On the Diffusion of Test Smells in Automatically Generated Test Code: An Empirical Study

Fabio Palomba[1], Dario Di Nucci[1], Annibale Panichella[2]
Rocco Oliveto[3], Andrea De Lucia[1]
[1]University of Salerno, Italy — [2]Delft University of Technology, The Netherlands
[3]University of Molise, Italy

ABSTRACT

The role of software testing in the software development process is widely recognized as a key activity for successful projects. This is the reason why in the last decade several automatic unit test generation tools have been proposed, focusing particularly on high code coverage. Despite the effort spent by the research community, there is still a lack of empirical investigation aimed at analyzing the characteristics of the produced test code. Indeed, while some studies inspected the effectiveness and the usability of these tools in practice, it is still unknown whether test code is maintainable. In this paper, we conducted a large scale empirical study in order to analyze the diffusion of bad design solutions, namely *test smells*, in automatically generated unit test classes. Results of the study show the high diffusion of test smells as well as the frequent co-occurrence of different types of design problems. Finally we found that all test smells have strong positive correlation with structural characteristics of the systems such as size or number of classes.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

Keywords

Test Smells, Automatically Generated Test Classes, Mining Software Repositories, Software Quality, Empirical Studies

1. INTRODUCTION

Software testing has a fundamental role in any successful software development process. Despite this, the majority of developers does not write tests and if they do, they tend to overestimate their testing effort [8]. For this reason, in the last decade the automatic generation of test data has received substantial attention from researchers. Many techniques have been proposed and nowadays there are several tools able to produce test suites with high code coverage [14, 25, 35, 40, 43]. Besides techniques able to maximize the code coverage as main goal, also multi-objective approaches have been proposed in order to achieve additional desirable objectives, such as the minimization of (i) the oracle cost [12], (ii) dynamic memory consumption [22], (iii) number of test cases [31], (iv) execution time [38], as well as the maximization of the number of collateral targets that are accidentally covered [18].

Although the effort devoted by the research community in order to define techniques able to automatically generate test cases, there is still a lack of empirical investigations about the characteristics of test code produced by such tools. Indeed, while recent studies have been conducted to evaluate, on the one hand, the effectiveness of test cases [4, 36, 45] and, on the other hand, the usability of these tools in practice [42], it is still unclear whether the test code automatically generated is maintainable. In this paper, we start to face the problem by conducting a large scale empirical study on the *SF110* dataset, a set of 110 open source software projects [15], in order to investigate to what extent JUnit test classes automatically generated by a popular test case generation tool, *EvoSuite* [14], are affected by bad design solutions, namely *test smells*. Specifically, Van Deursen *et al.* defined test smells as symptoms of the presence of poor design or implementation choices in test code [50], following the concept of code smells occurring in production code [13].

In the past, Bavota *et al.* [6] investigated the impact of test smells in the context of test cases manually written by developers, demonstrating that test smells (i) are largely diffused in open source projects, and (ii) have a huge impact on test code comprehensibility and maintainability. Our empirical exploration has the main goal to replicate the study on diffusion of test smells proposed by Bavota *et al.* in the context of JUnit classes automatically generated by EvoSuite, by considering 8 test smells defined by Van Deursen *et al.* [50]. In details, we aim at answering the following research questions:

> **RQ$_1$:** *To what extent test smells are spread in automatically generated test classes?*
>
> **RQ$_2$:** *Which test smells occur more frequently in automatically generated test classes?*
>
> **RQ$_3$:** *Which test smells co-occur together?*
>
> **RQ$_4$:** *Is there a relationships between the presence of test smells and the project characteristics?*

SBST16, May 16-17, 2016, Austin, TX, USA

DOI: http://dx.doi.org/10.1145/2897010.2897016

The results of our empirical study differ from the ones reported in [6] for manually written test classes. Indeed, while the diffusion of test smells is similar (83% of the considered JUnit classes are affected by at least one test smell), we found that 3 test smell types, i.e., *Assertion Roulette*, *Test Code Duplication* and *Eager Test*, are particularly diffused in automatically generated test classes. Moreover, some test smells frequently co-occur together. Finally, almost all the test smells have strong positive correlations with structural characteristics of a system, such as size and number of classes in the project. Furthermore, we qualitatively analyzed several instances in order to understand the reasons behind the achieved results, but also to discover peculiar characteristics of *EvoSuite* that make possible the introduction of test smells during the generation process.

2. BACKGROUND AND RELATED WORK

As well as production code, also test code should be designed following good programming practices [44]. 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 [21, 23, 26, 28, 29, 30, 34, 48], as well as empirical studies aimed at assessing their impact on maintainability [1, 3, 9, 20, 24, 33, 37, 41, 47, 52, 53]. However, design problems affecting test code have been only partially explored. The importance to have well designed test code has been originally highlighted by Beck [7], while Van Deursen *et al.* [50] defined a catalogue of 11 test smells, i.e., a set of a poor design solutions to write tests, together with refactoring operations able to remove them. This catalogue takes into account different types of bad design choises made by developers during the implementation of test fixtures (e.g., `setUp()` method too generic 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). When considering automatically generated test classes, not all the test smells originally defined can be studied. Indeed, almost all the automatic tools for generating test classes do not produce test fixtures [14, 32, 35, 39, 43]. For this reason, our study focus its attention on 8 test smells, i.e., the ones that potentially can affect test code automatically generated. In the following, we provide an overview of such smells, as well as describing the related literature in the field.

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 [50]. Tests containing such smell are difficult to comprehend and maintain, due to the lack of information to understand them. To remove a *Mystery Guest* a *Setup External Resource* operation is needed [50].

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 [50]. Also in this case, to remove the smell a *Setup External Resource* refactoring [50] 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 [50], making the comprehension of the actual test target difficult. A possible solution is represented by the application of a *Extract Method* refactoring, able to split the test method in order to specialize its responsabilities [13].

Assertion Roulette (AR): As defined by Van Deursen *et al.*, this smell *comes from having a number of assertions in a test method that have no explanation* [50]. Thus, if an assertion fails, the identification of the assert that failed can be difficult. Besides removing the unneeded assertions, to remove this smell and make the test more clear an operation of *Add Assertion Explanation* can be applied [50].

Indirect Testing (IT): A test that checks the corresponding production class using methods of another class [50]. 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. Van Deursen *et al.* [50] 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 [50]. 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 [50].

Sensitive Equality (SE): When an assertion contains an equality checks 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. [50]. A simple solution for removing this smell is the application of a *Introduce Equality Method* refactoring, in which the use of the `toString` is replaced by a real equality check.

Test Code Duplication (TCD): Code duplication is a potentially serious problem that affects the maintainability and the comprehensibility of a software system. In test classes, the most common case in which this smell appears is the duplication of code in the same test class. For removing such smell, an *Extract Method* refactoring can be applied in order to put the duplicate code in one single method, and use such method in all the remaining tests.

Listing 1: Example of test method automatically generated using the EvoSuite tool.

```
1 public void test12() throws Throwable  {
2  JSTerm jSTerm0 = new JSTerm();
3  jSTerm0.makeVariable();
4  jSTerm0.add((Object) "");
5  jSTerm0.matches(jSTerm0);
6  assertEquals(false, jSTerm0.isGround());
7  assertEquals(true, jSTerm0.isVariable());
8 }
```

Besides the test smells defined by Van Deursen *et al.* [50], Meszaros defined other smells affecting test code [27]. Starting from these catalogues, Greiler *et al.* [16, 17] 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.* [51] proposed a heuristic structural metric-based approach to identify *General Fixture* and *Eager Test* instances. However, the results of an empirical study demonstrated that structural metrics have low accuracy in the detection of both test smells. As for the empirical study, Bavota *et al.* [6] 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 is affected by at least one test smell, but also that the presence of design flaws has a strong negative impact on maintainability.

Our paper. In this paper we complement, on a larger set of software projects (110 software projects against 18), the analysis on the diffusion of test smells conducted by Bavota *et al.* [6], by considering the diffusion of test smells that can affect JUnit classes automatically generated using search based algorithms, as well as the relationships among them and among structural properties of source code. Specifically, we believe that test smells can have a similar impact on the maintenance activities of automatically generated test cases, as well as they can hinder their comprehensibility. Our observations come from the analysis of the test code generated using different automatic tools. All of them can generate poorly readable test code [2, 11, 42]. For example, they provide *stub* assertions that testers need to modify in order to correctly check that the software behaves as intended [5]. At the same time, as shown in Listing 1, automatic tools provide test code having *stub* names (e.g., `test12`), and identifiers (i.e., usually following the `<class-name><increamental-number>` convention). This makes difficult to understand the actual target methods of the production classes, as well as the modifications needed for fixing incorrect behaviour of the source code. In this context, the presence of test smells can further increase the design problems of test code produced. It is worth noting that in the context of automatic test case generation, the problem of having maintainable test code could be relaxed, since test cases can be re-generated if there are maintainability issues. However, it is important to remark how automatically generated tests need to be maintained once they are generated, since testers must manually validate each test case to check the assertions [5]. As an example of test code presenting design problems, Listing 1 shows a test code that actually checks 2 methods of the production class, i.e., it contains an *Eager Test* smell. In this case, it is almost impossible to understand which method of the production class is the main target, and this potentially decreases the ability of developers to find faults [36].

3. EMPIRICAL STUDY DEFINITION AND DESIGN

This section reports the planning of the study we conducted in order to analyze the distribution of the test smells defined by Van Deursen *et al.* [50] in the context of automatically generated test cases. We choose to analyze the behavior of *EvoSuite* since (i) it is one of the most popular tools for generating test cases, and (ii) a large dataset of JUnit classes generated using this tool is publicly available [15]. The *goals* of the study are (i) determining to what extent unit tests generated by automatic tools present design smells; (ii) identifying the most frequent test smells; (iii) investigating which test smells co-occur; (iv) investigating the correlation between system characteristics (i.e., test class LOC, production code LOC, number of classes, number of JUnit classes) and presence of test smells. Specifically, in the context of our study we formulated the following research questions:

> **RQ$_1$:** *To what extent test smells are spread in automatically generated test classes?* This research question aims at quantifying the presence of test smells in the test classes automatically generated using the *EvoSuite* tool.
>
> **RQ$_2$:** *Which test smells occur more frequently in automatically generated test classes?* With this research question, we are interested in understanding which test smells are more frequently introduced during the automatic generation process.
>
> **RQ$_3$:** *Which test smells co-occur together?* In this analysis we analyze which test smells co-occur in the test classes automatically generated using the *EvoSuite* tool.
>
> **RQ$_4$:** *Is there a relationships between the presence of test smells and the project characteristics?* This research question aims at investigating the relationships between the test smell presence and the characteristics of a software system.

Table 1: Characteristics of the SF110 dataset [15]

Characteristic	Value
Number of Projects	110
Number of Testable Classes	23,886
Lines of Code	6,628,619
Number of Java Files	27,997

Table 2: The Rules used by the `Test Smell Detector` to Detect Candidate Test Smells.

Name	Abbr.	Description
Mystery Guest	MG	JUnit classes that use an external resource (e.g., a file or database).
Resource Optimism	RO	JUnit classes that use an external resource that is not present on the disk.
Eager Test	EG	JUnit classes having at least one method that uses more than one method of the tested class.
Assertion Roulette	AR	JUnit classes containing at least one method having more than one assertion statement, and having at least one assertion statement without explanation.
Indirect Testing	IT	JUnit classes invoking, besides methods of the tested class, methods of other classes in the production code.
For Testers Only	FTO	Classes in the production code having structural relationship (e.g., method invocations, inheritance) with only JUnit classes.
Sensitive Equality	SE	JUnit classes having at least one assert statement invoking a toString method.
Test Code Duplication	TCD	JUnit classes identified as containing clones by the Deckard clone detection tool [19].

To answer our research questions we firstly mined the JUnit test classes automatically generated by EvoSuite on the *SF110* dataset, a set of 110 open source software projects publicly available [15]. Table 1 reports the statistics of the dataset used in this paper, in terms of (i) number of projects, (ii) number of testable classes, (iii) lines of code, and (iv) number of Java files/classes belonging to the dataset. The choice of the systems to consider in the experiment is not random but guided by the will to have a large number of systems of different nature and different domains, that allow us to generalize the results of our study. Indeed, the dataset contains a statistically representative sample of the projects hosted on SourceForge[1]. Because of the number of systems analyzed, a manual detection of the 8 test smells is prohibitively expensive. This is the reason why we used a detection tool, named `Test Smell Detector`, implemented by Bavota *et al.* in [6]. Unlike other existing detection tools (e.g., [16] and [51]), the selected tool is able to identify all

[1]http://sourceforge.net

the test smells considered in this study by applying a heuristic metric-based technique that overestimate the presence of test design flaws in order to detect all the instances (100% of recall), having an average precision of 88%. Table 2 reports the set of rules used by the tool in order to detect instances of test smells. In the context of this study, to test the actual precision of the tool, we manually validate a statistically significant set of 378 JUnit test classes marked as smelly by the `Test Smell Detector`. Specifically, two of the authors performed the validation independently, and cases of disagreement were discussed. Such a (stratified) sample is deemed to be statistically significant for a 95% confidence level and ±5% confidence interval [46]. The results of the manual validation shows that the precision of the tool is 75%.

To answer $\mathbf{RQ}_1$ we verified what is the distribution of the considered test smells in the analyzed software projects. Once obtained the data of their diffusion, to answer $\mathbf{RQ}_2$ we verified which are the test smells that occur more frequently in the systems of our study. As for $\mathbf{RQ}_3$, we investigated how often the presence of a test smell in a JUnit class implies the presence of another test smell. Specifically, for each test smell t_i we measured the percentage of times its presence in a JUnit class co-occurs with another test smell t_j ($i \neq j$). Formally, for each pair of test smells t_i and t_j we compute the percentage of co-occurrences of t_i and t_j using the formula shown below:

$$co\text{-}occurrences_{t_i,t_j} = \frac{|t_i \wedge t_j|}{|t_i|} \quad (1)$$

where where $|t_i \wedge t_j|$ | is the number of times t_i co-occurs with t_j, and $|t_i|$ is the number of occurrences of t_i. Finally, we verified if there exists a relationship between the presence of smells and some structural characteristics, i.e., production code LOC, number of Classes, number of JUnit Classes, and LOC of JUnit Classes, of a system. In particular, we computed, for each system in our study, the Pearson product-Moment Correlation Coefficient (PMCC) [10] between the values of each system's characteristic and the percentage of occurrences of each test smell in this system (as done in [6] for manually written tests). PMCC is a measure of correlation between two variables X and Y defined in [-1; 1], where 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. Cohen *et al.* [10] provided a set of guidelines for the interpretation of the correlation coefficient. It is assumed 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.

4. ANALYSIS OF THE RESULTS

Table 3 reports the results about the diffusion of the 8 considered test smells, achieved by running the `Test Smell Detector` over the SF110 dataset. It is worth noting that the results for the *For Testers Only* test smell are not reported in the table, since this type of smell appears in the production code and not in the JUnit classes. However, we found 5,632 instances of this smell in our dataset (34% of the analyzed JUnit classes), i.e., 5,632 production classes are used only by some test methods. As for the other test smells considered in our study, the first thing that leaps to the eye is the percentage of JUnit test classes actually containing at least one test smell. Indeed, we found that 13,791 out of the total 16,603 JUnit classes (i.e., 83% of them) are affected by design problems. This result highlights on the one hand the high diffusion of test smells, and on the other hand the inability of the automatic test case generation tools to deal with design problems during the generation of test classes. Particularly interesting is the case of the `ifx-framework` project, where all the 3,899 JUnit classes are affected by at least one test smell.

Listing 2: Test Method automatically generated by EvoSuite containing a Sensitive Equality test smell.

```
1 public void test8()  throws Throwable  {
2  BankAcctTaxInqRq_Type
    bankAcctTaxInqRq_Type0 = new
    BankAcctTaxInqRq_Type();
3  DeliveryMethod deliveryMethod0 = new
    DeliveryMethod();
4  bankAcctTaxInqRq_Type0.setDeliveryMethod(
    deliveryMethod0);
5  assertEquals("org.sourceforge.ifx.framework
    .element.DeliveryMethod = {\n  String =
    NULL\n}", deliveryMethod0.toString());
6 }
```

As an example, Listing 2 shows the test method `test8` of the `BankAcctTaxInqRq_TypeEvoSuiteTest` contained in the package `org.sourceforge.ifx.framework.complextype`.

This method is affected by a *Sensitive Equality* test smell, since the assert statement contains an equality check through the use of the `toString` method. More in general, in this project we found a large number of instances of half of the types of smells we considered. Indeed, over the total 3,899 test classes, we found 1,190 instances of *Assertion Roulette*, 168 instances of *Eager Test*, 1,180 *Sensitive Equality*, and 1,178 *Test Code Duplication*. On the other hand, the project does not contain any instance of *Mystery Guest*, *Resource Optimism* and *Indirect Testing*.

Looking at how frequently specific design flaws are present in the software projects, we can observe that three test smells, i.e. *Assertion Roulette*, *Eager Test*, and *Test Code Duplication*, are particularly diffused. Specifically, in our dataset the *Assertion Roulette* smell occurs in 54% of the JUnit classes, *Eager Test* instances are present in 29% of them, while the *Test Code Duplication* occurs in 33% of the classes. The high diffusion of *Assertion Roulette* confirms previous findings [6], while the number of instances of *Test Code Duplication* that we found is slightly higher with respect to what reported in [6] for manually produced test suites. These design problems are probably generated because the automatic test case generation tools have, as primary objectives, the goal to cover as more as possible the production classes, without considering the quality of the generated test classes. An interesting example, shown in Listing 4 regards the case of the `JGaap` project, where the `DocumentEvoSuiteTest` JUnit class is affected by an *Assertion Roulette* instance.

Here, there are four assertions without a clear explanation of their individual responsibility. Indeed, it can be difficult to understand what is the behavior under test, as well as

Table 3: Distribution of Test Smells in SF110 Corpus of Classes

Project	# JUnit Tests	# JUnit Tests With Test Smells	Assertion Roulette	Eager Test	Mystery Guest	Sensitive Equality	Resource Optimism	Indirect Testing	Test Code Duplication
jgaap	19	13 (68%)	10 (77%)	3 (23%)	1 (8%)	6 (46%)	0 (0%)	0 (0%)	0 (0%)
netweaver	177	149 (84%)	122 (82%)	81 (54%)	16 (11%)	14 (9%)	5 (3%)	0 (0%)	92 (62%)
squirrel-sql	636	419 (66%)	369 (88%)	196 (47%)	50 (12%)	63 (15%)	13 (3%)	4 (1%)	126 (30%)
sweethome3d	152	96 (63%)	81 (84%)	58 (60%)	15 (16%)	8 (8%)	7 (7%)	4 (4%)	38 (40%)
vuze	729	605 (83%)	548 (91%)	345 (57%)	81 (13%)	110 (18%)	18 (3%)	15 (2%)	174 (29%)
freemind	205	129 (63%)	113 (88%)	55 (43%)	18 (14%)	28 (22%)	5 (4%)	5 (4%)	28 (22%)
checkstyle	18	12 (67%)	10 (83%)	4 (33%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	1 (8%)
weka	352	329 (93%)	304 (92%)	195 (59%)	45 (14%)	98 (30%)	12 (4%)	54 (16%)	177 (54%)
liferay	4207	3510 (83%)	2469 (70%)	1506 (43%)	122 (3%)	237 (7%)	44 (1%)	2 (0%)	2084 (59%)
pdfsam	315	214 (68%)	184 (86%)	103 (48%)	46 (21%)	16 (7%)	37 (17%)	1 (0%)	84 (39%)
water-simulator	48	37 (77%)	32 (86%)	11 (30%)	0 (0%)	2 (5%)	0 (0%)	0 (0%)	2 (5%)
firebird	185	164 (89%)	145 (88%)	83 (51%)	28 (17%)	14 (9%)	0 (0%)	1 (1%)	80 (49%)
imsmart	15	14 (93%)	7 (50%)	3 (21%)	7 (50%)	0 (0%)	1 (7%)	1 (7%)	7 (50%)
dsachat	31	24 (77%)	16 (67%)	11 (46%)	2 (8%)	8 (33%)	0 (0%)	2 (8%)	2 (8%)
jdbacl	108	94 (87%)	86 (91%)	47 (50%)	17 (18%)	20 (21%)	3 (3%)	0 (0%)	29 (31%)
omjstate	8	8 (100%)	7 (88%)	4 (50%)	0 (0%)	1 (13%)	0 (0%)	0 (0%)	1 (13%)
beanbin	73	55 (75%)	42 (76%)	19 (35%)	7 (13%)	3 (5%)	2 (4%)	0 (0%)	8 (15%)
templatedetails	1	1 (100%)	1 (100%)	1 (100%)	1 (100%)	1 (100%)	0 (0%)	0 (0%)	1 (100%)
inspirento	26	23 (88%)	21 (91%)	13 (57%)	1 (4%)	5 (22%)	0 (0%)	1 (4%)	2 (9%)
jsecurity	138	124 (90%)	101 (81%)	54 (44%)	17 (14%)	15 (12%)	0 (0%)	0 (0%)	33 (27%)
jmca	33	11 (33%)	9 (82%)	4 (36%)	1 (9%)	2 (18%)	0 (0%)	0 (0%)	2 (18%)
tullibee	17	17 (100%)	15 (88%)	5 (29%)	1 (6%)	0 (0%)	0 (0%)	0 (0%)	3 (18%)
nekomud	7	6 (86%)	3 (50%)	2 (33%)	1 (17%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)
geo-google	52	49 (94%)	49 (100%)	41 (84%)	1 (2%)	1 (2%)	1 (2%)	0 (0%)	35 (71%)
byuic	12	9 (75%)	9 (100%)	6 (67%)	0 (0%)	1 (11%)	0 (0%)	0 (0%)	4 (44%)
jwbf	48	30 (63%)	28 (93%)	23 (77%)	6 (20%)	6 (20%)	4 (13%)	0 (0%)	5 (17%)
saxpath	8	8 (100%)	7 (88%)	6 (75%)	0 (0%)	1 (13%)	0 (0%)	0 (0%)	5 (63%)
jni-inchi	22	11 (50%)	11 (100%)	7 (64%)	1 (9%)	1 (9%)	0 (0%)	0 (0%)	0 (0%)
jipa	2	2 (100%)	2 (100%)	1 (50%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)
gangup	92	43 (47%)	42 (98%)	27 (63%)	0 (0%)	19 (44%)	0 (0%)	0 (0%)	6 (14%)
greencow	1	1 (100%)	1 (100%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)
apbsmem	44	42 (95%)	40 (95%)	11 (26%)	1 (2%)	2 (5%)	1 (2%)	0 (0%)	21 (50%)
a4j	22	22 (100%)	22 (100%)	17 (77%)	5 (23%)	12 (55%)	0 (0%)	0 (0%)	14 (64%)
bpmail	21	17 (81%)	17 (100%)	10 (59%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)
xisemele	38	31 (82%)	11 (35%)	9 (29%)	2 (6%)	0 (0%)	2 (6%)	0 (0%)	4 (13%)
httpanalyzer	18	11 (61%)	11 (100%)	1 (9%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)
javaviewcontrol	14	14 (100%)	12 (86%)	5 (36%)	1 (7%)	7 (50%)	1 (7%)	0 (0%)	2 (14%)
sbmlreader2	6	5 (83%)	5 (100%)	0 (0%)	1 (20%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)
corina	131	90 (69%)	71 (79%)	36 (40%)	9 (10%)	23 (26%)	4 (4%)	1 (1%)	16 (18%)
schemaspy	21	13 (62%)	6 (46%)	6 (46%)	1 (8%)	0 (0%)	0 (0%)	0 (0%)	1 (8%)
petsoar	9	6 (67%)	5 (83%)	2 (33%)	1 (17%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)
javabullboard	40	39 (98%)	37 (95%)	29 (74%)	17 (44%)	6 (15%)	0 (0%)	1 (3%)	8 (21%)
diffi	10	8 (80%)	8 (100%)	5 (63%)	0 (0%)	2 (25%)	0 (0%)	0 (0%)	1 (13%)
gaj	9	8 (89%)	7 (88%)	3 (38%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)
glengineer	15	15 (100%)	12 (80%)	7 (47%)	0 (0%)	10 (67%)	0 (0%)	0 (0%)	6 (40%)
follow	20	15 (75%)	14 (93%)	7 (47%)	1 (7%)	2 (13%)	1 (7%)	0 (0%)	1 (7%)
asphodel	8	8 (100%)	3 (38%)	5 (63%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)
lilith	165	107 (65%)	95 (89%)	39 (36%)	10 (9%)	27 (25%)	10 (9%)	0 (0%)	46 (43%)
summa	163	112 (69%)	90 (80%)	55 (49%)	9 (8%)	27 (24%)	5 (4%)	0 (0%)	28 (25%)
lotus	54	39 (72%)	16 (41%)	3 (8%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	1 (3%)
nutzenportfolio	47	45 (96%)	22 (49%)	21 (47%)	0 (0%)	8 (18%)	0 (0%)	0 (0%)	40 (89%)
dvd-homevideo	9	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)
resources4j	5	3 (60%)	3 (100%)	3 (100%)	1 (33%)	0 (0%)	1 (33%)	0 (0%)	0 (0%)
diebierse	12	6 (50%)	5 (83%)	2 (33%)	0 (0%)	2 (33%)	0 (0%)	0 (0%)	1 (17%)
rif	14	11 (79%)	6 (55%)	6 (55%)	2 (18%)	0 (0%)	0 (0%)	0 (0%)	2 (18%)
biff	5	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)
jiprof	102	85 (83%)	77 (91%)	34 (40%)	9 (11%)	15 (18%)	0 (0%)	2 (2%)	38 (45%)
lagoon	66	39 (59%)	28 (72%)	9 (23%)	5 (13%)	2 (5%)	4 (10%)	1 (3%)	10 (26%)
shp2kml	4	3 (75%)	3 (100%)	1 (33%)	1 (33%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)
db-everywhere	75	69 (92%)	54 (78%)	27 (39%)	6 (9%)	16 (23%)	0 (0%)	0 (0%)	41 (59%)
lavalamp	24	15 (63%)	11 (73%)	12 (80%)	1 (7%)	9 (60%)	0 (0%)	0 (0%)	2 (13%)
jhandballmoves	50	46 (92%)	37 (80%)	8 (17%)	37 (80%)	9 (20%)	3 (7%)	0 (0%)	15 (33%)
hft-bomberman	124	68 (55%)	31 (46%)	17 (25%)	3 (4%)	5 (7%)	0 (0%)	0 (0%)	29 (43%)
fps370	12	6 (50%)	3 (50%)	1 (17%)	1 (17%)	1 (17%)	0 (0%)	0 (0%)	2 (33%)
mygrid	34	32 (94%)	32 (100%)	24 (75%)	3 (9%)	2 (6%)	0 (0%)	0 (0%)	29 (91%)
templateit	15	14 (93%)	13 (93%)	8 (57%)	0 (0%)	4 (29%)	0 (0%)	0 (0%)	1 (7%)
sugar	25	22 (88%)	20 (91%)	14 (64%)	5 (23%)	7 (32%)	4 (18%)	0 (0%)	10 (45%)
noen	377	272 (72%)	188 (69%)	105 (39%)	7 (3%)	27 (10%)	0 (0%)	0 (0%)	130 (48%)
dom4j	72	69 (96%)	67 (97%)	54 (78%)	6 (9%)	9 (13%)	5 (7%)	0 (0%)	10 (14%)
objectexplorer	70	53 (76%)	45 (85%)	28 (53%)	1 (2%)	8 (15%)	0 (0%)	0 (0%)	8 (15%)
jtailgui	38	26 (68%)	19 (73%)	7 (27%)	8 (31%)	4 (15%)	5 (19%)	0 (0%)	4 (15%)
gsftp	14	7 (50%)	7 (100%)	5 (71%)	2 (29%)	1 (14%)	1 (14%)	0 (0%)	2 (29%)
openjms	508	373 (73%)	299 (80%)	201 (54%)	15 (4%)	43 (12%)	1 (0%)	0 (0%)	170 (46%)
gae-app-manager	8	4 (50%)	4 (100%)	2 (50%)	1 (25%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)
biblestudy	20	17 (85%)	17 (100%)	15 (88%)	6 (35%)	6 (35%)	0 (0%)	0 (0%)	3 (18%)
lhamacaw	97	64 (66%)	51 (80%)	34 (53%)	8 (13%)	17 (27%)	0 (0%)	1 (2%)	33 (52%)
jnfe	51	18 (35%)	6 (33%)	5 (28%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	9 (50%)
echodep	79	61 (77%)	26 (43%)	11 (18%)	7 (11%)	11 (18%)	6 (10%)	0 (0%)	21 (34%)
ext4j	24	18 (75%)	14 (78%)	4 (22%)	1 (6%)	0 (0%)	0 (0%)	0 (0%)	4 (22%)
battlecry	11	5 (45%)	4 (80%)	2 (40%)	1 (20%)	1 (20%)	0 (0%)	0 (0%)	2 (40%)
fim1	50	35 (70%)	33 (94%)	16 (46%)	5 (14%)	13 (37%)	2 (6%)	0 (0%)	10 (29%)
fixsuite	22	18 (82%)	15 (83%)	6 (33%)	2 (11%)	2 (11%)	2 (11%)	0 (0%)	5 (28%)
openhre	79	70 (89%)	62 (89%)	52 (74%)	4 (6%)	32 (46%)	1 (1%)	1 (1%)	36 (51%)
dash-framework	14	12 (86%)	2 (17%)	1 (8%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)
io-project	14	9 (64%)	5 (56%)	4 (44%)	1 (11%)	0 (0%)	0 (0%)	0 (0%)	2 (22%)
caloriecount	571	444 (78%)	357 (80%)	213 (48%)	41 (9%)	80 (18%)	29 (7%)	0 (0%)	82 (18%)
twfbplayer	85	58 (68%)	49 (84%)	20 (34%)	0 (0%)	18 (31%)	0 (0%)	0 (0%)	17 (29%)
sfmis	18	14 (78%)	13 (93%)	8 (57%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	7 (50%)
wheelwebtool	92	78 (85%)	64 (82%)	45 (58%)	3 (4%)	30 (38%)	1 (1%)	0 (0%)	25 (32%)
javathena	48	39 (81%)	37 (95%)	30 (77%)	11 (28%)	5 (13%)	0 (0%)	0 (0%)	21 (54%)
gaj	0	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)
ipcalculator	10	6 (60%)	4 (67%)	3 (50%)	1 (17%)	0 (0%)	0 (0%)	0 (0%)	2 (33%)
xbus	177	96 (54%)	64 (67%)	32 (33%)	9 (9%)	4 (4%)	2 (2%)	0 (0%)	32 (33%)
ifx-framework	3899	3899 (100%)	1190 (31%)	168 (4%)	0 (0%)	1180 (30%)	0 (0%)	0 (0%)	1178 (30%)
shop	33	31 (94%)	29 (94%)	13 (42%)	0 (0%)	9 (29%)	0 (0%)	0 (0%)	7 (23%)
at-robots2-j	191	132 (69%)	114 (86%)	68 (52%)	5 (4%)	24 (18%)	4 (3%)	0 (0%)	21 (16%)
jaw-br	22	10 (45%)	9 (90%)	5 (50%)	1 (10%)	2 (20%)	0 (0%)	0 (0%)	2 (20%)
jopenchart	28	19 (68%)	16 (84%)	10 (53%)	1 (5%)	6 (32%)	0 (0%)	0 (0%)	1 (5%)
jiggler	124	112 (90%)	96 (86%)	76 (68%)	14 (13%)	56 (50%)	0 (0%)	0 (0%)	30 (27%)
gfarcegestionfa	46	38 (83%)	25 (66%)	7 (18%)	7 (18%)	5 (13%)	1 (3%)	0 (0%)	6 (16%)
dcparseargs	6	6 (100%)	4 (67%)	1 (17%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	1 (17%)
classviewer	6	5 (83%)	5 (100%)	5 (100%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	1 (20%)
jcvi-javacommon	305	218 (71%)	165 (76%)	123 (56%)	59 (27%)	38 (17%)	56 (26%)	1 (0%)	65 (30%)
quickserver	63	53 (84%)	44 (83%)	34 (64%)	2 (4%)	9 (17%)	0 (0%)	0 (0%)	17 (32%)
jclo	4	1 (25%)	1 (100%)	1 (100%)	0 (0%)	1 (100%)	0 (0%)	0 (0%)	0 (0%)
celwars2009	7	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)
heal	89	83 (93%)	82 (99%)	37 (45%)	8 (10%)	14 (17%)	1 (1%)	0 (0%)	38 (46%)
feudalismgame	11	9 (82%)	6 (67%)	1 (11%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	1 (11%)
trans-locator	5	4 (80%)	4 (100%)	2 (50%)	0 (0%)	1 (25%)	0 (0%)	0 (0%)	0 (0%)
newzgrabber	39	24 (62%)	22 (92%)	14 (58%)	1 (4%)	6 (25%)	0 (0%)	0 (0%)	4 (17%)
falselight	8	8 (100%)	5 (63%)	0 (0%)	1 (13%)	1 (13%)	0 (0%)	0 (0%)	0 (0%)
All systems	**16,603**	**13,791 (83%)**	**8,939 (54%)**	**4,819 (29%)**	**856 (5%)**	**2,541 (15%)**	**305 (2%)**	**51 (0,3%)**	**5,405 (33%)**

whether the generated assertions are correct. As recently demonstrated by Panichella *et al.* [36], problems like this can have a huge impact on the ability of developers in finding faults in production code.

Listing 3: Test Method automatically generated by EvoSuite containing an Assertion Roulette test smell.

```
1  public void test8()  throws Throwable  {
2   Document document0=new Document("", "");
3   assertNotNull(document0);
4
5   document0.procText.add((Character) 's');
6   String string0 = document0.stringify();
7   assertEquals("s", document0.stringify());
8   assertNotNull(string0);
9   assertEquals("s", string0);
10 }
```

Listing 4: Test Method automatically generated by EvoSuite containing an Eager Test smell.

```
1  public void test2()  throws Throwable  {
2   SecureMessageServiceClientImpl
     secureMessageServiceClientImpl0 = new
     SecureMessageServiceClientImpl("VH_:
     A7ZXfnoaf I%e<6", 0, 0, (RSAPublicKey)
     null, (SecureMessageServiceClientAdapter)
     null);
3   SecureMessageServiceClientMessageImpl
     secureMessageServiceClientMessageImpl0 = (
     SecureMessageServiceClientMessageImpl)
     secureMessageServiceClientImpl0.
     sendMessage((Map) null, (Object) "
     1111111111111111", "VH_:A7ZXfnoaf I%e<6")
     ;
4   assertNotNull(
     secureMessageServiceClientMessageImpl0);
5
6   secureMessageServiceClientImpl0.cancel((
     SecureMessageServiceClientMessage)
     secureMessageServiceClientMessageImpl0);
7   secureMessageServiceClientImpl0.cancel((
     SecureMessageServiceClientMessage)
     secureMessageServiceClientMessageImpl0);
8   assertEquals("VH_:A7ZXfnoaf I%e<6",
     secureMessageServiceClientMessageImpl0.
     getString());
9  }
```

Concerning *Eager Test*, several test classes automatically generated are affected by such a smell (33% of the classes in our dataset). An example of test class containing this smell is the `SecureMessageServiceClientImplEvoSuiteTest` from the `Vuze` project. Listing 4 clearly shows that the test method named `test2` is responsible to check, on the one hand, the behavior of the production method `sendMessage` through the usage of the assertion on line 4, and on the other hand the production method `getString` on line 8. Also in this case, the presence of such smell can seriously affects the comprehensibility of the behavior of the test class, as well as the maintenance activities aimed at modifying it [6].

Finally, *Test Code Duplication* affects the 30% of the JUnit classes in the SF110 dataset. Looking in depth into the causes for this smell for generated tests, we observed that the main problem is that EvoSuite (but also the other automatic test case generation tools) does not generate text fixtures, and this entails a duplication of lines of code over all the test methods contained in a JUnit class. An example is shown below, where a pair of test methods from the `GenericPropertiesCreatorEvoSuiteTest` class of the `Weka` project is proposed.

Listing 5: Pair of Test Methods automatically generated by EvoSuite containing a Test Code Duplication smell

```
1  public void test8()  throws Throwable  {
2   GenericPropertiesCreator
     genericPropertiesCreator0 = new
     GenericPropertiesCreator();
3   boolean boolean0 =
     genericPropertiesCreator0.isValidClassname
     ("*(Uyn-KZX!S$YwYJaZV");
4   assertEquals(false,
     genericPropertiesCreator0.
     getExplicitPropsFile());
5   assertEquals("weka/gui/GenericObjectEditor.
     props", genericPropertiesCreator0.
     getOutputFilename());
6   assertEquals("weka/gui/
     GenericPropertiesCreator.props",
     genericPropertiesCreator0.getInputFilename
     ());
7   assertEquals(false, boolean0);
8  }
```

```
1  public void test9()  throws Throwable  {
2   GenericPropertiesCreator
     genericPropertiesCreator0 = new
     GenericPropertiesCreator();
3   boolean boolean0 =
     genericPropertiesCreator0.isValidClassname
     ("%1rZ0LSExSk%");
4   assertEquals("weka/gui/GenericObjectEditor.
     props", genericPropertiesCreator0.
     getOutputFilename());
5   assertEquals("weka/gui/
     GenericPropertiesCreator.props",
     genericPropertiesCreator0.getInputFilename
     ());
6   assertEquals(true, boolean0);
7   assertEquals(false,
     genericPropertiesCreator0.
     getExplicitPropsFile());
8  }
```

The smell arises since both the methods share a portion of code, i.e., the construction of the class `GenericPropertiesCreator`. These lines of code are also shared by the other test methods in the suite. This means that such lines should be included in a separate method, namely the `setUp` one, which is responsible for the setting of all the test methods. The quite high diffusion of this smell remarks the need to automatically generate text fixtures. As for the other test smells, we observed quite low diffusion in the analyzed systems. These results are in line with what reported in [6] for manually written test classes. Table 4 reports the results about the co-occurrences of test smells inside the JUnit classes. The first interesting thing to note is that all the test smells tend to co-occur frequently with *Assertion Roulette*. On the one hand, the reason of this result can be found in the high diffusion of this smell (occurring in the 54% of JUnit classes of the dataset). On the other hand, smells as *Sensitive Equality* and *Eager Test* are, in different ways, in-

trinsically related to the *Assertion Roulette* smell. Indeed, if a test case contains several assertions, then, it would likely contain a equality check through the use of the `toString` method (i.e., the test case contains a *Sensitive Equality*). Similarly, if a test case performs many assertions, then it is reasonable to think that it checks more than one method of the production class (i.e., it is also an *Eager Test*). Another interesting relationship is the one between *Mystery Guest* and *Resource Optimism*. In this case, when a test case uses external resources it is likely that it also makes assumptions about the state or the existence of such external resources. Potentially more interesting is what we found about the co-occurrences between *Mystery Guest* and *Indirect Testing*. In this case every time that an instance of *Mystery Guest* occurs, also an instance of *Indirect Testing* is present. As shown in the example reported in Listing 6, when a test case refers to an external resource, it always uses an intermediate class in order to perform actions on that resources. Specifically, in the case of the test methods `test3` and `test4` from the `SweetHome3D` project, the goal is to test the `SweetHome3D` class. However, both the tests use the intermediate class `HomeFileRecorder` in order to perform operations on the class under test.

Table 4: Co-occurrences among Test Smells in the SF110 Dataset

	AR	ET	MG	SE	RO	IT	TCD
AR	-	0.49	0.07	0.27	0.02	0.01	0.48
ET	**0.91**	-	0.10	0.24	0.04	0.01	**0.50**
MG	**0.77**	**0.54**	-	0.15	0.36	0.06	**0.62**
SE	**0.96**	0.45	0.05	-	0.02	0.01	**0.67**
RO	**0.78**	**0.58**	**1.00**	0.12	-	0.03	0.45
IT	**0.73**	**0.59**	**1.00**	0.16	0.16	-	**0.80**
TCD	**0.80**	0.45	0.10	0.32	0.03	0.01	-

As for *Test Code Duplication*, this smell frequently occurs with the *Indirect Testing* smell. Looking more in depth to the test cases produced by EvoSuite, we observed that such co-occurrence is due to the fact that the test cases refer to the intermediate class using the same set of statements. The example in Listing 6 is also valid to explain this co-occurrence. Indeed, we can see that both `test3` and `test4` refer to the intermediate class through the usage of the same set of instructions, i.e., lines 4 and 5 of the test methods.

Listing 6: Pair of Test Methods automatically generated by EvoSuite containing Mystery Guest together with Indirect Testing and Test Code Duplication smells.

```
1 public void test3()  throws Throwable  {
2  SweetHome3D sweetHome3D0 = new SweetHome3D
    ();
3  HomeRecorder.Type homeRecorder_Type0 =
    HomeRecorder.Type.DEFAULT;
4  HomeFileRecorder homeFileRecorder0 = (
    HomeFileRecorder)sweetHome3D0.
    getHomeRecorder(homeRecorder_Type0);
5  assertNotNull(homeFileRecorder0);
6 }
```

For the other test smells, we did not observe any other interesting relationship. Comparing our results with the ones reported by Bavota *et al.* for manually written test classes [6], we found several differences. Indeed, they demonstrated that all the test smells frequently co-occur with *Assertion Roulette*, while in our study we found several peculiarities that characterize automatic test case generation tools, such as co-occurrences of different types of test smells inside JUnit test classes.

```
1 public void test4()  throws Throwable  {
2  SweetHome3D sweetHome3D0 = new SweetHome3D
    ();
3  HomeRecorder.Type homeRecorder_Type0 =
    HomeRecorder.Type.COMPRESSED;
4  HomeFileRecorder homeFileRecorder0 = (
    HomeFileRecorder)sweetHome3D0.
    getHomeRecorder(homeRecorder_Type0);
5  HomeFileRecorder homeFileRecorder1 = (
    HomeFileRecorder)sweetHome3D0.
    getHomeRecorder(homeRecorder_Type0);
6  assertNotNull(homeFileRecorder1);
7  assertSame(homeFileRecorder1,
    homeFileRecorder0);
8 }
```

Table 5: Correlations between Structural Characteristics and Test Smell Presence (PMCC)

	# Classes	# JUnit Classes	LOC	JUnit LOC
Assertion Roulette	**0.98**	**0.95**	**0.94**	**0.98**
Eager Test	**0.94**	**0.80**	**0.98**	**0.98**
Mystery Guest	**0.78**	**0.58**	**0.79**	**0.74**
Sensitive Equality	**0.57**	**0.80**	0.32	0.45
Resource Optimism	**0.58**	0.44	**0.56**	**0.55**
Indirect Testing	0.40	0.19	0.36	0.24
Test Code Duplication	**0.95**	**0.97**	**0.90**	**0.96**

As additional analysis, we also verify possible correlations between the presence of test smells and peculiar system's characteristics. Table 5 reports the PMCC for the analyzed correlations. As we can see, there are many strong correlations between the system characteristics and the presence of test smells in their test classes. Indeed, with the exception of *Indirect Testing*, all the test smells have positive correlations with all the structural characteristics taken into account. In other words, the larger the system (in terms of number of classes, number of JUnit classes, production class LOC, or JUnit class LOC), the higher the likelihood that its JUnit classes are affected by *Assertion Roulette*, *Eager Test*, *Mystery Guest*, *Sensitive Equality*, *Resource Optimism*, and *Test Code Duplication*. This result strongly differs from the findings reported for manually written tests [6]. However, such differences can be explained by looking more in depth into the way the test case generation process is performed in EvoSuite. Specifically, considering that the main goal is to cover as many branches as possible, it is reasonable to think that projects having large size are more complex, and thus, more difficult to test. Hence, in order to test large and/or complex classes likely a more complex test code is needed. Similarly, to obtain higher coverage often the instantiation of different objects that interact with the class under test might be required. As a result, test classes become more complex as well, leading to the introduction of test smells. Moreover, while developers can keep under control the quality of source code while writing test classes, an automatic tool cannot do this if not property set.

Summary for RQ$_1$. Over the 110 software projects considered in this study, we observed that 83% of the test classes

are affected by at least one test smell. Thus, we can conclude that test smells are highly diffused in the automatically generated test classes.

Summary for RQ_2. Three test smells, i.e. *Assertion Roulette*, *Eager Test*, and *Test Code Duplication*, frequently occur in test classes automatically generated by EvoSuite. This can be mainly due to the fact that automatic test case generation tools do not consider test code quality as one of the objectives to target in their process.

Summary for RQ_3. We found that all the test smells frequently co-occurs with *Assertion Roulette*. At the same time, we also observed some interesting co-occurrences between *Mystery Guest* and *Resource Optimism*, *Mystery Guest* and *Indirect Testing*, and between *Indirect Testing* and *Test Code Duplication*. We investigated the reasons behind these results, finding different strategies performed by EvoSuite during the process of test case generation that tend to increase the number and the co-existance of different test smells in the automatically generated JUnit classes.

Summary for RQ_4. We found that 6 test smells, i.e., *Assertion Roulette*, *Eager Test*, *Mystery Guest*, *Sensitive Equality*, *Resource Optimism*, and *Test Code Duplication*, have strong correlations with the size of the system, in terms of number of classes, number of JUnit classes, production class LOC, and JUnit class LOC. This means that, as the size of a system increases, automatic test case generation tools do not have appropriate mechanisms to deal with such complexity.

5. THREATS TO VALIDITY

This section discusses the threats that could affect the validity of our study.

As for threats related to the relationship between theory and observation (*construct validity*), the main threat in our study is about the way we detected test smells in the considered projects. In order to mitigate the problem, we rely on a test smell detection tool which has a very good precision (88%) and recall (100%), and that has been successfully used in previous work [6]. We further analyze the precision of the tool in our context through a manual validation of a statistically significant subset of test smells, finding that the precision is 75%.

As for *external validity* threats, one of the problems usually considered is related to the generalization of our findings. To produce highly reliable results, we conducted our study on a statistically significant sample of the projects available on *Sourceforge*, namely the *SF110* dataset originally produced by Fraser and Arcuri [15] with the specific goal of replicability of results. However, due to the limitations of the detection tool that we used, we limited the context of our study only to Java systems. Replications of our work on systems written in other languages are desirable.

6. CONCLUSION

In this paper we conducted an empirical investigation on the diffusion of test smells in the JUnit test classes automatically generated by EvoSuite [14] on 110 open source software projects from the SF110 Corpus of Classes [15]. Results indicate that (i) test smells are largely diffused, i.e., 83% of JUnit classes are affected by at least one test smell; (ii) the *Assertion Roulette* test smell is the most frequent one (contained in 54% of classes), followed by *Test Code Duplication* and *Eager Test* (contained in 33% and 29% of JUnit classes, respectively); (iii) all the test smells frequently co-occur with *Assertion Roulette*; (iv) three pairs of smells, namely *Mystery Guest* and *Resource Optimism*, *Mystery Guest* and *Indirect Testing*, and *Indirect Testing* and *Test Code Duplication* tend to co-occur quite frequently; and (v) all the test smells but *For Testers Only* and *Indirect Testing* have strong positive correlations with structural characteristics of a system, such as size and number of classes in the project. The results of our study provide two valuable findings for the research community:

Lesson 1. Current implementations of search based algorithms for automatic test case generation do not take into account the quality of the produced test classes. This implies the introduction of several design flaws, such as *Assertion Roulette* and *Eager Test* smells, that potentially have a huge impact on comprehensibility and maintainability [6], but can also have a negative impact on the effectiveness of test cases and on the other testing-related activities, such as the task to find faults [36].

Lesson 2. Automatic test case generation tools do not produce text fixtures during their computation, and this implies the introduction of several code clones in the resulting JUnit classes. This peculiarity can negatively affect the maintainability of test classes because of the effort needed for modifying all the clones in response to a change in the production code [49].

These observations represent the main input for our future research agenda, mainly focused on designing and developing new algorithms that, on the one hand, try to balance branch coverage criteria with test code quality and, on the other hand, try to automatically create text fixtures for test cases generated using existing tools.

7. REFERENCES

[1] M. Abbes, F. Khomh, Y.-G. Guéhéneuc, and 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*, pages 181–190. IEEE Computer Society, 2011.

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

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

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

[5] E. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo. The oracle problem in software testing: A survey. *Software Engineering, IEEE Transactions on*, 41(5):507–525, May 2015.

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

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

[8] M. Beller, G. Gousios, A. Panichella, and A. Zaidman. When, how, and why developers (do not) test in their ides. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 179–190. ACM, 2015.

[9] A. Chatzigeorgiou and 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)*, pages 106–115. IEEE, 2010.

[10] J. Cohen. *Statistical power analysis for the behavioral sciences*. Lawrence Earlbaum Associates, 2nd edition edition, 1988.

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

[12] J. Ferrer, F. Chicano, and E. Alba. Evolutionary algorithms for the multi-objective test data generation problem. *Software: Practice and Experience*, 42(11):1331–1362, 2012.

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

[14] G. Fraser and A. Arcuri. Evosuite: Automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 416–419, New York, NY, USA, 2011. ACM.

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

[16] M. Greiler, A. van Deursen, and M.-A. Storey. Automated detection of test fixture strategies and smells. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*, pages 322–331, March 2013.

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

[18] M. Harman, S. G. Kim, K. Lakhotia, P. McMinn, and 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*, pages 182–191. IEEE Computer Society, 2010.

[19] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 96–105, Washington, DC, USA, 2007. IEEE Computer Society.

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

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

[22] K. Lakhotia, M. Harman, and P. McMinn. A multi-objective approach to search-based test data generation. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, pages 1098–1105. ACM, 2007.

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

[24] A. Lozano, M. Wermelinger, and B. Nuseibeh. Assessing the impact of bad smells using historical information. In *Proceedings of the International workshop on Principles of Software Evolution (IWPSE)*, pages 31–34. ACM, 2007.

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

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

[27] G. Meszaros. *xUnit Test Patterns: Refactoring Test Code*. Addison Wesley, 2007.

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

[29] 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)*, page 15. IEEE, September 2005.

[30] R. Oliveto, F. Khomh, G. Antoniol, and 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)*, pages 248–251. IEEE, 2010.

[31] N. Oster and F. Saglietti. Automatic test data generation by multi-objective optimisation. In *SAFECOMP*, volume 4166, pages 426–438. Springer, 2006.

[32] C. Pacheco and 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, pages 815–816, New York, NY, USA, 2007. ACM.

[33] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, and 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)*, pages 101–110. IEEE, 2014.

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

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

[36] S. Panichella, A. Panichella, M. Beller, A. Zaidman, and S. Gall HC. The impact of test case summaries on bug fixing performance: An empirical investigation. In *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*, page to appear, 2016.

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

[38] G. H. Pinto and S. R. Vergilio. A multi-objective genetic algorithm to test data generation. In *Tools with Artificial Intelligence (ICTAI), 2010 22nd IEEE International Conference on*, volume 1, pages 129–134. IEEE, 2010.

[39] I. Prasetya, T. Vos, and A. Baars. Trace-based reflexive testing of oo programs with t2. In *Software Testing, Verification, and Validation, 2008 1st International Conference on*, pages 151–160, April 2008.

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

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

[42] J. M. Rojas, G. Fraser, and 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, pages 338–349, New York, NY, USA, 2015. ACM.

[43] A. Sakti, G. Pesant, and Y.-G. GueÌĄheÌĄneuc. Instance generator and problem representation to improve object oriented code coverage. *Software Engineering, IEEE Transactions on*, 41(3):294–313, March 2015.

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

[45] S. Shamshiri, R. Just, J. M. Rojas, G. Fraser, P. McMinn, and 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.

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

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

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

[49] R. J. Turver and M. Munro. An early impact analysis technique for software maintenance. *Journal of Software Maintenance: Research and Practice*, 6(1):35–52, 1994.

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

[51] B. Van Rompaey, B. Du Bois, S. Demeyer, and 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):800–817, Dec 2007.

[52] A. Yamashita and 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)*, pages 682–691. IEEE, 2013.

[53] A. F. Yamashita and L. Moonen. Do developers care about code smells? an exploratory survey. In *Proceedings of the Working Conference on Reverse Engineering (WCRE)*, pages 242–251. IEEE, 2013.