the placed assertions. For instance, Listing 2 shows an example of test case generated by Randoop: as it is possible to observe, the test case contains 12 different assertions without any explanation, either as a comment or as an optional message passed as first parameter to the assertion. This aspect, together with the poor readability of the test case, would make the understanding of the exercised behavior hard. We found this kind of issue for all the automated test code generators.

EvoSuite only excepts to this behavior when it fails a test verifying an exception. For the sake of comprehensibility, let consider the test shown in Listing 3. The test first initializes local objects of the the `BrowseNode` class and, then, exercises the code to check whether a `NullPointerException` is thrown when calling the method `getSubNode` with a string parameter referring to a non-existing node. When testing for this behavior, we noticed that EvoSuite uses the assertion `fail` (line 8 in

```java
  public void test009() throws Throwable {
    ...
    org.junit.Assert.assertTrue("'" + str3 + "' != '" + "null - null -- hi!" + "'", str3.equals("null - null -- hi!"));
    org.junit.Assert.assertNull(str4);
    org.junit.Assert.assertTrue("'" + str8 + "' != '" + "null - null -- hi!" + "'", str8.equals("null - null -- hi!"));
    org.junit.Assert.assertNotNull(arrayList21);
    org.junit.Assert.assertTrue("'" + str26 + "' != '" + "null - null -- hi!" + "'", str26.equals("null - null -- hi!"));
    org.junit.Assert.assertNull(str27);
    org.junit.Assert.assertTrue("'" + str31 + "' != '" + "null - null -- hi!" + "'", str31.equals("null - null -- hi!"));
    org.junit.Assert.assertTrue("'" + str37 + "' != '" + "null - null -- hi!" + "'", str37.equals("null - null -- hi!"));
    org.junit.Assert.assertNull(str38);
    org.junit.Assert.assertTrue("'" + str42 + "' != '" + "null - null -- hi!" + "'", str42.equals("null - null -- hi!"));
    org.junit.Assert.assertNotNull(arrayList52);
    org.junit.Assert.assertNotNull(arrayList56);
}
```

Listing 2: Test method automatically generated by Randoop suffering of Assertion Roulette

```java
1  public void test02()  throws Throwable  {
2  BrowseNode browseNode0 = new BrowseNode();
3  BrowseNode browseNode1 = new BrowseNode();
4  browseNode1.addSubNode(browseNode0);
5  // Undeclared exception!
6  try {
7  browseNode1.getSubNode(" -- \n");
8  fail("Expecting exception: NullPointerException");
9  } catch(NullPointerException e) {
10 verifyException("net.kencochrane.a4j.beans.
     BrowseNode", e);
11 }
12 }
13
```

Listing 3: Test case generated by EvoSuite that expect a `NullPointerException`

Listing 3) by including a textual message that explain what the test is supposed to verify in production code. Thus, in these cases the tool precludes the introduction of an *Assertion Roulette.*

From our analyses, we can therefore conclude that all the three tools tend to introduce an instance of this smell as soon as a new assertion is added in a JUnit class. To better understand the extent of the issue, we also computed the assertion density [95], *i.e.,* the number of assertions per test case, of the tests given as output by the three tools. We found that the tests generated by EvoSuite have an assertion density of 2, while Randoop and JTExpert generate on average suites with 14 and 5 assertions per test, respectively. Those results are due to the mechanisms used to place the assertions (described in Section 2): as such, while EvoSuite reduces the set of assertions generated via mutation testing, the other tools have very little control over the number of placed assertions and possibly increase the presence of *Assertion Roulette* instances.

Some other insights can be discussed when considering the second most diffused smell, *i.e., Eager Test*, that arises whether a test case invokes more than one method production class. This smell is naturally related to the method sequences that form the final test cases. More specifically, with our qualitative investigation into the algorithms exploited by the considered tools we could conclude that the presence of *Eager Test* instances is caused by the intrinsic randomness in generating the method sequences forming the final test cases. As detailed in Section 2, EvoSuite randomly generates the initial population (*i.e.,* the initial set of tests) by putting together random method sequences that likely refer to different production methods. Therefore, the tests are scented since the beginning. Indeed, crossover and mutation operations performed though their evolution do not change the structure of the tests, *i.e.,* these operators do not act on method sequences, thus not removing the initial smells nor introducing new ones.

A similar phenomenon occurs with Randoop and JTExpert. The former extends the sequences either by appending a method call at the end of the sequence or combining two existing ones: it is easy to observe how such a mechanism leads to eager sequences. JTExpert, instead, builds the method sequences by combining *state-modifier* methods with a *target-viewfinder* one. While the former are often needed to expose faulty object states, the tools has little control over the number of *state-modifier* methods that form the sequence. This mechanism may induce the introduction of sequences referring to different production methods, thus introducing a smell.

A final discussion point regards a peculiarity of JTExpert. While the distribution of test smells is generally similar in all the three considered tools, the diffuseness of *Sensitive Equality* instances in JTExpert is much higher than the others. To recall, *Sensitive Equality* happens when an assertion contains an equality check though the use of a `toString` method. Listing 4 shows a related example of a test case generated by JTExpert for the class `VariableLabelTableModel`. As reported, the tool tends to generate statements where the asserts are composed of calls to the `toString` Java API in case of checks involving `String` objects, leading to the introduction of *Sensitive Equality* instances. This is true for most of the `String` equality controls in our dataset (94% of them are done by means of the `toString` function).

Summary of RQ$_1$. Test smells are widely diffused in test cases automatically generated by all the experimented tools: 81%, 92%, and 98% of the JUnit classes generated by EvoSuite, JTExpert, and Randoop, respectively, can be considered smelly. Among the studied smells, we found that *Assertion Roulette* and *Eager Test* are the most spread within our dataset.

4.2. ***RQ$_2$****: On the Co-Occurrence of Test Smells*

In the second research question, we investigated to what extend test smells co-occur in automatically generated tests.

```java
@Test public void TestCase9() throws Throwable {
VariableLabelTableModel
   clsUTVariableLabelTableModel=null;
clsUTVariableLabelTableModel=new
   VariableLabelTableModel(
   clsUTVariableLabelTableModelP1P1,
   clsUTVariableLabelTableModelP1P2);
int clsUTVariableLabelTableModelP2R=0;
clsUTVariableLabelTableModelP2R=
   clsUTVariableLabelTableModel.getColumnCount();
assertEquals(3,clsUTVariableLabelTableModelP2R);
ArrayList clsUTVariableLabelTableModelP3R=null;
clsUTVariableLabelTableModelP3R=
   clsUTVariableLabelTableModel.getData();
Object clsUTVariableLabelTableModelP3RP0R=null;
clsUTVariableLabelTableModelP3RP0R=
   clsUTVariableLabelTableModelP3R.clone();
assertEquals("[]",clsUTVariableLabelTableModelP3RP0
   R.toString());
assertEquals(1,clsUTVariableLabelTableModelP3RP0R.
   hashCode());
assertEquals("[]",clsUTVariableLabelTableModelP3R.
   toString());
assertEquals(true,clsUTVariableLabelTableModelP3R.
   isEmpty());
String clsUTVariableLabelTableModelP4R=null;
...
}

```

Listing 4: Example of Test Case suffering of *Sensitive Equality* generated by JTExpert.

Results. Table 3 shows, for each tool, the five most important association rules detected when considering both the test class granularity (upper part of the table) and test method granularity (lower part of the table). It is easy to note that $ET \rightarrow AR$ is the rule that most frequently occurs with a high support and confidence. Indeed, in case of EvoSuite and Randoop, it represents the rule with the highest number of occurrences (492 and 399, respectively), while for JTExpert 510 occurrences were detected (*i.e.,* for half of the generated test suites). On the one hand, this rule might be a reflection of the high diffuseness of those two smells in the analyzed dataset: it may be that the massive presence of an *Eager Test* accidentally coincides with a large amount of co-occurrences of *Assertion Roulette* instances. On the other hand, however, we can also explain such a rule with the following reasoning: with the aim of maximizing the overall branch coverage, the investigated tools tend to exercise more production code as possible. Often, this would eventually break the Single-Condition Tests principle [67], *i.e.,* the best practice of having a single behavior exercised by each test and lead to the introduction of an *Eager Test* instance. Moreover, different behaviors imply more production code executed by a test case that could result in more assertions generated via mutation testing: as a result, some *Assertion Roulette* instances might be generated in the same place.

Our findings also revealed that *Mystery Guest* instances often co-occur with both *Assertion Roulette* and *Eager Test* in the cases of EvoSuite and Randoop. The relation that *Sensitive Equality* has with *Assertion Roulette* when considering EvoSuite and JTExpert is

Table 3: Top 5 rules detected via Rule-Association Mining for each of the tool used. We report the values both at test suite and test case granularity

Test Class Granularity											
RANDOOP				JTEXPERT				EVOSUITE			
rule	support	confidence	count	rule	support	confidence	count	rule	support	confidence	count
$ET \rightarrow AR$	0.48	0.98	399	$SE \rightarrow AR$	0.86	0.99	533	$ET \rightarrow AR$	0.53	0.91	492
$MG \rightarrow AR$	0.10	0.92	80	$ET \rightarrow AR$	0.86	0.99	510	$MG \rightarrow AR$	0.10	0.81	89
$MG \rightarrow ET$	0.04	0.43	37	$ET \rightarrow SE$	0.79	0.96	491	$MG \rightarrow ET$	0.08	0.73	79
$ET, MG \rightarrow AR$	0.04	0.81	30	$ET, SE \rightarrow AR$	0.79	0.99	490	$SE \rightarrow AR$	0.08	0.98	72
$AR, MG \rightarrow ET$	0.04	0.38	30	$AR, ET \rightarrow SE$	0.79	0.96	490	$SE \rightarrow ET$	0.08	0.96	70
Test Case Granularity											
$ET \rightarrow AR$	0.72	0.99	548,689	$ET \rightarrow AR$	0.92	0.99	4,628	$ET \rightarrow AR$	0.28	0.65	5,856
$MG \rightarrow AR$	0.12	0.95	92,879	$SE \rightarrow AR$	0.91	0.99	4,599	$MG \rightarrow AR$	0.20	0.42	416
$MG \rightarrow ET$	0.07	0.59	57,822	$SE \rightarrow ET$	0.88	0.96	4,473	$MG \rightarrow ET$	0.01	0.33	327
$ET, MG \rightarrow AR$	0.07	0.95	55,396	$ET, SE \rightarrow AR$	0.88	0.99	4,452	$ET, MG \rightarrow AR$	0.01	0.55	231
$AR, MG \rightarrow ET$	0.07	0.59	55,396	$AR, SE \rightarrow ET$	0.88	0.96	4,452	$AR, ET \rightarrow MG$	0.01	0.04	231

pretty expected, as the former smell naturally appears when the latter arises. The same tools also shared the relation between *Sensitive Equality* and *Eager Test* ($SE \rightarrow ET$ in the case of EVOSUITE, $ET \rightarrow SE$ in the case of JTEXPERT). Finally, it is worth remarking that also other composite relations appeared for RANDOOP and JTEXPERT (*e.g.,* $ET, MG \rightarrow AR$). The test smells involved in these rules are exactly the same as those discussed so far and the derived rules do not provide further insights on the co-occurrences between them.

Discussion. Through the analysis of the mechanisms used by the considered tools for the generation of method sequences and assertions (explained in Section 2), we could practically explain the most recurring co-occurrences observed in our study. Let first considering the most frequent co-occurrence, *i.e.,* $ET \rightarrow AR$: this relation is to be considered intrinsic and natural when considering the way assertions are placed by the considered tools. Indeed, the presence of an *Eager Test* implies a larger number of calls that aim at exercising the behavior of different production methods; this may lead a test to exercise methods having returning values, which are the elements triggering the introduction of assertions. As an example, RANDOOP iterates over all the statements that form a sequence and introduces an assertion for each non-void returning value. Similarly, the more the called methods the higher the number of mutants generated by EVOSUITE; as a consequence, the higher the number of possible assertions.

When considering JTEXPERT, we observed two co-occurrences that involve the *Sensitive Equality* smell and that do not appear in the cases of the other testing tools, *i.e.,* $SE \rightarrow AR$ and $ET \rightarrow SE$. The former can be explained as follow: JTEXPERT mostly uses `toString` calls to check object states in the assertions, therefore increasing its chances to introduce a *Sensitive Equality* in combination with an *Assertion Roulette*. For instance, Listing 5 shows a test case generated by JTEXPERT, in which the mechanism presented above is implemented when assessing the equality of a string object. As for the second

```
@Test public void TestCase0() throws Throwable {
CalEventModelImpl clsUTCalEventModelImpl=null;
clsUTCalEventModelImpl=new CalEventImpl();
CacheModel clsUTCalEventModelImplP2R=null;
clsUTCalEventModelImplP2R=clsUTCalEventModelImpl.
      toCacheModel();
Object clsUTCalEventModelImplP2RP0R=null;
clsUTCalEventModelImplP2RP0R=clsUTCalEventModelImplP2
      R.toEntityModel();
assertEquals("{uuid=, eventId=0, groupId=0, companyId
      =0, userId=0, userName=, createDate=null,
      modifiedDate=null, title=, description=, location
      =, startDate=null, endDate=null, durationHour=0,
      durationMinute=0, allDay=false, timeZoneSensitive
      =false, type=, repeating=false, recurrence=,
      remindBy=0, firstReminder=0, secondReminder=0}",
      clsUTCalEventModelImplP2RP0R.toString());
...
```

Listing 5: Test generated by JTExpert using `toString()` to check an assertion

co-occurrence, we can argue that *Eager Test* and *Sensitive Equality* often appear together as a reflection of the relation that these two smells have with *Assertion Roulette*: indeed, as we have seen, more method calls imply more assertions, that in the case of JTEXPERT are often implemented with a `toString` call.

A final discussion point regards the rule $MG \rightarrow AR$, which frequently appears in both EVOSUITE and RANDOOP. In this regard, we did not find any specific causality explaining this relation: we could just conclude that this is due to the fact that test cases relying on external resources often contain assertions, thus leading to the co-occurrence between *Mystery Guest* and *Assertion Roulette* instances.

Summary of RQ$_2$. The presence of *Eager Test* often implies an *Assertion Roulette*: this is true for all the considered tools with a high support and confidence. The relation between *Sensitive Equality* and *Assertion Roulette* is naturally confirmed, while other smells tend to appear less together or their relation is not causal.

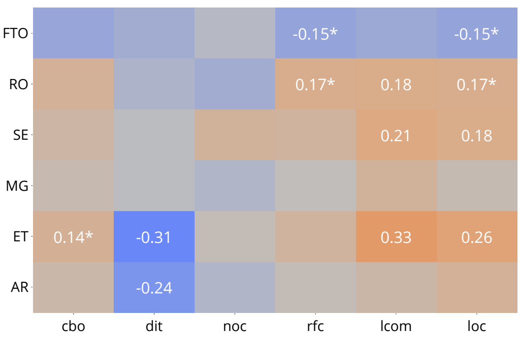

Figure 1: Kendall's correlations between the smells for the test suites generated by EvoSuite and the structure properties of source code.

*4.3. **RQ**$_3$: The Relationship between Test Smells and Structural Properties of Source Code*

To answer **RQ**$_3$, we correlated the structural metrics of the production code with the occurrences of the detected test smells. Note that we calculated the results considering the testing tools independently.

Results. Our analyses did not highlight relevant insights in the cases of Randoop and JTExpert: indeed, most of the correlations between code metrics and test smells are either *negligible* or not statistically significant, *i.e.,* $p \geq 0.10$. The only exception is related to the correlation between DIT and *Eager Test* for the suites generated by Randoop; however we did not find any relevant reason to support the causality of such a correlation. Thus, we conclude that approaches implementing random algorithms—that do not consider the nature of production code when generating tests—do not share relevant relations with the exercised unit, and thus they are not influenced by code metrics. For sake of readability, we included a detailed report of our analyses for these two tools in our online appendix [88], while in this section we report the most interesting results related to EvoSuite.

Figure 1 shows a heap map reporting the Kendall's τ values achieved when studying the relationship between the presence of test smells in tests generated by EvoSuite and the code quality metrics of the production classes. The figure only shows the statistically relevant τ, *i.e.,* the ones with $p \leq 0.05$. The values marked with * indicate a p-value between 0.05 and 0.10. As it is possible to see, different test smell types have noticeable correlation values with different characteristics of the production code. For instance, the number of lines of code of the production classes (LOC) seems to influence the presence of several types of smells: *Eager Test* ($\tau = 0.26$), *Sensitive Equality* ($\tau = 0.18$), and *Resource Optimism* ($\tau = 0.17$) have a positive correlation. On the contrary, *For Tester Only* shows a negative correlation, *i.e.,* $\tau = -0.15$. We observe similar correlations for the LCOM metrics: *Eager Tests* has a *medium* correlation ($\tau = 0.33$), while *Sensitive Equality* and *Resource Optimism* a small one.

While LCOM and LOC are the most positively correlated metrics, we report negative correlations (*i.e.,* the percentage of smells decreases when the metric increases) for the depth of inheritance tree (DIT) metric. We found a *medium* correlation for *Eager Test* ($\tau = -0.31$) and a small one ($\tau = -0.24$) for *Assertion Roulette*. This result seems to support what has been previously shown by Nogueira *et al.* [96] on the ability of inheritance metrics to provide information on the quality of test cases.

Discussion. The findings on correlation highlight some potential relationships between production and test code. During our manual analysis, we tried to further elaborate on such relations to discover the causes behind them. As a result, we first found that the correlation between *Eager Test* and lines of production code is simply due to high density of complex methods in production classes. Moreover, large classes in the dataset also contains large methods with non-cohesive responsibilities which lead the testing tools to produce test methods that check more than one method of the class to be tested. This result corroborates previous findings reported by Tufano *et al.* [73], who found that code affected by code smells characterizing complex and long code (*i.e.,* the *Spaghetti Code* [66]) tend to be related to the emergence of *Eager Test* instances. This is also the motivation for the strong correlation that we found between this test smell and the LCOM metric. Indeed, such metric possibly indicates that a class does not follow a responsibility-driven design [97], increasing the probability that a test method is enforced to test more production methods at the same time.

Size and cohesion of production code also influence the presence of other test smells such as *Assertion Roulette* and *Sensitive Equality*. Indeed, the more the code to be tested, the higher the chance of adding assertions in test code that verify the status of the production code. At the same time, the higher the number of assertions the higher the likelihood to test string objects and add *Sensitive Equality* instances. So, in this latter case, the correlation seems to be a reflection of the intrinsic relation that this smell has with *Assertion Roulette*.

In summary, after this analysis we can claim that the quality of production code may influence the presence of smells in test code automatically generated by EvoSuite. Thus, developers can improve the ability of the tool to generate qualitative tests by (i) keeping some production code quality indicators under control and (ii) apply refactoring solutions that simplify the structure of production code, thus making EvoSuite more able to produce high-quality tests. Unfortunately, this rule does not hold for Randoop and JTExpert, both implementing random meta-heuristics that have no specific relations with production code. From a practical perspective, this implies that the quality of tests generated by those tools cannot be monitored by developers; researchers are instead called to define more effective solutions enabling a quality-aware random generation of test cases.

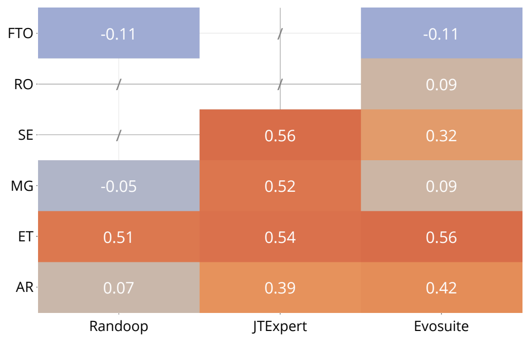

Figure 2: Kendall's correlations between the test smells and the test-suite sizes generated by the three analyzed tools

Summary of RQ$_3$. The presence of smells in tools implementing random algorithms is not influenced by the quality of production code. On the contrary, we discovered some correlations between code metrics and test smells present in tests generated by EvoSuite that might be useful for developers to keep the quality of production code under control, immediately providing benefits for the generation of good tests.

4.4. *RQ$_4$: The Relationship between Test Smells and Size of the Test Suites*

Our last research question aimed at targeting the relation between test smells and the size of the test suites generated by the investigated automatic tools.

Results. Figure 2 reports the Kendall's values achieved analyzing the relationships between the presence of test smells and the size of the resulting test classes. The results are reported separately for each test data generator. A white box indicates that no correlation has been calculated since the corresponding smell has not been detected over the tests generated by a certain tool.

According to our results, we observe that the size of the generated test suite is generally moderately correlated with the presence of test smells. Indeed, for each of the three analyzed tools, we report a *strong* correlation between the test-suite size and *Eager Test*, meaning that the larger the generated tests the higher the probability to include instances of this smell. It is worth remarking that such a smell represents one of the most harmful ones for the maintainability of both test and production code [22], and having "cheap" solutions to reduce their introduction might provide an important benefit for improving the overall reliability of software systems.

For Randoop this is the unique noticeable correlation: this might due to the fact that the Randoop's default configuration gives as output test suites with a fixed amount of test cases. Indeed, the size of the generated tests does not vary that much amongst the subjects of our study. Looking at JTExpert, we found *strong* correlations also for *Sensitive Equality* and *Mystery Guest*, while for *Assertion Roulette* we observe a *medium* correlation with the test suite size. Finally, when analyzing EvoSuite, we observed *medium* correlations for *Assertion Roulette* and *Sensitive Equality*, confirming the finding of our previous study [25].

Discussion. On the basis of the correlation and manual analyses done with respect to **RQ$_4$**, we could provide two main observations. In the first place, not all test smells have a correlation with test size: this means that some of them, *e.g., Mystery Guest*, are independent from the mechanisms used by automatic testing tools to keep test suite size under control. As such, our findings inform researchers and tool vendors on the need for different mechanisms to deal with these types of test smells: for example, post-processing refactoring activities should be preferred to inner-working instruments that possibly interfere with the test case generation process. At the same time, there are specific test smells, like *Eager Test* and *Assertion Roulette*, that are influenced by test size: to some extent, this may be expected because their appearance is naturally related to the number of lines of code in a test suite. For example, the higher the size of a test class the higher the likelihood of the presence of tests that exercise more production methods. The results of our study possibly suggest that the introduction of certain smells may be reduced or even precluded by keeping test suite size under control. To further investigate this aspect, we conducted an additional analysis by investigating the dataset provided by Palomba *et al.* [15]: it contains a set of test classes automatically generated by Q-Mosa, namely a quality-aware version of EvoSuite that takes into account cohesion and coupling metrics while optimizing for code coverage. The test classes output by Q-Mosa have been shown to have a statistically lower size than the standard version of EvoSuite: thus, we can use these classes as baseline to show the effect of keeping size under control on the presence of test smells. To this end, we ran our test smell detector on such classes in order to compare the test smell diffusion in the tests generated by Q-Mosa and the standard EvoSuite. As a result, we found that Q-Mosa is able to generate 57% less *Eager Test* and 33% less *Assertion Roulette* instances with respect to EvoSuite, while reaching a higher code coverage, as reported by Palomba *et al.* [15].

All in all, our analyses provide a first compelling evidence that keeping test suite size under control might substantially reduce certain types of test smells appearing in test code, without having any negative effect on the resulting code coverage. Of course, we are aware that further investigations into this aspect are desirable and part of our future research agenda.

Summary of RQ$_4$. Test suite size is often correlated to the presence of *Eager Test* and *Assertion Roulette*. Moreover, the number of instances for these two smells may be successfully reduced by keeping test size under control during the generation process, without damaging the resulting code coverage.

5. Further Discussion and Implications

The results of our study provide a number of insights that can be used by researchers and tool vendors to improve automatic test case generation tools and approaches with respect to the design quality of the produced tests.

Toward quality-aware testing tools. Our results revealed the high diffuseness of test smells in the code automatically generated by all the experimented testing tools. As shown by recent research in the field [19, 22], test smells—especially the ones occurring the most, *e.g., Eager Test* and *Assertion Roulette*—heavily impact test/production code quality and maintainability as well as test effectiveness; as a consequence, we argue that such testing tools, whose aim is to provide developers with additional tests that can possibly catch faults hard to identify manually, should carefully take into account quality-related aspects in the generation process to avoid the natural introduction of design defects that might threat the overall reliability of software systems. Some researchers have started working toward this direction by proposing cohesion and coupling metrics as secondary objectives to reach while optimizing for code coverage [15], however we argue that further research into this matter would be desirable. For instance, explicitly taking test smell-related information into account within the generation process (*e.g.,* by considering the number of test smells as secondary objective) may have a positive effect of the resulting design quality without necessarily impacting the overall effectiveness of tests. At the same time, the research community would benefit from an improved understanding of how lack of production code quality impacts test code, *e.g.,* whether test cases exercising complex/smelly production classes tend to become smelly as well. Also in this case, even if some preliminary studies are already available [72, 73], more research on the topic would provide to tool vendors further insights on how to improve software testing tools.

Documenting tests. The smell occurring the most is by far *Assertion Roulette*, which is naturally generated by currently available automated test case generation tools because of their inability to produce explanatory messages when adding assertions in test code. The basic issue of this smell is that it lowers the ability of developers to understand the actual reason behind a failure. On the basis of our findings, we envision two main implications for researchers and tool vendors. On the one hand, the number of *Assertion Roulette* instances may be reduced by keeping quality-related aspects into account during the generation process, as shown in our additional analysis involving the dataset made available by Palomba *et al.* [15]: this may possibly indicate that more focused tests, *i.e.,* test cases that target single production methods, have less chance to be affected by this smell. On the other hand, a mitigation of the problem can be provided by automated solutions to document test cases: for instance, a recent work [20] devised an approach to generate summaries for tests, showing that they help when developers fix bugs in production code. We argue that (i) quality-aware solutions and (ii) finer summarization techniques working at assertion-level might reduce or even eliminate the problems created by having *Assertion Roulette* instances in test code.

On the refactoring of test smells. Previous works have shown that code/test smell removal can eliminate some of the problems caused by those design issues [19, 51]. A natural implication of our study is a call for more research on refactoring and in particular test smells refactoring [98]. In this regard there are some key challenges to face. First, the refactoring approaches should not affect test case reliability and effectiveness. Second, it is not clear *when* refactoring should be performed. On the one hand, quality-aware test case generation approaches could integrate *on-the-fly* refactoring mechanisms that avoid test smell appearance in the first place; on the other hand, the definition of ad-hoc post-processing steps that remove test smells could represent another alternative to pursue.

Keeping test suite size under control. In our study, we found that test size has a detrimental effect on test quality. This practically means that testing tools monitoring test size while generating tests could reduce the overall number of test smells, especially those referring to *Eager Test* and *Assertion Roulette*. We argue that this aspect should be more carefully considered in order to produce more effective tools. In this sense, a key challenge is the development of novel algorithms able to effectively integrate such control mechanisms into the automated test case generation process. To support this hypothesis, we conducted an additional analysis to understand how the number of test smells can be reduced by keeping quality aspects under control. By detecting test smells in a set of test cases generated by Q-Mosa, a quality-aware test case generation tool, we discovered that it is possible to control test smell diffuseness while improving code coverage and reducing test code size as previously shown by Palomba *et al.* [15]. As such, the introduction of quality metrics as secondary objectives of currently available tools may represent a valuable way to control both size and presence of test smells. Based on our findings, the research community is called to further investigate and devise algorithms able to automatically generate shorter and/or less smelly test cases.

On production code quality. In the case of EvoSuite, we discovered that the quality of production code has a correlation with the amount of test smells generated by the tool. While this finding should be further corroborated by analyses aimed at better investigating causality, our results seem to delineate a trend for which developers might support testing tools by writing good production code. More specifically, our findings seem to support the existence of an interplay between source code quality and effectiveness, since it may be possible to improve test code generated automatically by taking production code quality under control. Our results may increase the practitioners' awareness in measuring production code quality and applying refactoring to reduce its complexity. Furthermore, we believe that researchers should further investigate to what extent automatically generated test code is sensible to variations in production code, *e.g.,* to what extent the effectiveness of generated test cases varies before and after production code refactoring. Finally, we observed that Randoop and JTExpert are not influenced by production code quality: this means that refactoring production code does not enable the generation of better test cases when these tools are adopted. For these tools novel solutions enabling a quality-aware random generation of test cases should be devised.

6. Threats to Validity

In this section we discuss the threats that might have affected the validity of our study.

6.1. Threats to construct validity

As for threats related to the relationship between theory and observation, the main threat in our study concerns the way test smells were detected in the considered projects. Given the amount of analyzed tests, we could not perform a manual detection, therefore, we relied on an automated solution. We employed the publicly available test smell detector originally developed by Bavota *et al.* [28], as it has been reported to be highly effective in the identification of instances of the seven test smells considered. Furthermore, we also conducted an additional analysis aimed at assessing its performance in our context, manually evaluating the precision of the tool on a statistically significant set of 341 test classes. The results achieved showed a precision of 75%, and thus we could consider the tool suitable for our study. It is important to note that we cannot speculate on the recall of the detector because of the lack of a comprehensive oracle for the considered set of classes. However, from our precision re-assessment, it seems that the performance reported by Bavota et al. [28] holds in our context. This makes us confident of the fact that it also holds for recall but, of course, we cannot exclude the presence of false negatives.

All the automated test case generation tools experimented were tested using the default configuration parameters to conduct a fair comparison. Moreover, we ran the tools ten times to address the non deterministic nature of the underlying algorithms.

6.2. Threats to internal validity

Threats to internal validity are related to factors, internal to our study, that could have influenced our findings. In this case, the main potential problem is related to cause-effect relationships between what we measured and what are the actual factors influencing the diffuseness of test smells (**RQ**$_3$ and **RQ**$_4$). As explained in Section 3, we computed a set of code metrics with the aim of understanding if there is a correlation between such metrics and the presence of test smells: we consciously performed a correlation analysis since our goal was to identify factors that might be possibly employed to avoid the generation of test smells during the process of automated test case generation. To deal with the randomness of the employed algorithms, we repeated the test cases generation phase for 10 times [99]. Such randomness might have influenced our analysis. Therefore, we computed the mean and the standard deviation of the test smell occurrences for each considered test smell over the different generations: we observed low deviations as an indication that the behavior of the tools do not differ much across different runs. For instance, we detected a mean of 22 instances of *Assertion Roulette* for the test suites generated by EvoSuite with a mean standard deviation over the different runs of 2.

6.3. Threats to conclusion validity

Threats to conclusion validity concern the relationship between experimentation and outcome. To investigate the diffuseness of test smells (**RQ**$_1$) in the test classes automatically generated by the testing tools, we verified the distribution of the smelly instances output by the detector. To assess test smell co-occurrences (**RQ**$_2$) we used the well-known association rule mining, while we measured the relation between structural characteristics and test smell presence (**RQ**$_3$ and **RQ**$_4$) through the use of appropriate statistical procedures. Finally, to complement the quantitative side of the study, we manually investigated the reasons behind the smell-proneness of certain tools.

6.4. Threats to external validity

Threats in this category are mainly related to the generalizability of our findings. We took into account three different automated test case generation tools relying on different algorithms with the aim of considering a wider range of approaches with respect to the creation of test smells. Furthermore, we conducted the experiments on a dataset composed of a large number of classes extracted from the `SF110` dataset [26]. While previous research [7, 24] widely exploited such a dataset, experimenting a different one (*e.g.,* `XCorpus` [100]) would increases the generalizability of the results. This is part of our future agenda. It is worth noting that we filtered out all the trivial classes originally belonging to the `SF110` dataset as done in previous work [7, 30]. Due to the limitations of the test case generation and test smell detection tools, we limited the context of our study only to Java systems. Replications of our work on systems written in other languages are therefore desirable.

7. Conclusion

Test code quality can influence the effectiveness of test cases [19–22]. While previous studies investigated how well-known indicators of poor test code quality, *i.e.,* test smells, influence manually-generated tests [28], in this paper we analyzed the extent to which test smells affect automatically generated test cases. In particular, we conducted a large-scale empirical analysis on the diffuseness of seven test smells in the tests generated by three state-of-the-art testing tools such as RANDOOP [29], JTEXPERT [9], and EVOSUITE [5]. Furthermore, we also assessed the co-occurrences of smells and whether structural indicators of test code correlate with the presence of test smells.

Our findings showed that all the considered tools tend to generate a large amount of instances related to two test smells, *i.e., Assertion Roulette* and *Eager Test*. As a consequence, these tools naturally expose tests to the risk of being less effective when catching faults in production code. Furthermore, test size is correlated with the generation of smelly test cases. Therefore, such characteristic should be keep under control to improve test code quality.

Our future research agenda is based on the output of this paper. We aim at improving current automated testing tools in a way that they avoid the generation of smelly test suites. Furthermore, we aim at replicating the study taking into account testing tools that work on different programming languages as well as different datasets (*e.g.,* the Xcorpus one [100]).

Acknowledgements

Grano and Gall acknowledge the Swiss National Science Foundation (SNSF) under project named "SURF-MobileAppsData" (Project no. 200021_166275). Palomba gratefully acknowledges the support of UZH through the project No. K-34144-01 and the Swiss National Science Foundation through the SNF Project No. PP00P2_170529. The authors would like to thank the anonymous reviewers for the detailed and constructive comments on the preliminary version of this paper, which were instrumental to improving the quality of the work.

References

References

[1] G. J. Myers, C. Sandler, T. Badgett, The art of software testing, John Wiley & Sons, 2011.

[2] M. Beller, G. Georgios, A. Panichella, S. Proksch, S. Amann, A. Zaidman, Developer testing in the ide: Patterns, beliefs, and behavior, IEEE Transactions on Software Engineering (1) (2017) 1–1.

[3] L. S. Pinto, S. Sinha, A. Orso, Understanding myths and realities of test-suite evolution, in: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, ACM, 2012, p. 33.

[4] S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, P. Mcminn, A. Bertolino, et al., An orchestrated survey of methodologies for automated software test case generation, Journal of Systems and Software 86 (8) (2013) 1978–2001.

[5] G. Fraser, 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, ESEC/FSE '11, ACM, New York, NY, USA, 2011, pp. 416–419.

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

[7] A. Panichella, F. M. Kifetew, P. Tonella, Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets, IEEE Transactions on Software Engineering 44 (2) (2018) 122–158.

[8] I. W. B. Prasetya, T3, a combinator-based random testing tool for java: benchmarking, in: Future Internet Testing, Springer, 2014, pp. 101–110.

[9] A. Sakti, G. Pesant, Y.-G. Yann-Gaël Guéhéneuc, Instance generator and problem representation to improve object oriented code coverage, Software Engineering, IEEE Transactions on 41 (3) (2015) 294–313.

[10] J. Ferrer, F. Chicano, E. Alba, Evolutionary algorithms for the multi-objective test data generation problem, Softw. Pract. Exper. 42 (11) (2012) 1331–1362.

[11] K. Lakhotia, M. Harman, P. McMinn, A multi-objective approach to search-based test data generation, in: 9th Conference on Genetic and Evolutionary Computation, GECCO '07, ACM, 2007, pp. 1098–1105.

[12] N. Oster, F. Saglietti, Automatic test data generation by multi-objective optimisation, in: SAFECOMP, Vol. 4166, Springer, 2006, pp. 426–438.

[13] G. Pinto, S. Vergilio, A multi-objective genetic algorithm to test data generation, in: Tools with Artificial Intelligence (ICTAI), 2010 22nd IEEE International Conference on, Vol. 1, 2010, pp. 129–134.

[14] M. Harman, S. G. Kim, K. Lakhotia, P. McMinn, S. Yoo, Optimizing for the number of tests generated in search based test data generation with an application to the oracle cost problem., in: ICST Workshops, IEEE Computer Society, 2010, pp. 182–191.

[15] F. Palomba, A. Panichella, A. Zaidman, R. Oliveto, A. De Lucia, Automatic test case generation: What if test code quality matters?, in: Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, ACM, New York, NY, USA, 2016, pp. 130–141.

[16] G. Grano, F. Palomba, H. C. Gall, Lightweight assessment of test-case effectiveness using source-code-quality indicators, IEEE Transactions on Software Engineering.

[17] A. Arcuri, G. Fraser, On the effectiveness of whole test suite generation, in: Proceedings of the Sixth International Conference on Search Based Software Engineering, SSBSE'14, Springer-Verlag, Berlin, Heidelberg, 2014, pp. 1–15.

[18] G. Grano, S. Scalabrino, H. C. Gall, R. Oliveto, An empirical investigation on the readability of manual and generated test cases, in: ICPC, ACM, 2018, pp. 348–351.

[19] F. Palomba, A. Zaidman, The smell of fear: on the relation between test smells and flaky tests, Empirical Software Engineering (2019) 1–40.

[20] S. Panichella, A. Panichella, M. Beller, A. Zaidman, H. C. Gall, The impact of test case summaries on bug fixing performance: An empirical investigation, in: Proceedings of the 38th International Conference on Software Engineering, ICSE '16, ACM, New York, NY, USA, 2016, pp. 547–558.

[21] S. Shamshiri, R. Just, J. M. Rojas, G. Fraser, P. McMinn, A. Arcuri, Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges, in: Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2015.

[22] D. Spadini, F. Palomba, A. Zaidman, M. Bruntink, A. Bacchelli, On the relation of test smells to software code quality, in: Proceedings of the International Conference on Software Maintenance and Evolution (ICSME). IEEE, 2018.

[23] J. M. Rojas, G. Fraser, A. Arcuri, Automated unit test generation during software development: A controlled experiment and think-aloud observations, in: Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015, ACM, New York, NY, USA, 2015, pp. 338–349.

[24] G. Fraser, A. Arcuri, Whole test suite generation, IEEE Transactions on Software Engineering 39 (2) (2013) 276–291.

[25] F. Palomba, D. Di Nucci, A. Panichella, R. Oliveto, A. De Lucia, On the diffusion of test smells in automatically generated test code: An empirical study, in: Proceedings of the 9th International Workshop on Search-Based Software Testing, SBST '16, ACM, New York, NY, USA, 2016, pp. 5–14.

[26] G. Fraser, A. Arcuri, A large scale evaluation of automated unit test generation using evosuite, ACM Transactions on Software Engineering and Methodology (TOSEM) 24 (2) (2014) 8.

[27] A. van Deursen, L. Moonen, A. Bergh, G. Kok, Refactoring test code, in: Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP), 2001, pp. 92–95.

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

[29] C. Pacheco, S. K. Lahiri, M. D. Ernst, T. Ball, Feedback-directed random test generation, in: Proceedings of the 29th international conference on Software Engineering, IEEE Computer Society, 2007, pp. 75–84.

[30] S. Shamshiri, J. M. Rojas, G. Fraser, P. McMinn, Random or genetic algorithm search for object-oriented test suite generation?, in: Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation, ACM, 2015, pp. 1367–1374.

[31] S. R. Chidamber, C. F. Kemerer, A metrics suite for object oriented design, IEEE Transactions on software engineering 20 (6) (1994) 476–493.

[32] A. Schneider, Junit best practices, Java World, 2000.

[33] D. Di Nucci, F. Palomba, D. A. Tamburri, A. Serebrenik, A. De Lucia, Detecting code smells using machine learning techniques: are we there yet?, in: 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), IEEE, 2018, pp. 612–621.

[34] F. Khomh, S. Vaucher, Y.-G. Guéhéneuc, H. Sahraoui, A bayesian approach for the detection of code and design smells, in: Proceedings of the International Conference on Quality Software (QSIC), IEEE, Hong Kong, China, 2009, pp. 305–314.

[35] M. Lanza, R. Marinescu, Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems, Springer, 2006.

[36] R. Marinescu, Detection strategies: Metrics-based rules for detecting design flaws, in: Proceedings of the International Conference on Software Maintenance (ICSM), 2004, pp. 350–359.

[37] N. Moha, Y.-G. Guéhéneuc, L. Duchien, A.-F. L. Meur, Decor: A method for the specification and detection of code and design smells, IEEE Transactions on Software Engineering 36 (1) (2010) 20–36.

[38] M. J. Munro, Product metrics for automatic identification of "bad smell" design problems in java source-code, in: Proceedings of the International Software Metrics Symposium (METRICS), IEEE, 2005, p. 15.

[39] R. Oliveto, F. Khomh, G. Antoniol, Y.-G. Guéhéneuc, Numerical signatures of antipatterns: An approach based on B-splines, in: Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR), IEEE, 2010, pp. 248–251.

[40] F. Palomba, A. Panichella, A. De Lucia, R. Oliveto, A. Zaidman, A textual-based technique for smell detection, in: Program Comprehension (ICPC), 2016 IEEE 24th International Conference on, IEEE, 2016, pp. 1–10.

[41] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, D. Poshyvanyk, A. De Lucia, Mining version histories for detecting code smells, IEEE Transactions on Software Engineering 41 (5) (2015) 462–489.

[42] F. Palomba, M. Zanoni, F. A. Fontana, A. De Lucia, R. Oliveto, Toward a smell-aware bug prediction model, IEEE Transactions on Software Engineering.

[43] N. Tsantalis, A. Chatzigeorgiou, Identification of move method refactoring opportunities, IEEE Transactions on Software Engineering 35 (3) (2009) 347–367.

[44] M. Abbes, F. Khomh, Y.-G. Guéhéneuc, G. Antoniol, An empirical study of the impact of two antipatterns, Blob and Spaghetti Code, on program comprehension, in: 15th European Conference on Software Maintenance and Reengineering, CSMR 2011, 1-4 March 2011, Oldenburg, Germany, IEEE Computer Society, 2011, pp. 181–190.

[45] R. Arcoverde, A. Garcia, E. Figueiredo, Understanding the longevity of code smells: preliminary results of an explanatory survey, in: Proceedings of the International Workshop on Refactoring Tools, ACM, 2011, pp. 33–36.

[46] G. Bavota, A. De Lucia, M. Di Penta, R. Oliveto, F. Palomba, An experimental investigation on the innate relationship between quality and refactoring, Journal of Systems and Software 107 (2015) 1–14.

[47] A. Chatzigeorgiou, A. Manakos, Investigating the evolution of bad smells in object-oriented code, in: Proceedings of the International Conference on the Quality of Information and Communications Technology (QUATIC), IEEE, 2010, pp. 106–115.

[48] F. Khomh, M. Di Penta, Y.-G. Guéhéneuc, G. Antoniol, An exploratory study of the impact of antipatterns on class change- and fault-proneness, Empirical Software Engineering 17 (3) (2012) 243–275.

[49] A. Lozano, M. Wermelinger, B. Nuseibeh, Assessing the impact of bad smells using historical information, in: Proceedings of the International workshop on Principles of Software Evolution (IWPSE), ACM, 2007, pp. 31–34.

[50] F. Palomba, G. Bavota, M. Di Penta, F. Fasano, R. Oliveto, A. De Lucia, A large-scale empirical study on the lifecycle of code smell co-occurrences, Information and Software Technology 99 (2018) 1–10.

[51] F. Palomba, G. Bavota, M. Di Penta, F. Fasano, R. Oliveto, A. De Lucia, On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation, Empirical Software Engineering 23 (3) (2018) 1188–1221.

[52] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, Do they really smell bad? a study on developers' perception of bad code smells, in: Proceedings of the International Conference on Software Maintenance and Evolution (ICSME), IEEE, 2014, pp. 101–110.

[53] F. Palomba, A. Panichella, A. Zaidman, R. Oliveto, A. De Lucia, The scent of a smell: An extensive comparison between textual and structural smells, IEEE Transactions on Software Engineering.

[54] F. Palomba, A. Zaidman, R. Oliveto, A. De Lucia, An exploratory study on the relationship between changes and refactoring, in: Program Comprehension (ICPC), 2017 IEEE/ACM 25th International Conference on, IEEE, 2017, pp. 176–185.

[55] R. Peters, A. Zaidman, Evaluating the lifespan of code smells using software repository mining, in: Proceedings of the European Conference on Software Maintenance and ReEngineering (CSMR), IEEE, 2012, pp. 411–416.

[56] D. Ratiu, S. Ducasse, T. Gîrba, R. Marinescu, Using history information to improve design flaws detection, in: Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR), IEEE, 2004, pp. 223–232.

[57] D. I. K. Sjøberg, A. F. Yamashita, B. C. D. Anda, A. Mockus, T. Dybå, Quantifying the effect of code smells on maintenance effort, IEEE Trans. Software Eng. 39 (8) (2013) 1144–1156.

[58] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, D. Poshyvanyk, When and why your code starts to smell bad (and whether the smells go away), IEEE Transactions on Software Engineering 43 (11) (2017) 1063–1088.

[59] C. Vassallo, F. Palomba, H. C. Gall, Continuous refactoring in ci: A preliminary study on the perceived advantages and barriers (2018) 564–568.

[60] C. Vassallo, S. Panichella, F. Palomba, S. Proksch, A. Zaidman, H. C. Gall, Context is king: The developer perspective on the usage of static analysis tools, in: 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), IEEE, 2018, pp. 38–49.

[61] A. Yamashita, L. Moonen, Exploring the impact of inter-smell relations on software maintainability: An empirical study, in: Proceedings of the International Conference on Software Engineering (ICSE), IEEE, 2013, pp. 682–691.

[62] A. F. Yamashita, L. Moonen, Do developers care about code smells? an exploratory survey, in: Proceedings of the Working Conference on Reverse Engineering (WCRE), IEEE, 2013, pp. 242–251.

[63] Beck, Test Driven Development: By Example, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[64] C. Pacheco, M. D. Ernst, Randoop: Feedback-directed random testing for java, in: Companion to the 22Nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion, OOPSLA '07, ACM, New York, NY, USA, 2007, pp. 815–816.

[65] I. Prasetya, T. Vos, A. Baars, Trace-based reflexive testing of oo programs with t2, in: Software Testing, Verification, and Validation, 2008 1st International Conference on, 2008, pp. 151–160.

[66] M. Fowler, Refactoring: improving the design of existing code, Addison-Wesley, 1999.

[67] G. Meszaros, xUnit test patterns: Refactoring test code, Pearson Education, 2007.

[68] M. Greiler, A. van Deursen, M.-A. Storey, Automated detection of test fixture strategies and smells, in: Proceedings of the International Conference on Software Testing, Verification and Validation (ICST), 2013, pp. 322–331.

[69] M. Greiler, A. Zaidman, A. van Deursen, M.-A. Storey, Strategies for avoiding text fixture smells during software evolution, in: Proceedings of the 10th Working Conference on Mining Software Repositories (MSR), IEEE, 2013, pp. 387–396.

[70] B. Van Rompaey, B. Du Bois, S. Demeyer, M. Rieger, On the detection of test smells: A metrics-based approach for general fixture and eager test, IEEE Transactions on Software Engineering 33 (12) (2007) 800–817.

[71] F. Palomba, A. Zaidman, A. Lucia, Automatic test smell detection using information retrieval techniques, in: Proceedings of the International Conference on Software Maintenance and Evolution (ICSME). IEEE, 2018.

[72] A. Tahir, S. Counsell, S. G. MacDonell, An empirical study into the relationship between class features and test smells, in: 2016 23rd Asia-Pacific Software Engineering Conference (APSEC), 2016, pp. 137–144.

[73] M. Tufano, F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, D. Poshyvanyk, An empirical investigation into the nature of test smells, in: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, ACM, New York, NY, USA, 2016, pp. 4–15.

[74] J. De Bleser, D. Di Nucci, C. De Roover, Assessing diffusion and perception of test smells in scala projects, in: Proceedings of the 16th International Conference on Mining Software Repositories, 2019.

[75] P. Tonella, Evolutionary testing of classes, in: Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA '04, ACM, New York, NY, USA, 2004, pp. 119–128.

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

[77] R. Hamlet, Random testing, Encyclopedia of software Engineering.

[78] C. Pacheco, M. D. Ernst, Eclat: Automatic generation and classification of test inputs, in: Proceedings of the 19th European conference on Object-Oriented Programming, Springer-Verlag, 2005, pp. 504–527.

[79] B. F. Jones, H.-H. Sthamer, D. E. Eyres, Automatic structural testing using genetic algorithms, Software Engineering Journal 11 (5) (1996) 299–306.

[80] R. P. Pargas, M. J. Harrold, R. R. Peck, Test-data generation using genetic algorithms, Software Testing, Verification and Reliability 9 (4) (1999) 263–282.

[81] S. Scalabrino, G. Grano, D. Di Nucci, R. Oliveto, A. De Lucia, Search-based testing of procedural programs: Iterative single-target or multi-target approach?, in: International Symposium on Search Based Software Engineering, Springer, 2016, pp. 64–79.

[82] A. Arcuri, It does matter how you normalise the branch distance in search based software testing, in: International Conference on Software Testing, Verification and Validation, IEEE, 2010, pp. 205–214.

[83] A. Panichella, F. M. Kifetew, P. Tonella, Incremental control dependency frontier exploration for many-criteria test case generation, in: International Symposium on Search Based Software Engineering, Springer, 2018, pp. 309–324.

[84] S. Afshan, P. McMinn, M. Stevenson, Evolving readable string test inputs using a natural language model to reduce human oracle cost, in: Proceedings International Conference on Software Testing, Verification and Validation (ICST), IEEE, 2013, pp. 352–361.

[85] E. Daka, J. Campos, G. Fraser, J. Dorn, W. Weimer, Modeling readability to improve unit tests, in: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, ACM, New York, NY, USA, 2015, pp. 107–118.

[86] A. Panichella, F. Kifetew, P. Tonella, Reformulating branch coverage as a many-objective optimization problem, in: Software Testing, Verification and Validation (ICST), 2015 IEEE 8th International Conference on, 2015, pp. 1–10.

[87] D. Spinellis, Tool writing: a forgotten art?(software tools), IEEE Software 22 (4) (2005) 9–11.

[88] G. Grano, F. Palomba, D. Di Nucci, A. De Lucia, H. Gall, Scented since the beginning: On the diffuseness of test smells in automatically generated test code - online appendix, `https://github.com/sealuzh/smells-automated`.

[89] G. Fraser, A. Arcuri, Evosuite: automatic test suite generation for object-oriented software, in: Joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ACM, 2011, pp. 416–419.

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

[91] D. Sheskin, Handbook of Parametric and Nonparametric Statistical Procedures., second edition Edition, Chapman & Hall/CRC, 2000.

[92] P. Jaccard, Étude comparative de la distribution florale dans une portion des alpes et des jura, Bull Soc Vaudoise Sci Nat 37 (1901) 547–579.

[93] R. Agrawal, T. Imielinski, A. N. Swami, Mining association rules between sets of items in large databases, in: Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, 1993, pp. 207–216.

[94] M. G. Kendall, A new measure of rank correlation, Biometrika 30 (1/2) (1938) 81–93.

[95] G. Kudrjavets, N. Nagappan, T. Ball, Assessing the relationship between software assertions and faults: An empirical investigation, in: 2006 17th International Symposium on Software Reliability Engineering, IEEE, 2006, pp. 204–212.

[96] A. F. Nogueira, J. C. B. Ribeiro, M. Zenha-Rela, On the evaluation of software maintainability using automatic test case generation, in: 2014 9th International Conference on the Quality of Information and Communications Technology, 2014, pp. 300–305.

[97] R. Wirfs-Brock, B. Wilkerson, Object-oriented design: A responsibility-driven approach, SIGPLAN Not. 24 (10) (1989) 71–75.

[98] G. Bavota, A. De Lucia, A. Marcus, R. Oliveto, Recommending refactoring operations in large software systems, in: Recommendation Systems in Software Engineering, Springer, 2014, pp. 387–419.

[99] J. Campos, A. Arcuri, G. Fraser, R. Abreu, Continuous test generation: enhancing continuous integration with automated test generation, in: Proceedings of the 29th ACM/IEEE international conference on Automated software engineering, 2014, pp. 55–66.

[100] J. Dietrich, H. Schole, L. Sui, E. D. Tempero, Xcorpus - an executable corpus of java programs, Journal of Object Technology 16 (4) (2017) 1:1–24. doi:10.5381/jot.2017.16.4.a1. URL `https://doi.org/10.5381/jot.2017.16.4.a1`