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

Fabio Palomba¹, Dario Di Nucci¹, Annibale Panichella², Rocco Oliveto³, Andrea De Lucia¹

¹University of Salerno, Italy — ²Delft University of Technology, The Netherlands — ³University 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

General Terms
Experimentation

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, developers do not test as they should and 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, 44]. 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, 46] 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:

RQ1: To what extent test smells are spread in automatically generated test classes?

RQ2: Which test smells occur more frequently in automatically generated test classes?

RQ3: Which test smells co-occur together?
RQ: Is there a relationship between the presence of test smells and the project characteristics?

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 *EcoSuite* that make possible the introduction of test smells during the generation process.

Structure of the paper. The paper is organized as follows. Section 2 overviews the background on test smells as well as the related literature, while Section 3 describes the design of our empirical study. In Section 4 we report and discuss the obtained results. Section 5 discusses the threats that could affect the validity of the results of our study, while Section 6 concludes the paper.

2. BACKGROUND AND RELATED WORK

As well as production code, also test code should be designed following good programming practices [45]. 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 poor design solutions to write tests, together with refactoring operations able to remove them. This catalogue takes into account different types of bad design choices 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).

Indirect Testing (IT): A test that checks the corresponding production class using methods of another class [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].

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].

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 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. [50]. 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.

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.
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, 43]. 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 `<classname><incremental-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. As an example, the test code shown in Listing 1 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].

| JSTerm0 = new JSTerm(); |
| JSTerm0.add((Object) ); |
| JSTerm0.matches(JSTerm0); |
| assertEquals( false , JSTerm0.isGround() ); |
| assertEquals( true , JSTerm0.isVariable( ) ); |

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

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:

**RQ1:** To what extent test smells are spread in automatically generated test cases? This research question aims at quantifying the presence of test smells in the test cases automatically generated using the EvoSuite tool.

**RQ2:** 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.

**RQ3:** Which test smells co-occur together? In this analysis we analyze which test smells co-occur in the test cases automatically generated using the EvoSuite tool.

**RQ4:** 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.

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 SourceForge1. Because of the number of

---

1http://sourceforge.net

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Projects</td>
<td>110</td>
</tr>
<tr>
<td>Number of Testable Classes</td>
<td>23,886</td>
</tr>
<tr>
<td>Lines of Code</td>
<td>6,628,619</td>
</tr>
<tr>
<td>Number of Java Files</td>
<td>27,997</td>
</tr>
</tbody>
</table>

Table 1: Characteristics of the SF110 dataset [15]
systems analyzed, a manual detection of the 8 test smells is prohibitively expensive. This is 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 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.

To answer RQ1 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 RQ2 we verified which are the test smells that occur more frequently in the systems of our study. As for RQ3, 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 ti we measured the percentage of times its presence in a JUnit class co-occurs with another test smell tj (i ≠ j). Formally, for each pair of test smells ti and tj we compute the percentage of co-occurrences of ti and tj using the formula shown below:

\[ \text{co-occurrence}_{ti, tj} = \frac{|t_i \land t_j|}{|t_i|} \]  

where where \(|t_i \land t_j|\) is the number of times ti co-occurs with tj, and \(|t_i|\) is the number of occurrences of ti. 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 ≤ ρ < 0.1, small correlation when 0.1 ≤ ρ < 0.3, medium correlation when 0.3 ≤ ρ < 0.5, and strong correlation when 0.5 ≤ ρ ≤ 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 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.

code

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

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

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
<table>
<thead>
<tr>
<th>Project</th>
<th>JUnit Tests</th>
<th>JUnit Tests With Test Smells</th>
<th>Assertion Roulette</th>
<th>Eager Test</th>
<th>Mystery Guest</th>
<th>Sensitive Equality</th>
<th>Resource Optimism</th>
<th>Indirect Testing</th>
<th>Test Code Duplication</th>
</tr>
</thead>
<tbody>
<tr>
<td>all</td>
<td>151 (100%)</td>
<td>508 (100%)</td>
<td>185 (100%)</td>
<td>1190 (31%)</td>
<td>254 (76%)</td>
<td>56 (50%)</td>
<td>30 (22%)</td>
<td>19 (25%)</td>
<td>6 (100%)</td>
</tr>
<tr>
<td>jcvi-javacommon</td>
<td>120 (100%)</td>
<td>122 (100%)</td>
<td>1190 (31%)</td>
<td>197 (25%)</td>
<td>9 (5%)</td>
<td>1 (100%)</td>
<td>1 (100%)</td>
<td>0 (0%)</td>
<td>1 (100%)</td>
</tr>
<tr>
<td>ifx-framework</td>
<td>88 (100%)</td>
<td>165 (100%)</td>
<td>1190 (31%)</td>
<td>197 (25%)</td>
<td>9 (5%)</td>
<td>1 (100%)</td>
<td>1 (100%)</td>
<td>0 (0%)</td>
<td>1 (100%)</td>
</tr>
<tr>
<td>javathena</td>
<td>50 (100%)</td>
<td>98 (100%)</td>
<td>1190 (31%)</td>
<td>197 (25%)</td>
<td>9 (5%)</td>
<td>1 (100%)</td>
<td>1 (100%)</td>
<td>0 (0%)</td>
<td>1 (100%)</td>
</tr>
<tr>
<td>io-project</td>
<td>58 (100%)</td>
<td>165 (100%)</td>
<td>1190 (31%)</td>
<td>197 (25%)</td>
<td>9 (5%)</td>
<td>1 (100%)</td>
<td>1 (100%)</td>
<td>0 (0%)</td>
<td>1 (100%)</td>
</tr>
<tr>
<td>dash-framework</td>
<td>119 (100%)</td>
<td>197 (100%)</td>
<td>1190 (31%)</td>
<td>197 (25%)</td>
<td>9 (5%)</td>
<td>1 (100%)</td>
<td>1 (100%)</td>
<td>0 (0%)</td>
<td>1 (100%)</td>
</tr>
<tr>
<td>noen</td>
<td>10 (100%)</td>
<td>10 (100%)</td>
<td>1190 (31%)</td>
<td>197 (25%)</td>
<td>9 (5%)</td>
<td>1 (100%)</td>
<td>1 (100%)</td>
<td>0 (0%)</td>
<td>1 (100%)</td>
</tr>
<tr>
<td>sugar</td>
<td>1 (100%)</td>
<td>1 (100%)</td>
<td>1190 (31%)</td>
<td>197 (25%)</td>
<td>9 (5%)</td>
<td>1 (100%)</td>
<td>1 (100%)</td>
<td>0 (0%)</td>
<td>1 (100%)</td>
</tr>
<tr>
<td>rif</td>
<td>11 (100%)</td>
<td>11 (100%)</td>
<td>1190 (31%)</td>
<td>197 (25%)</td>
<td>9 (5%)</td>
<td>1 (100%)</td>
<td>1 (100%)</td>
<td>0 (0%)</td>
<td>1 (100%)</td>
</tr>
<tr>
<td>nutzenportfolio</td>
<td>108 (100%)</td>
<td>1190 (31%)</td>
<td>1190 (31%)</td>
<td>197 (25%)</td>
<td>9 (5%)</td>
<td>1 (100%)</td>
<td>1 (100%)</td>
<td>0 (0%)</td>
<td>1 (100%)</td>
</tr>
<tr>
<td>schemaspy</td>
<td>108 (100%)</td>
<td>1190 (31%)</td>
<td>1190 (31%)</td>
<td>197 (25%)</td>
<td>9 (5%)</td>
<td>1 (100%)</td>
<td>1 (100%)</td>
<td>0 (0%)</td>
<td>1 (100%)</td>
</tr>
<tr>
<td>sbmlreader2</td>
<td>119 (100%)</td>
<td>1190 (31%)</td>
<td>1190 (31%)</td>
<td>197 (25%)</td>
<td>9 (5%)</td>
<td>1 (100%)</td>
<td>1 (100%)</td>
<td>0 (0%)</td>
<td>1 (100%)</td>
</tr>
<tr>
<td>xisemele</td>
<td>90 (100%)</td>
<td>1190 (31%)</td>
<td>1190 (31%)</td>
<td>197 (25%)</td>
<td>9 (5%)</td>
<td>1 (100%)</td>
<td>1 (100%)</td>
<td>0 (0%)</td>
<td>1 (100%)</td>
</tr>
<tr>
<td>jwbf</td>
<td>1 (100%)</td>
<td>1 (100%)</td>
<td>1190 (31%)</td>
<td>197 (25%)</td>
<td>9 (5%)</td>
<td>1 (100%)</td>
<td>1 (100%)</td>
<td>0 (0%)</td>
<td>1 (100%)</td>
</tr>
<tr>
<td>byuic</td>
<td>14 (100%)</td>
<td>98 (100%)</td>
<td>1190 (31%)</td>
<td>197 (25%)</td>
<td>9 (5%)</td>
<td>1 (100%)</td>
<td>1 (100%)</td>
<td>0 (0%)</td>
<td>1 (100%)</td>
</tr>
<tr>
<td>tullibee</td>
<td>1 (100%)</td>
<td>1 (100%)</td>
<td>1190 (31%)</td>
<td>197 (25%)</td>
<td>9 (5%)</td>
<td>1 (100%)</td>
<td>1 (100%)</td>
<td>0 (0%)</td>
<td>1 (100%)</td>
</tr>
<tr>
<td>firebird</td>
<td>1 (100%)</td>
<td>1 (100%)</td>
<td>1190 (31%)</td>
<td>197 (25%)</td>
<td>9 (5%)</td>
<td>1 (100%)</td>
<td>1 (100%)</td>
<td>0 (0%)</td>
<td>1 (100%)</td>
</tr>
<tr>
<td>water-simulator</td>
<td>1 (100%)</td>
<td>1190 (31%)</td>
<td>1190 (31%)</td>
<td>197 (25%)</td>
<td>9 (5%)</td>
<td>1 (100%)</td>
<td>1 (100%)</td>
<td>0 (0%)</td>
<td>1 (100%)</td>
</tr>
<tr>
<td>freemind</td>
<td>1 (100%)</td>
<td>1 (100%)</td>
<td>1190 (31%)</td>
<td>197 (25%)</td>
<td>9 (5%)</td>
<td>1 (100%)</td>
<td>1 (100%)</td>
<td>0 (0%)</td>
<td>1 (100%)</td>
</tr>
<tr>
<td>sweethome3d</td>
<td>71 (79%)</td>
<td>1190 (31%)</td>
<td>1190 (31%)</td>
<td>197 (25%)</td>
<td>9 (5%)</td>
<td>1 (100%)</td>
<td>1 (100%)</td>
<td>0 (0%)</td>
<td>1 (100%)</td>
</tr>
</tbody>
</table>

Table 3: Distribution of Test Smells in SF110 Corpus of Classes
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 JMap project, where the DocumentEvoSuiteTest JUnit class is affected by an Assertion Roulette instance.

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

```java
public void test2() throws Throwable {
    Document document0 = new Document("","");
    document0.procText.add((Character) 's');
    String string0 = document0.stringify();
    assertEquals("s",document0.stringify());
    assertEquals("s", string0);
    }}
```

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

```java
public void test2() throws Throwable {
    SecureMessageServiceClientImpl secureMessageServiceClientImpl0 = new SecureMessageServiceClientImpl("\\A7ZXfnoaf\F",0, 0, (RSAPublicKey) null);
    SecureMessageServiceClientMessageImpl secureMessageServiceClientMessageImpl0 = ( SecureMessageServiceClientMessageImpl) new SecureMessageServiceClientMessageImpl(\n        ( Map) null, (Object) "\\111111111111111111", \n        "\\A7ZXfnoaf\F\"\",\n    );
    assertEquals(false, boolean0);
    }
```

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 class 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.

```java
public void test8() throws Throwable {
    GenericPropertiesCreator genericPropertiesCreator0 = new GenericPropertiesCreator();
    boolean boolean0 = genericPropertiesCreator0.isValidateClassName("\\s\\(Un-kZXF155WyYk3V\\)\"";)
    assertEquals(false, genericPropertiesCreator0.getExplicitPropsFile());
    assertEquals("\\weka/gui/GenericObjectEditor.properties",
        genericPropertiesCreator0.getInputFilename());
    assertEquals(false, boolean0);
}
```

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 test 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: Co-occurrences among Test Smells in the SF110 Dataset

<table>
<thead>
<tr>
<th></th>
<th>AR</th>
<th>ET</th>
<th>MG</th>
<th>SE</th>
<th>RO</th>
<th>IT</th>
<th>TCD</th>
</tr>
</thead>
<tbody>
<tr>
<td>AR</td>
<td>0.49</td>
<td>0.07</td>
<td>0.27</td>
<td>0.02</td>
<td>0.01</td>
<td>0.48</td>
<td></td>
</tr>
<tr>
<td>ET</td>
<td></td>
<td>0.10</td>
<td>0.24</td>
<td>0.04</td>
<td>0.01</td>
<td>0.50</td>
<td></td>
</tr>
<tr>
<td>MG</td>
<td></td>
<td></td>
<td>0.15</td>
<td>0.36</td>
<td>0.06</td>
<td>0.62</td>
<td></td>
</tr>
<tr>
<td>SE</td>
<td>0.99</td>
<td>0.45</td>
<td>0.05</td>
<td>0.02</td>
<td>0.01</td>
<td>0.67</td>
<td></td>
</tr>
<tr>
<td>RO</td>
<td>0.78</td>
<td>0.58</td>
<td>1.00</td>
<td>0.12</td>
<td>-</td>
<td>0.45</td>
<td></td>
</tr>
<tr>
<td>IT</td>
<td>0.73</td>
<td>0.59</td>
<td>1.00</td>
<td>0.16</td>
<td>0.16</td>
<td>0.80</td>
<td></td>
</tr>
<tr>
<td>TCD</td>
<td>0.80</td>
<td>0.45</td>
<td>0.10</td>
<td>0.32</td>
<td>0.03</td>
<td>0.01</td>
<td></td>
</tr>
</tbody>
</table>

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, intrinsically related to the Assertion Roulette smell. Indeed, if a test case contains several assertions, then, it would likely contain an 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 all the times 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. 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.

```java
public void test3() throws Throwable {
    SweetHome3D sweetHome3D = new SweetHome3D();
    HomeRecorder.Type homeRecorder_Type0 = HomeRecorder.Type.DEFAULT;
    HomeFileRecorder homeFileRecorder0 = (HomeFileRecorder) sweetHome3D.getHomeRecorder(homeRecorder_Type0);
    assertNotSame(homeFileRecorder0);
}
```

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.

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

<table>
<thead>
<tr>
<th></th>
<th># Classes</th>
<th># JUnit Classes</th>
<th>LOC</th>
<th>JUnit LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assertion Roulette</td>
<td>0.98</td>
<td>0.95</td>
<td>0.94</td>
<td>0.98</td>
</tr>
<tr>
<td>Eager Test</td>
<td>0.94</td>
<td>0.80</td>
<td>0.98</td>
<td>0.98</td>
</tr>
<tr>
<td>Mystery Guest</td>
<td>0.78</td>
<td>0.58</td>
<td>0.79</td>
<td>0.74</td>
</tr>
<tr>
<td>Sensitive Equality</td>
<td>0.57</td>
<td>0.80</td>
<td>0.12</td>
<td>0.45</td>
</tr>
<tr>
<td>Resource Optimism</td>
<td>0.58</td>
<td>0.44</td>
<td>0.56</td>
<td>0.55</td>
</tr>
<tr>
<td>Indirect Testing</td>
<td>0.40</td>
<td>0.19</td>
<td>0.36</td>
<td>0.24</td>
</tr>
<tr>
<td>Test Code Duplication</td>
<td>0.95</td>
<td>0.97</td>
<td>0.90</td>
<td>0.96</td>
</tr>
</tbody>
</table>

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, 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 RQ1. Over the 110 software projects considered in this study, we observed that 83% of them 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 RQ2. 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 RQ3. We found that all the test smells frequently co-occur 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-existence of different test smells in the automatically generated JUnit classes.

Summary for RQ4. 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]. However, we cannot exclude the presence of false positives/negatives in the set of detected test smells.

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 therefore 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 test 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 test fixtures for test cases generated using existing tools.

7. REFERENCES


